

Zusammenfassung vom 12.04.2022

Numerische Ableitung

Taylor-Entwicklung erlaubt Ableitungen aus Differenzenquotienten zu bestimmen.

Optimale Schrittweite h hängt von Genauigkeit der Gleitkommadarstellung ab.

$$f'(x_i) \approx \frac{f_{i+1} - f_i}{h} \approx \frac{f_i - f_{i-1}}{h}$$

$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2h} + \mathcal{O}(h^2)$$

Numerische Integration: Trapezregel

Integration stückweise in Teilintervallen

Taylor-Entwicklung innerhalb der Intervalle: einfache Integrationsmethode betrachteten Trapezregel

$$\int_a^b dx \ f(x) = \frac{h}{2} f_0 + h (f_1 + \cdots + f_{N-2}) + \frac{h}{2} f_{N-1} + \underbrace{(N-1) \cdot \mathcal{O}(h^3)}_{\mathcal{O}(h^2)}$$

```
/* Datei: beispiel-3.2.c    Datum: 12.4.2016 */
```

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

Definition der zu integrierenden Funktion

```
/* Definition der zu integrierenden Funktion */
```

```
double f(double x)
{
    return exp(x);
}
```

Funktion die Stützstellen und Gewichte festlegt.

```
/* Routine, die Gitterpunkte und Gewichte fuer ein Intervall [a,b] festlegt */
```

```
void trapez(int n, double a, double b, double *xp, double *wp)
```

```
/* n legt die Anzahl der Stuetzstellen fest.
```

```
a,b sind "normale" double Parameter
```

```
xp und wp sind Zeiger(Pointer) auf ein double Feld,
es wird die Adresse des Feldes gespeichert !!! */
```

```
{  
int i;  
double h;
```

Adresse des Feldes xp und wp ist Parameter!

n-1 Umwandlung in “double” (type casting)

```
h=(b-a)/(double)(n-1); /* Berechne Schrittweite */
```

```
for(i=1;i<n-1;i++)
```

```
{  
    xp[i]=a+i*h; /* xp = Anfangsadresse des Feldes */  
    wp[i]=h; /* xp[i] = Nehme die Speicherstelle, */  
              /* die i Speicherstellen weiter liegt */  
}
```

for-Schleife legt “normale” x und w fest

```
xp[0]=a; /* Lege Punkte und Gewichte am Rand fest */
```

```
wp[0]=h/2.0;
```

```
xp[n-1]=b;
```

```
wp[n-1]=h/2.0;
```

Spezialfall der Randpunkte

keine Rückgabe, die Felder xp und wp wurden verändert

```

int main()
{
    double a,b;          /* Intervallgrenzen */
    int i,n;             /* Schleifenvariable, Anzahl der Stuetzstellen */
    double exact,diff,sum; /* Variablen, um Ergebnis zu speichern */
    double *x,*w;         /* Zeiger auf Speicherplaetze, d.h. Adressen */

    printf("Bitte geben Sie a,b und n ein: "); /* Eingabe */
    scanf("%lf %lf %d",&a,&b,&n);

```

x und w sind Adresse (=Zeiger) auf Felder

**Eingabe: a,b und n sollen verändert werden:
Speicherort=Adresse ist nötig und
wird mit “&” Operator bestimmt**

```

x=(double *)malloc(n*sizeof(double));
w=(double *)malloc(n*sizeof(double));

```

**malloc reserviert Speicherplatz und gibt Adresse
dieses Speicherplatzes zurück
Umwandlung in Zeiger auf double erforderlich**

```

trapez(n,a,b,x,w); /* uebergibt
/* Adressen */

```

Aufruf von Trapez definiert Stützstellen und Gewichte

```

/* Gitterpunkte und Gewichte sind nun bei x und w gespeichert */
/* Diese kann man fuer beliebige Funktionen benutzen */

```

```

sum=0.0;           /* Bestimmung eines Integrals mit den Gitter und Gewichten */

```

```

for(i=0;i<n;i++)
{
    sum+=f(x[i])*w[i]; /* "+=" Operator summiert f(xi)*w(xi) auf Summe auf */
}

```

```

exact=exp(b)-exp(a);
diff=fabs(sum-exact);

```

```

printf("N      trapez      exact      diff \n\n");
printf("%d      %15.6e      %15.6e      %15.6e \n",n,sum,exact,diff);

```

```

free(x); /* Speicherbereich wieder freigeben */
free(w); /* ("deallozieren") */
}
/* Ergebnis: fuer a b n = 0.0 1.0 30
   N      trapez      exact      diff
   30     1.7184520869e+00     1.7182818285e+00     1.7025840042e-04
*/

```

Bestimmung des Integrals

**Ausgabe des Ergebnisses, des exakten Ergebnisse
und des Fehlers**

| N | trapez | exact | diff |
|----|------------------|------------------|------------------|
| 30 | 1.7184520869e+00 | 1.7182818285e+00 | 1.7025840042e-04 |

Analog können Integrationsregeln höherer Ordnung hergeleitet werden

Simpsons-Regel  $\mathcal{O}(h^4)$ (oft verwendet)

Bodes-Regel  $\mathcal{O}(h^6)$

Wegen der Unstabilität bei sehr hohen Ordnungen
(auch weil positive/negative Gewichte auftreten) belässt man es meist bei niedrigen Ordnungen.

Später: weitere Verfeinerungen und Integrationsregel mit nicht-äquidistanten Punkten

Nullstellensuche

Problemstellung: Finde Nullstelle einer Funktion f im Intervall $[a,b]$

Wir nehmen an, dass f genau eine Nullstelle in $[a,b]$ hat, stetig ist und $f(a) \cdot f(b) < 0$

In diesem Fall kann man das **Bisektionsverfahren** formulieren

1. Starte bei $x_0 = a$ und $x_1 = b$

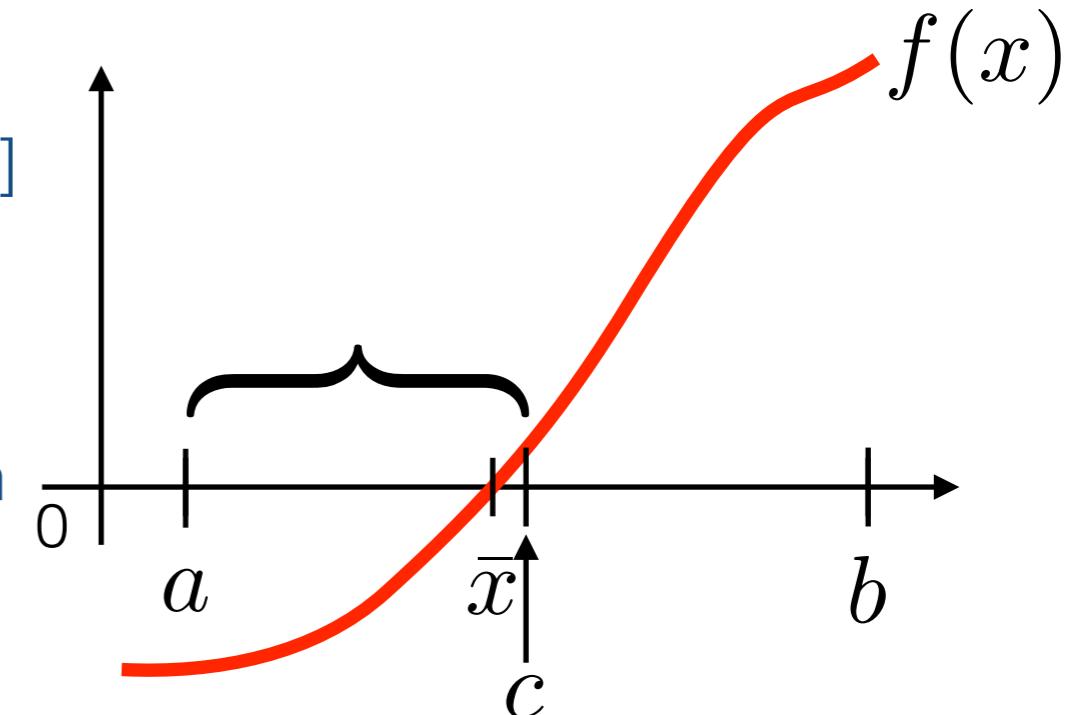
2. Finde nächste Approximation durch Mittelung

$$\tilde{x} = \frac{x_i + x_{i-1}}{2}$$

3. Nullstelle liegt in Intervall

$$[x_{i-1}, \tilde{x}] \quad \text{für} \quad f(\tilde{x}) \cdot f(x_{i-1}) < 0 \quad \rightarrow \quad x_{i+1} = \tilde{x} \quad \text{und} \quad x_i = x_{i-1}$$

$$[\tilde{x}, x_i] \quad \text{für} \quad f(\tilde{x}) \cdot f(x_i) < 0 \quad \rightarrow \quad x_{i+1} = x_i \quad \text{und} \quad x_i = \tilde{x}$$



Damit haben wir eine Iterationsvorschrift, die die Nullstelle lokalisiert.

Die Genauigkeit ist durch die Intervalllänge nach n Schritten gegeben $\Delta = \frac{b-a}{2^n}$

$$\Rightarrow n = -\frac{\ln(\Delta/(b-a))}{\ln 2}$$

Bei einer Anfangsintervalllänge von 1 und einer gewünschten Genauigkeit von 10^{-6} benötigt man

$$n \approx 20 \quad [\Delta = 10^{-6}, (b-a) = 1]$$

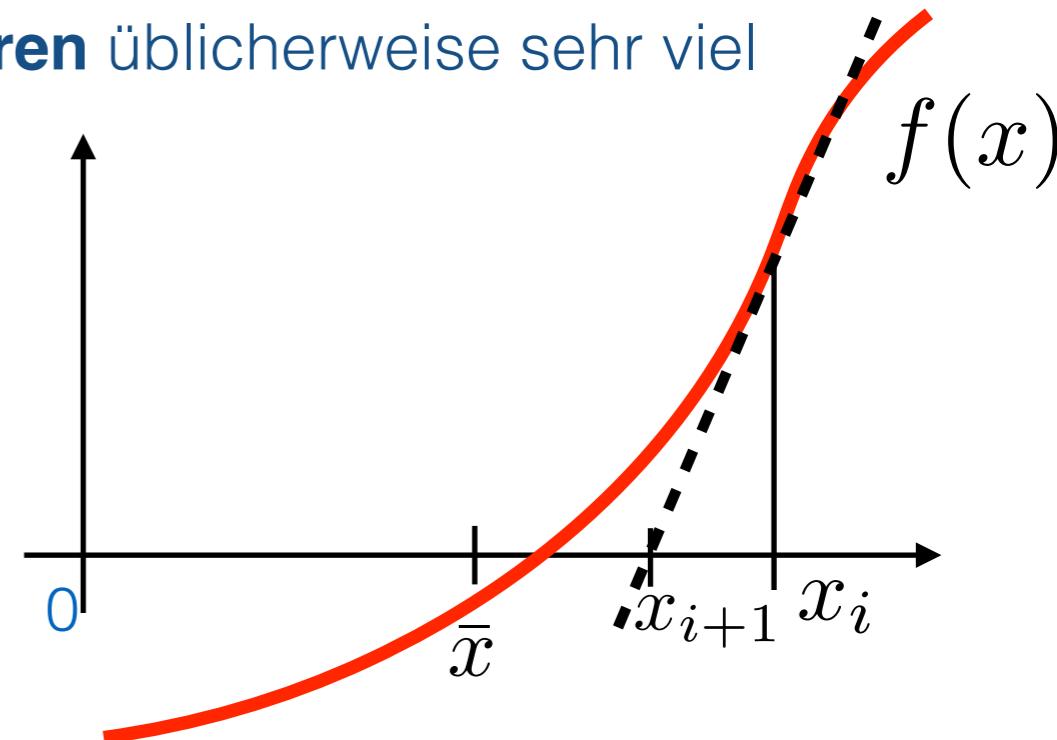
Schritte.

Für differenzierbare Funktionen ist das **Newton-Verfahren** üblicherweise sehr viel effizienter

Nutzen lineare Approximation an die Funktion

$$f(x) = f(x_i) + (x - x_i) f'(x_i) + \dots = 0$$

→ $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$



In der Regel konvergiert das **Newton-Verfahren** wesentlich schneller.

Aber:

- unstabil in der Nähe lokaler Extrema ($f'(x_i) \approx 0$)
- $f'(x)$ muss bekannt sein

Falls die Ableitung nicht analytisch bekannt ist, kann man sie durch eine numerische Ableitung ersetzen. Das wird üblicherweise **Sekantenverfahren** genannt.

Für die Ableitung werden Funktionswerte aus den vorhergehenden Schritten verwendet

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

Eine neue Näherung erhält man dann durch

$$x_{i+1} = x_i - f(x_i) \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

Das Verfahren benötigt wie das Bisektionsverfahren zwei Startwerte x_0 und x_1 .

Im folgenden ein Beispiel für eine Implementierung des Sekantenverfahrens.

```
/* Datei: beispiel-3.3.c  Datum: 14.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Routine fuer Nullstellensuche mit dem Sekantenverfahren */
double secant(double x1, double x2, double (*func)(double), int *schritt)
/* x1,x2  Startwerte
   func    ist die "Referenz" auf eine Funktion mit einem double Parameter,
           die double zurueckgibt (Referenz = Adresse der Funktion)
   schritt ist auch Referenz: Veraenderungen an der Variable wirken sich auf
           das aufrufende Programm aus !!! */
{
    const double tol=1e-12; /* geforderte Genauigkeit als Konstante */
    double xn;              /* neuer Schaeztwert */

    *schritt=0; /* noch kein Schritt */

    do
    {
        /* naechster Schaeztwert x1,x2 -> xn*/
        xn=x2-func(x2)*(x2-x1)/(func(x2)-func(x1));
        x1=x2;      /* bereite den naechsten Schritt vor: x2 -> x1 */
        x2=xn;      /*                      xn -> x2 */

        (*schritt)++; /* Schritte=Schritte+1 */
    }
    while(fabs(x2-x1)>tol); /* solange Genauigkeitsziel nicht erreicht */

    return xn; /* Gebe Nullstelle zurueck */
}

double f(double x)      /* eine Beispielfunktion */
{                      /* beide sind double Funktionen */
    return x*log(x)-x; /* mit einem double Parameter */
}
```

Funktion secant sucht Nullstelle von func

Nullstelle der Funktion f soll gesucht werden

```

/* Datei: beispiel-3.3.c  Datum: 14.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Routine fuer Nullstellensuche mit dem Sekantenverfahren */,
double secant(double x1, double x2, double (*func)(double), int *schritt)
/* x1,x2  Startwerte
   func    ist die "Referenz" auf eine Funktion mit einem double Parameter,
           die double zurueckgibt (Referenz = Adresse der Funktion)
   schritt ist auch Referenz: Veraenderungen an der Variable wirken sich auf
           das aufrufende Programm aus !!! */
{
    const double tol=1e-12; /* geforderte Genauigkeit als Konstante */
    double xn;              /* neuer Schaeztwert */

    *schritt=0; /* noch kein Schritt */

    do
    {
        /* naechster Schaeztwert x1,x2 -> xn*/
        xn=x2-func(x2)*(x2-x1)/(func(x2)-func(x1));
        x1=x2;      /* bereite den naechsten Schritt vor: x2 -> x1 */
        x2=xn;      /* xn -> x2 */

        (*schritt)++; /* Schritte=Schritte+1 */
    }
    while(fabs(x2-x1)>tol); /* solange Genauigkeitsziel ni

    return xn; /* Gebe Nullstelle zurueck */
}

double f(double x)      /* eine Beispielfunktion */
{                      /* beide sind double Funktionen */
    return x*log(x)-x; /* mit einem double Parameter */
}

```

Startwerte der Iteration: x1 und x2

Funktion ist ebenfalls Parameter

Schritt wird als “Referenz” uebergeben

* Operator: bearbeite Variable an in
schritt gespeicherter Adresse

Iteration nach dem Sekantenverfahren

Abbruchbedingung auf Basis des
Abstandes der Schätzwerthe

secant erhält als Wert die
gefundenen Nullstelle

```
int main()
{
```

```
    double a,b;          /* fuer die Startwerte */
    double exact,diff,res; /* fuer die Ergebnisse */
    int n;                /* Anzahl der Schritte */
```

```
    printf("Bitte geben Sie a,b ein: ");
    scanf("%lf %lf",&a,&b);
```

```
    printf("      sekant      exakt      diff\n\n");
```

/ Suche Nullstelle und speichere Ergebnis in "res"*

Referenzdefinitionen bei "secant":

Es werden automatisch die Adressen der Objekte f und n uebergeben

*n wird veraendert und enthaelt die Anzahl der Schritte nach Aufruf !!! */*

```
    res=secant(a,b,&f,&n);
```

```
    exact=exp(1.0);           /* Vergleich mit exaktem Ergebnis */
```

```
    diff=fabs(res-exact);
```

```
    printf("%15d  %15.6e  %15.6e  %15.6e \n",n,res,exact,diff);
```

```
}
```

/ Ergebnis:*

Bitte geben Sie a,b ein: 1 2

| | | |
|--------|-------|------|
| sekant | exakt | diff |
|--------|-------|------|

| | | | |
|---|----------------|----------------|----------------|
| 8 | 2.71828183e+00 | 2.71828183e+00 | 4.44089210e-16 |
|---|----------------|----------------|----------------|

Beispielhauptprogramm wendet die Routine an

Argumente von secant:

a,b wie üblich “call by value”

&f = Adresse der Funktion f

(die genau ein double Argument hat)

&n Adresse der Variabel n

Genauigkeit nach 8 Iterationen ist deutlich besser als mit dem Bisektionsverfahren

Fortsetzung: numerische Integration: Romberg Verfahren

Wir hatten bereits eine einfache Integrationsregel, die Trapezregel, kennengelernt. Hier schauen wir uns die Fehlerabschätzung genauer an und formulieren auf der Basis eine verbesserte Regel, die leicht für ein **adaptives Verfahren** verwendet werden kann.

→ Romberg Integration

Nach der Trapezregel gilt:

$$\int_a^b dx f(x) = \frac{h}{2} f_0 + h (f_1 + \cdots + f_{N-2}) + \frac{h}{2} f_{N-1} + \mathcal{O}(h^2) = T(h) + \mathcal{O}(h^2)$$

Schauen wir uns den Fehler pro Intervall $[x_i, x_{i+1}]$ noch mal genauer an.
Wegen der Taylor-Entwicklung um die Intervallgrenzen

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

$$f(x) = f(x_{i+1}) + (x - x_{i+1})f'(x_{i+1}) + \frac{1}{2}(x - x_{i+1})^2 f''(x_{i+1}) + \cdots$$

Findet man für ein Intervall

$$\int_{x_i}^{x_{i+1}} dx f(x) = \frac{h}{2} f_i + \frac{h}{2} f_{i+1} + \frac{h^2}{4} (f'(x_i) - f'(x_{i+1})) + \frac{h^3}{12} (f''(x_i) + f''(x_{i+1})) + \cdots$$

obere/untere Grenze

Bei Summation der Intervalle erhält man die Trapezregel und Korrekturterme, die **gerade in h** sind

$$\int_a^b dx f(x) = T(h) - \frac{h^2}{4} (f'(b) - f'(a)) + \frac{h^3}{12} \frac{2}{h} (f'(b) - f'(a)) \quad \dots$$

numerische Näherung von
 $f''(x_i) + f''(x_{i+1}) = 2 \frac{f'(x_{i+1}) - f'(x_i)}{h}$

$$\int_a^b dx f(x) = T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k})$$

$$\implies T(h) \approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m}$$

wegen Mittelung nur gerade Terme! “Romberg Folge”

Wir können $T(h)$ für $m+1$ h_j $j=0\dots m$ ausrechnen.

$$h_0 = b - a$$

$$h_1 = \frac{b - a}{2}$$

Damit kann man die Koeffizienten τ_i $i=0\dots m$ eindeutig bestimmen:

$$\begin{array}{c} \vdots \\ h_m = \frac{h_{m-1}}{2} \end{array}$$

$$\tilde{T}_{mm}(h_j^2) = \tau_0 + \tau_1 h_j^2 + \dots + \tau_m h_j^{2m} = T(h_j)$$

Die Idee: Das Polynom $\tilde{T}_{mm}(h^2)$ **extrapolieren** für $h = 0$ $\lim_{h \rightarrow 0} \tilde{T}_{mm}(h^2) = \tau_0$

Direkte Lösung des Gleichungssystems ist langwierig. Schneller ist das sogenannte **Neville Schema**, um das Polynom $\tilde{T}_{mm}(x)$ zu finden.

Ziel: finde ein Polynom vom Grad m , dass $m + 1$ Punkte $(x_j, y_j) \ j = 0, \dots, m$ beschreibt.

Wir definieren dazu Polynome $\tilde{T}_{ik}(x)$ vom Grad k mit $\tilde{T}_{ik}(x_j) = y_j$ für $j = i - k, \dots, i$.

1. $k = 0$ $\tilde{T}_{i0}(x) = y_i$ erfüllt die Bedingung ✓

2. $k \neq 0$ wenn $\tilde{T}_{ik-1}(x)$ die Bedingung erfüllt gilt das auch für das Polynom vom Grad k

$$T_{ik}(x) = \frac{(x - x_{i-k})\tilde{T}_{ik-1}(x) - (x - x_i)\tilde{T}_{i-1k-1}(x)}{x_i - x_{i-k}}$$

denn:

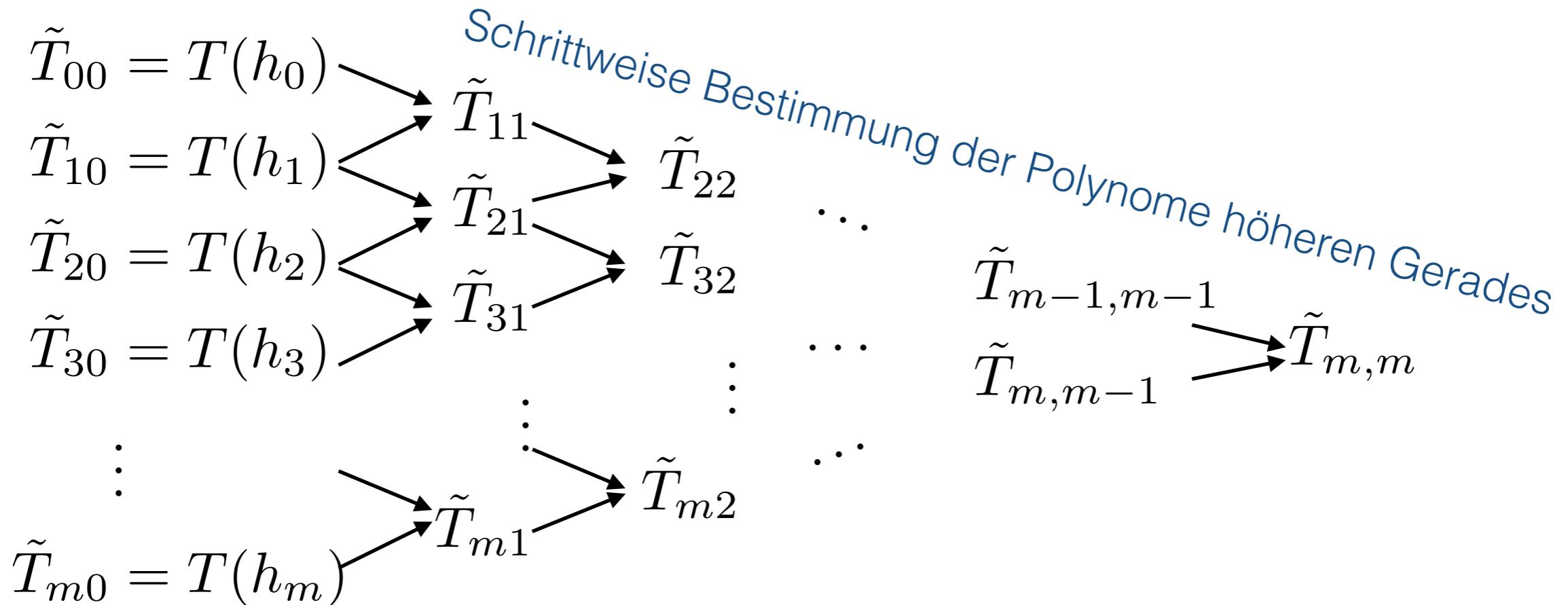
$$T_{ik}(x_{i-k}) = \frac{(x_{i-k} - x_{i-k})\tilde{T}_{ik-1}(x_{i-k}) - (x_{i-k} - x_i)\tilde{T}_{i-1k-1}(x_{i-k})}{x_i - x_{i-k}} = \tilde{T}_{i-1k-1}(x_{i-k}) = y_{i-k}$$

$$T_{ik}(x_i) = \frac{(x_i - x_{i-k})\tilde{T}_{ik-1}(x_i) - (x_i - x_i)\tilde{T}_{i-1k-1}(x_i)}{x_i - x_{i-k}} = \tilde{T}_{i-1k-1}(x_i) = y_i$$

$$T_{ik}(x_j) = \frac{(x_j - x_{i-k})\overbrace{\tilde{T}_{ik-1}(x_j)}^{y_j} - (x_j - x_i)\overbrace{\tilde{T}_{i-1k-1}(x_j)}^{y_j}}{x_i - x_{i-k}} = y_j \quad \text{für } j = i - k + 1, \dots, i - 1$$

Anwendung auf den Fall der Romberg Integration ($x_j = h_j^2, y_j = T(h_j)$)

Start der Rekursion



$\tilde{T}_{ik}(x)$ wird definiert für $i = k, \dots, m$!

Für die Implementierung ist es günstig aus ik einen **eindeutigen Index** zu erzeugen

Anzahl von Termen bis $k - 1$:
$$\sum_{l=0}^{k-1} m - l + 1 = k(m + 1) - \frac{k(k - 1)}{2}$$

deswegen wähle den Index:
$$i - k + k(m + 1) - \frac{k(k - 1)}{2}$$

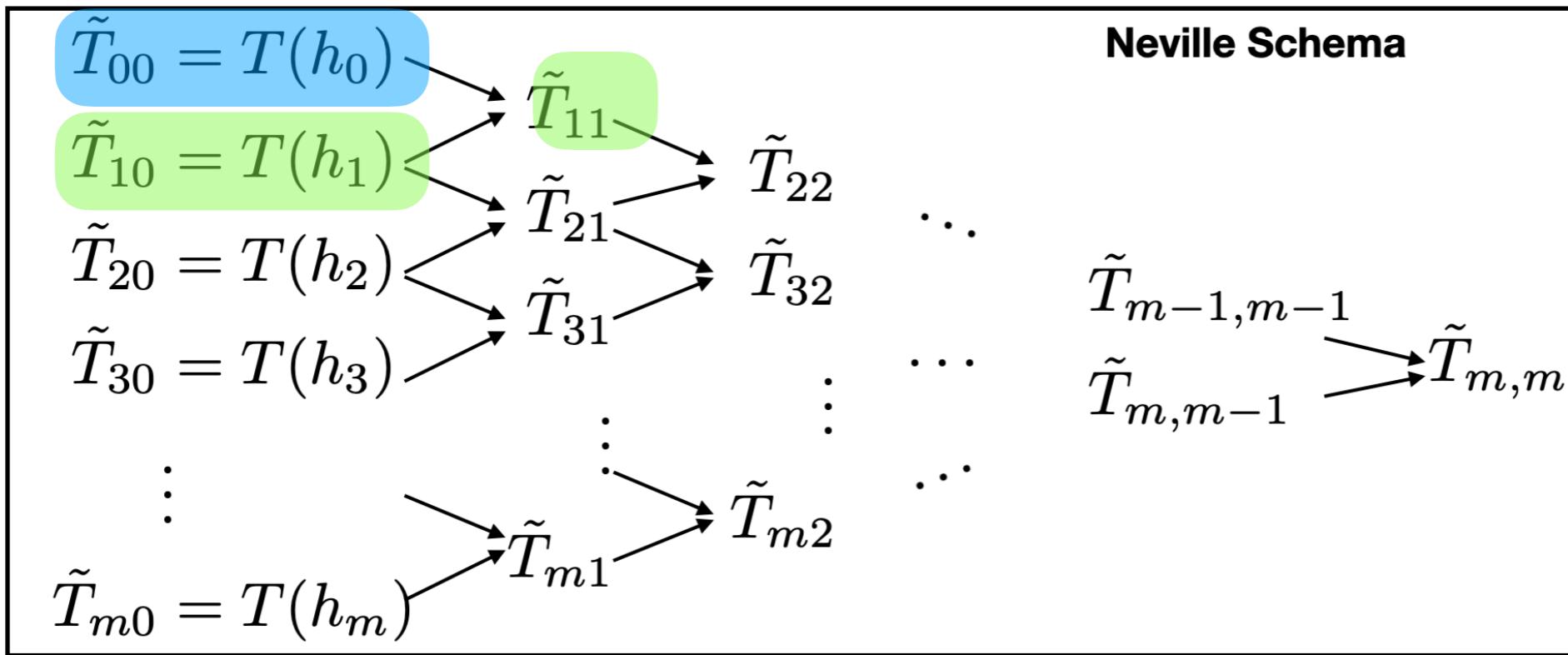
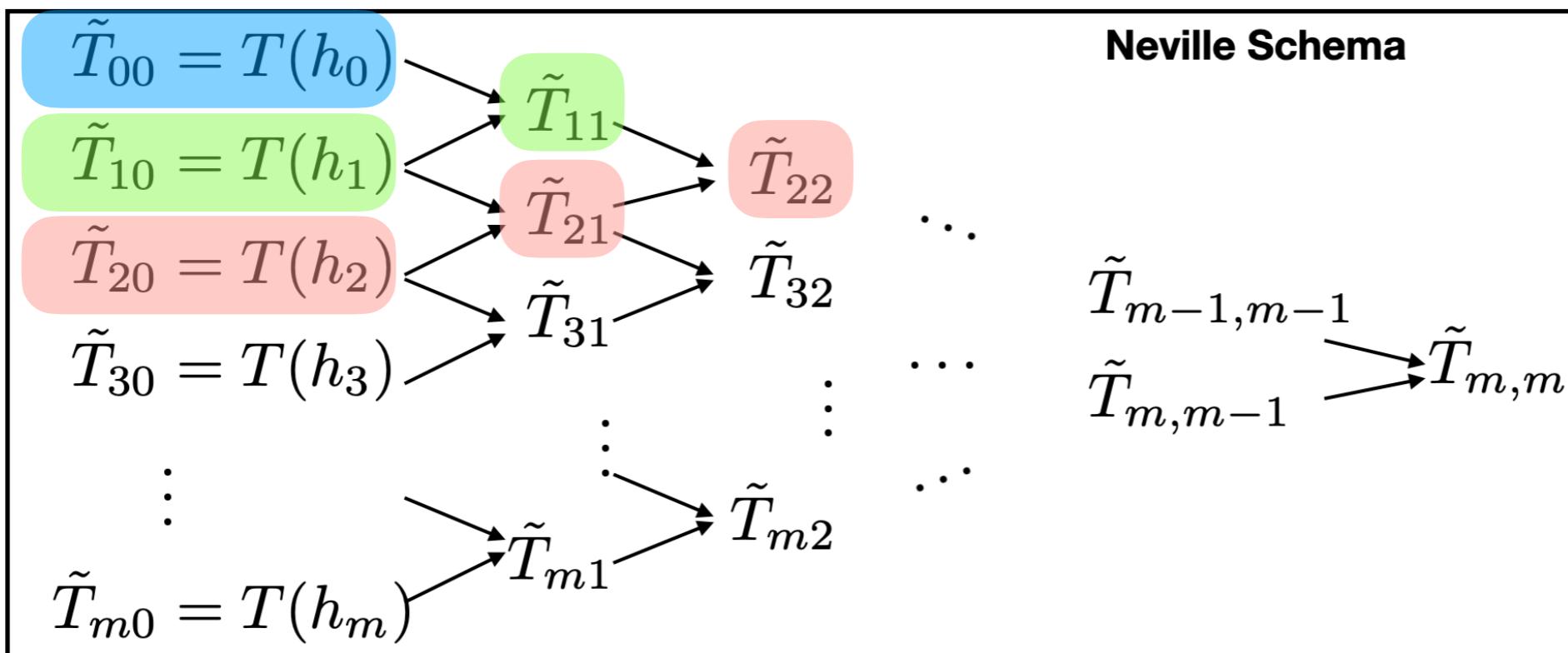
Verlauf der Romberg Integration:

- Berechnen Sie $T(h_j)$ ($= \tilde{T}_{j,0}(h_j^2)$) für $j = 1, \dots, m$ mit Trapezregel
- Bestimmen Sie die Polynomwerte $\tilde{T}_{ij}(0)$
 - Benutzen Sie $x=0$ in der Rekursionsformel (Neville Schema)
- $\tilde{T}_{m,m}(0)$ ist die gesuchte Annäherung unseres Integrals
- Wir stellen Sie die Schritte oben in eine **for**-Schleife
 - Die Auswertung mit ***h=0*** ergibt dann jeweils eine **Extrapolation zum gesuchten Integral**.
 - Wir können **abbrechen**, wenn eine gewünschte Genauigkeit erreicht wurde.
(= Terme höhere Ordnung tragen weniger als notwendig bei)

Für die Romberg Integration benötigen wir nur $x = 0$!

$$T_{ik}(0) = \frac{-x_{i-k} \tilde{T}_{ik-1}(0) + x_i \tilde{T}_{i-1k-1}(0)}{x_i - x_{i-k}}$$

Für jedes **neue *h*** kann man eine Ordnung höher gehen ohne jedesmal alles komplett neu zu berechnen (im Beispiel **nicht** realisiert, reduziert Rechenaufwand um Faktor 1/2)


 $m = 1$

 $m = 2$
 $\bullet \bullet \bullet$

```
/* Datei: beispiel-3.4.c      Datum: 19.4.2016 */  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>
```

```
int mmax=10; /* definiere globale Variable mit maximaler Anzahl der Schritte */
```

```
/* Definition der zu integrierenden Funktion */
```

```
double f(double x)  
{  
    return exp(x);  
}
```

```
/* Funktion die Trapezsumme mit n Funktionspunkten bestimmt fuer Integralgrenzen a,b und Funktion func */
```

```
double T(int n,double a, double b, double (*func)(double))  
{ double sum,h;  
    int i;  
  
    h=(b-a)/(n-1);  
  
    sum=0.5*func(a)+0.5*func(b);  
  
    for(i=1;i<n-1;i++)  
        { sum+=func(a+h*i); }  
  
    return h*sum; }
```

```
/* Funktion, die eindeutigen Index definiert von ik -> index_ik fuer i>=k und i,k<=mmax */  
int index_ik(int i,int k)  
{ return (i-k + k*(mmax+1)-k*(k-1)/2); }
```

**mmax ist globale Variable
wird in index_ik und romberg
benutzt**

Funktion, die zu integrieren ist.

Funktion die Integral mit Trapezregel berechnet

**index Funktion, die i und k auf
index_ik= $i-k+k^*(mmax+1)-k^*(k-1)/2$
abbildet
(eindeutig, zum Speichern vom T_{ik})**

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchführt.
n0 Anzahl der Stützstellen im ersten Schritt, maximale Anzahl der benutzten Stützstellen
a, b die Integralgrenzen und f die zu integrierende Funktion */
double romberg(int *n0, double a, double b, double (*func)(double), double eps)
{ int k,m,n; /* fuer Indizes */
  double *h; /* Schrittweiten fuer j=0,...,mmax */
  double *Tsum; /* Trapezsummen fuer j=0,...,mmax */
  double *tildeT; /* Neville-Schema tilde T_{jk} bei h=0 */
  double result;
  h=(double *)malloc((mmax+1)*sizeof(double)); /* Speicher fuer h in Schritt m */
  Tsum=(double *)malloc((mmax+1)*sizeof(double)); /* und T fuer diese h */
  tildeT=(double *)malloc(((mmax+1)*mmax)/2*sizeof(double)); /* Speicher fuer Neville Schema */
  h[0]=(b-a)/(double)(*n0-1);
  n=*n0;
  Tsum[0]=T(n,a,b,func);
  tildeT[index_ik(0,0)]=Tsum[0];

  for(m=1;m<=mmax;m++)
  { h[m]=h[m-1]/2; /* Trapezsumme fuer halbiertes h */
    n=2*n;
    Tsum[m]=T(n,a,b,func);
    tildeT[index_ik(m,0)]=Tsum[m]; /* fuer i=m und k=0 */

    for(k=1;k<=m;k++) /* generate tildeT i=m k=1,...,m */
    { tildeT[index_ik(m,k)]=-h[m-k]*h[m-k]/(h[m]*h[m]-h[m-k]*h[m-k])*tildeT[index_ik(m,k-1)]
      + h[m] *h[m] /(h[m]*h[m]-h[m-k]*h[m-k])*tildeT[index_ik(m-1,k-1)];
    }

    printf("%5d %15.6e %15.6e \n",m,tildeT[index_ik(m,m)],Tsum[m]); /* Ausdruck um Konvergenz
    if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; zu beobachten
    } Abruch mit "break"

  *n0=n;
  result=tildeT[index_ik(m,m)];
  free(tildeT); free(Tsum); free(h);
  return result; }

```

Deklarationen:
Felder für $h, T(h), T_{ik}(0)$

**Alloziere Speicher
für $h, T(h), T_{ik}(0)$**

**Vorbereitung für $m=0$
für $h=h_0, T(h_0), T_{00}(0)$**

**Gehe zu $m>0$
bestimme $T(h/2), T_{m0}(0)$**

bestimme $T_{mk}(0)$

**Ausdruck um Konvergenz
zu beobachten
Abruch mit "break"**

dealloziere den Speicher für $h, T(h)$ und $T_{ik}(0)$

```

int main()
{
    double a,b;          /* Intervallgrenzen */
    int n;               /* Stuetzstellen am Anfang */
    double exact,diff,sum; /* Variablen, um Ergebnis zu speichern */

    printf("Bitte geben Sie a,b und n ein: "); /* Eingabe der Parameter */
    scanf("%lf %lf %d",&a,&b,&n);

    sum=romberg(&n,a,b,&f,1.0E-6); /* uebergibt n = Anzahl der Punkte beim Start, a,b Intervallgrenzen */
                                     /* Adressen der Funktion und Genauigkeit */

    exact=exp(b)-exp(a);
    diff=fabs(sum-exact);

    printf("Nmax      romberg      exact      diff \n\n");
    printf("%d        %15.6e      %15.6e      %15.6e \n",n,sum,exact,diff);

    return 0;
}
/* Ergebnis:
Bitte geben Sie a,b und n ein: 0 1 2
  1      1.692503e+00      1.734162e+00
  2      1.718509e+00      1.721203e+00
  3      1.718237e+00      1.718918e+00
  4      1.718277e+00      1.718431e+00
  5      1.718281e+00      1.718318e+00
  6      1.718282e+00      1.718291e+00
Nmax      romberg      exact      diff

```

Hauptprogramm

Aufruf der Funktion “romberg”

Vergleich mit bekannten Ergebnis

Konvergenz nach 1-6 Schritten im Vergleich mit der Trapezregel

**Um $8 \cdot 10^{-8}$ Genauigkeit zu erreichen
braucht man mit Trapezregel
1200 Stützstellen (mit Beispiel 3.2)**