

Zusammenfassung vom 20.04.2020

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, höhere Ordnungen möglich (Simpson, Bode, ...) nur bei niedrigen Ordnungen rein positive Gewichte

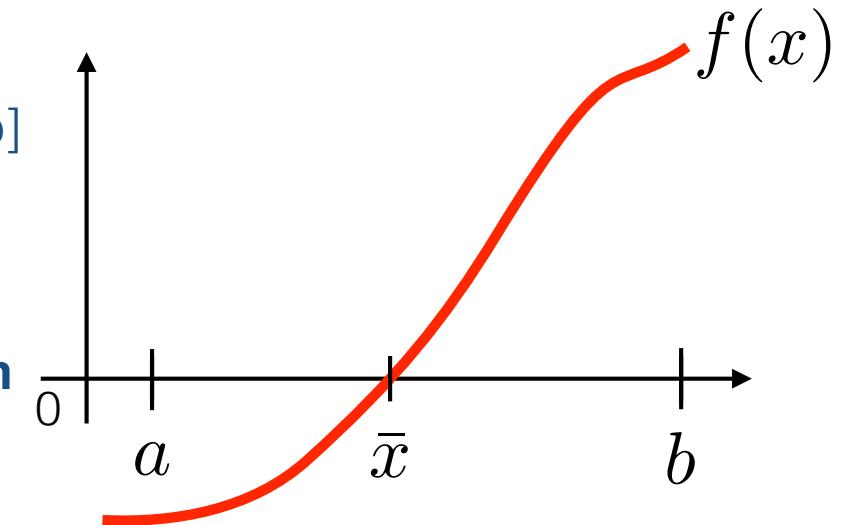
$$\int_a^b dx f(x) = \frac{h}{2} f_0 + h (f_1 + \cdots + f_{N-2}) + \frac{h}{2} f_{N-1} + \underbrace{(N-1) \cdot \mathcal{O}(h^3)}_{\mathcal{O}(h^2)}$$

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.

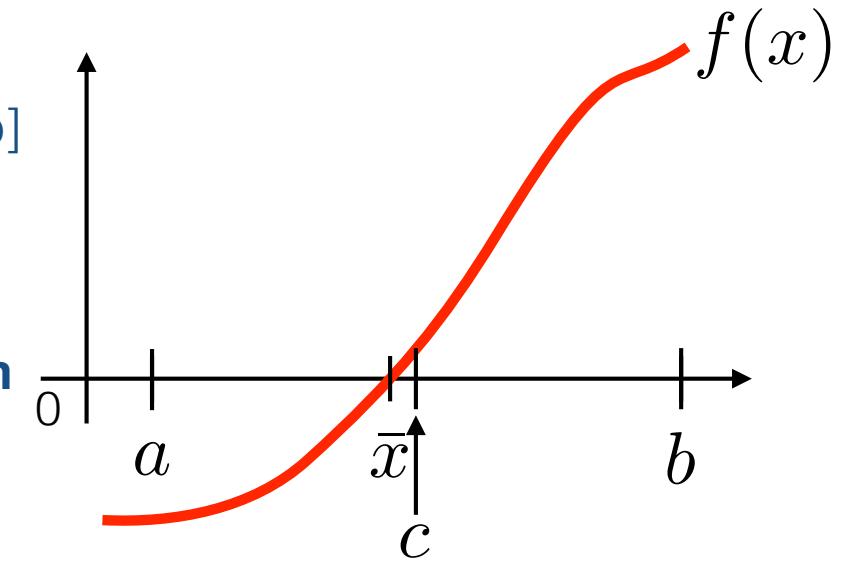
Nullstellensuche

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.

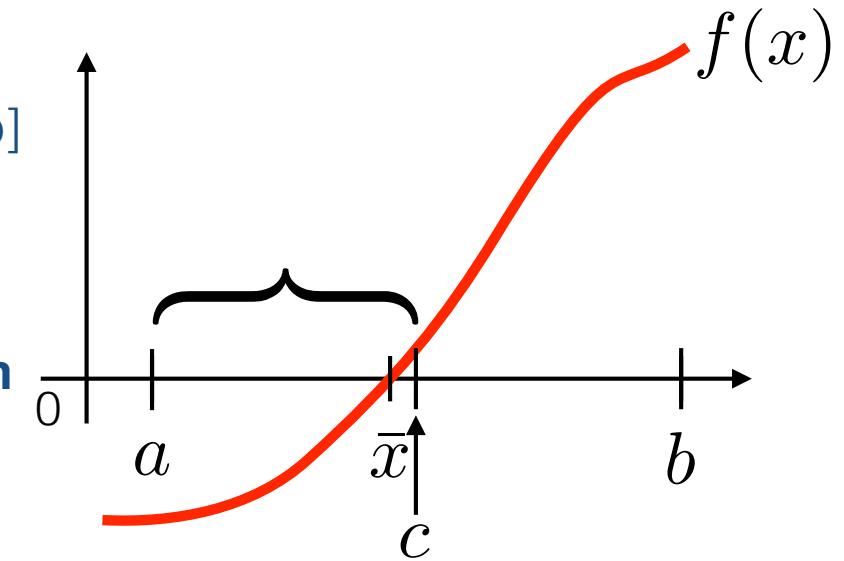
Nullstellensuche

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

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

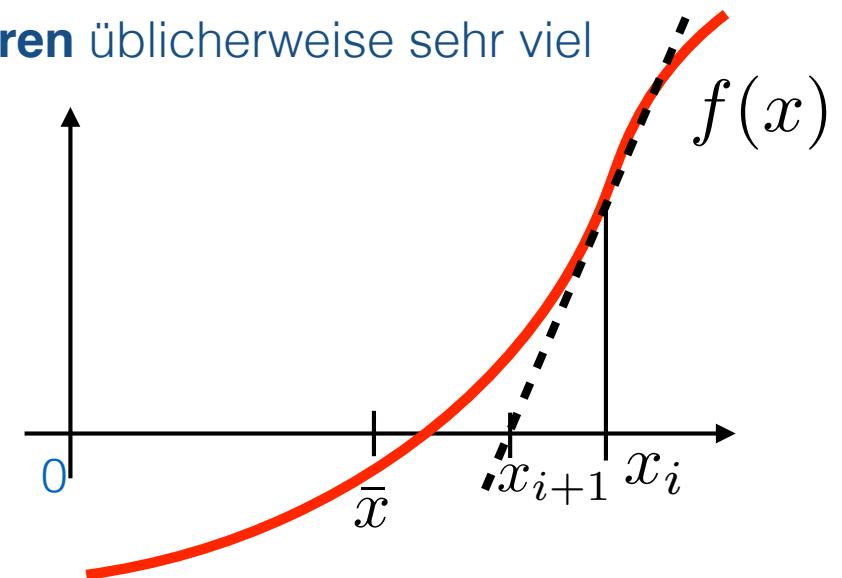
Schritte.

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)++;
    }
    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 */
}
```

```
/* 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 */
}
```

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)++;
    }
    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

Startwerte der Iteration: x1 und x2

```
/* 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)++;
    }
    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 */
}
```

Startwerte der Iteration: x1 und x2

```
/* 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)++;
    }
    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 ist ebenfalls Parameter

Startwerte der Iteration: x1 und x2

```
/* 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)++;
    }
    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 ist ebenfalls Parameter**Schritt wird als “Referenz” uebergeben**

Startwerte der Iteration: x1 und x2

```

/* 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)++;
    }
    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 ist ebenfalls Parameter

Schritt wird als “Referenz” uebergeben

*** Operator: bearbeite Variable an in schritt gespeicherter Adresse**

Startwerte der Iteration: x1 und x2

```

/* 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)++;
    }
    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 ist ebenfalls Parameter

Schritt wird als “Referenz” uebergeben

*** Operator: bearbeite Variable an in schritt gespeicherter Adresse**

Iteration nach dem Sekantenverfahren

```
/* 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=xn; /*
```

```
(*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

```
/* 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)++;
    }
    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 */
}
```

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 gefundene 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);
}
```

Beispielhauptprogramm wendet die Routine an

```
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);
}
```

```
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");
    /* 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);
}
```

Beispielhauptprogramm wendet die Routine an

Argumente von secant:

a,b wie üblich “call by reference”
&f = Adresse der Funktion f
(die genau ein double Argument hat)
&n Adresse der Variabel n

- Beispielhauptprogramm wendet die Routine an

```
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

```
/* Suche Nullstelle  
Referenz: Im Unterprogramm "secant" (auf der vorigen Folie)  
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 */
```

```
exact-  
diff-fabs(*schrift)++; /* Schritte=Schritte+1 */  
}
```

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

Argumente von secant:

a,b wie üblich “call by reference”

&f = Adresse der Funktion f

(die genau ein double Argument hat)

& n Adresse der Variabel n

Beispielhauptprogramm wendet die Routine an

```
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")
```

Argumente von secant:

a,b wie üblich “call by reference”
&f = Adresse der Funktion f
(die genau ein double Argument hat)
&n Adresse der Variabel n

```
/* Suche Nullstelle
Referenz: Im Unterprogramm "secant" (auf der vorigen Folie)
do
{
    /* naechster Schaetzwert 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 */
}
exact=
```

diff=fabs(res);

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

(*func)

- Beispielhauptprogramm wendet die Routine an

```
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

```
exact-  
diff-fabs(*schrift)++; /* Schritte=Schritte+1 */  
}
```

Argumente von secant:

a,b wie üblich “call by reference”
 $\&f$ = Adresse der Funktion f
(die genau ein double Argument hat)
 $\&n$ Adresse der Variabel n

2

```
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");
    /* 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);
}
```

Beispielhauptprogramm wendet die Routine an

Argumente von secant:

a,b wie üblich “call by reference”
&f = Adresse der Funktion f
(die genau ein double Argument hat)
&n Adresse der Variabel n

Beispielhauptprogramm wendet die Routine an

```

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
/* 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
*/

```

Argumente von secant:

a,b wie üblich “call by reference”
&f = Adresse der Funktion f
(die genau ein double Argument hat)
&n Adresse der Variabel n

Beispielhauptprogramm wendet die Routine an

```

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
/* 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
*/

```

Argumente von secant:

a,b wie üblich “call by reference”
&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:

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

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

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

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

$$\int_a^b dx \ f(x) = T(h) - \frac{h^2}{4} (f'(b) - f'(a)) + \frac{h^3}{12} \sum_{i=0}^{N-2} (f''(x_i) + f''(x_{i+1})) \cdots$$

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

$$\int_a^b dx \ f(x) = T(h) - \frac{h^2}{4} (f'(b) - f'(a)) + \frac{h^3}{12} \sum_{i=0}^{N-2} \overrightarrow{h} \left(f''(x_i) + f''(x_{i+1}) \right) \dots$$

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

$$\int_a^b dx f(x) = T(h) - \frac{h^2}{4} (f'(b) - f'(a)) + \frac{h^3}{12} \sum_{i=0}^{N-2} \left(\overbrace{f''(x_i)}^{\frac{f'(x_{i+1}) - f'(x_i)}{h}} + f''(x_{i+1}) \right) \cdots$$

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

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

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

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

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

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

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

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

“Romberg Folge”

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

$$h_0 = b - a$$

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

\vdots

$$h_m = \frac{h_{m-1}}{2}$$

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

“Romberg Folge”

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

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

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

$$h_0 = b - a$$

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

⋮

$$h_m = \frac{h_{m-1}}{2}$$

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

“Romberg Folge”

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

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

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

$$h_0 = b - a$$

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

⋮

$$h_m = \frac{h_{m-1}}{2}$$

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

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

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

$$\begin{aligned} \int_a^b dx f(x) &= T(h) - \frac{h^2}{12} (f'(b) - f'(a)) + O(h^4) + O(h^6) + \dots + O(h^{2k}) \\ \implies T(h) &\approx \underbrace{\int_a^b dx f(x)}_{\tau_0} + \tau_1 h^2 + \tau_2 h^4 + \dots + \tau_m h^{2m} \end{aligned}$$

“Romberg Folge”

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

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

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

$$h_0 = b - a$$

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

⋮

$$h_m = \frac{h_{m-1}}{2}$$

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

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

$$\tilde{T}_{00} = T(h_0)$$

Neville Schema

$$\tilde{T}_{10} = T(h_1)$$

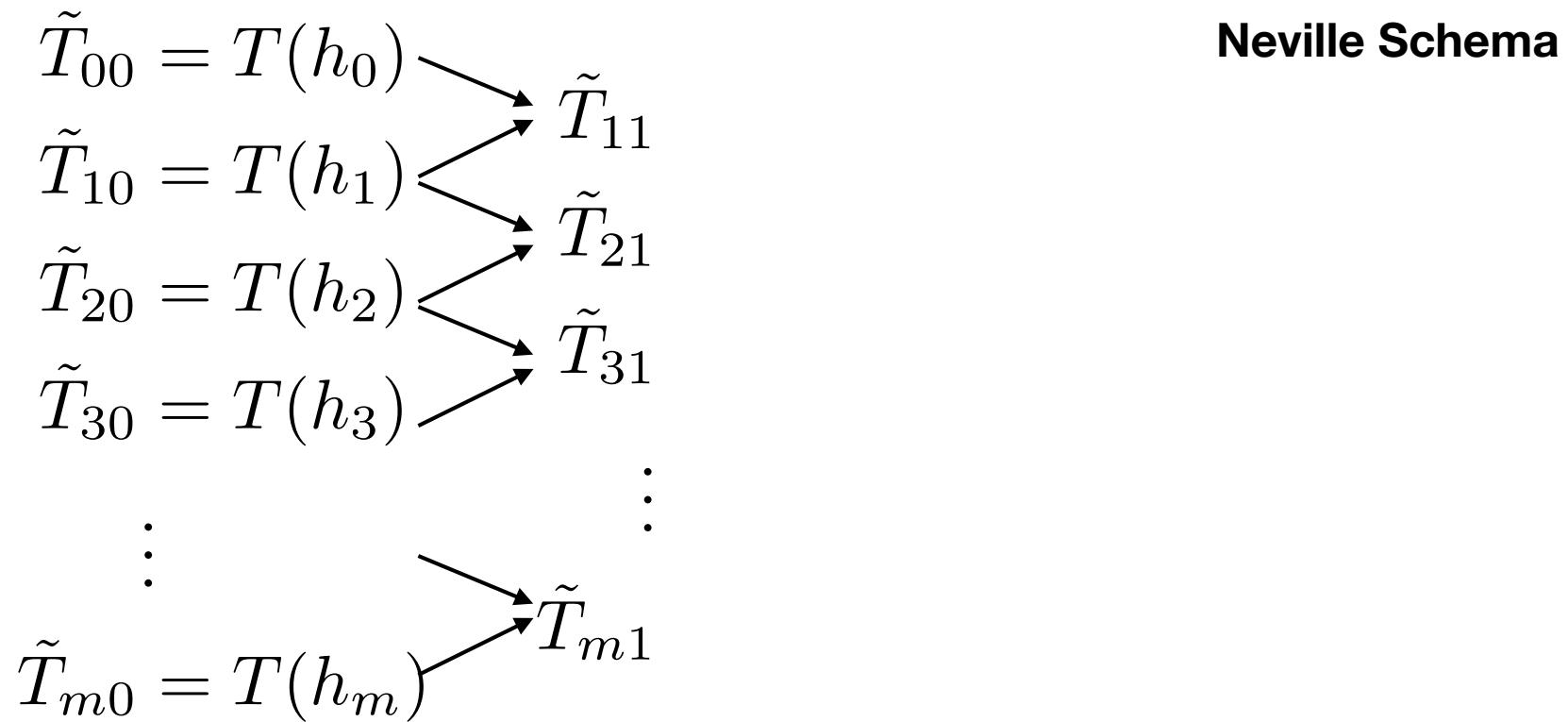
$$\tilde{T}_{20} = T(h_2)$$

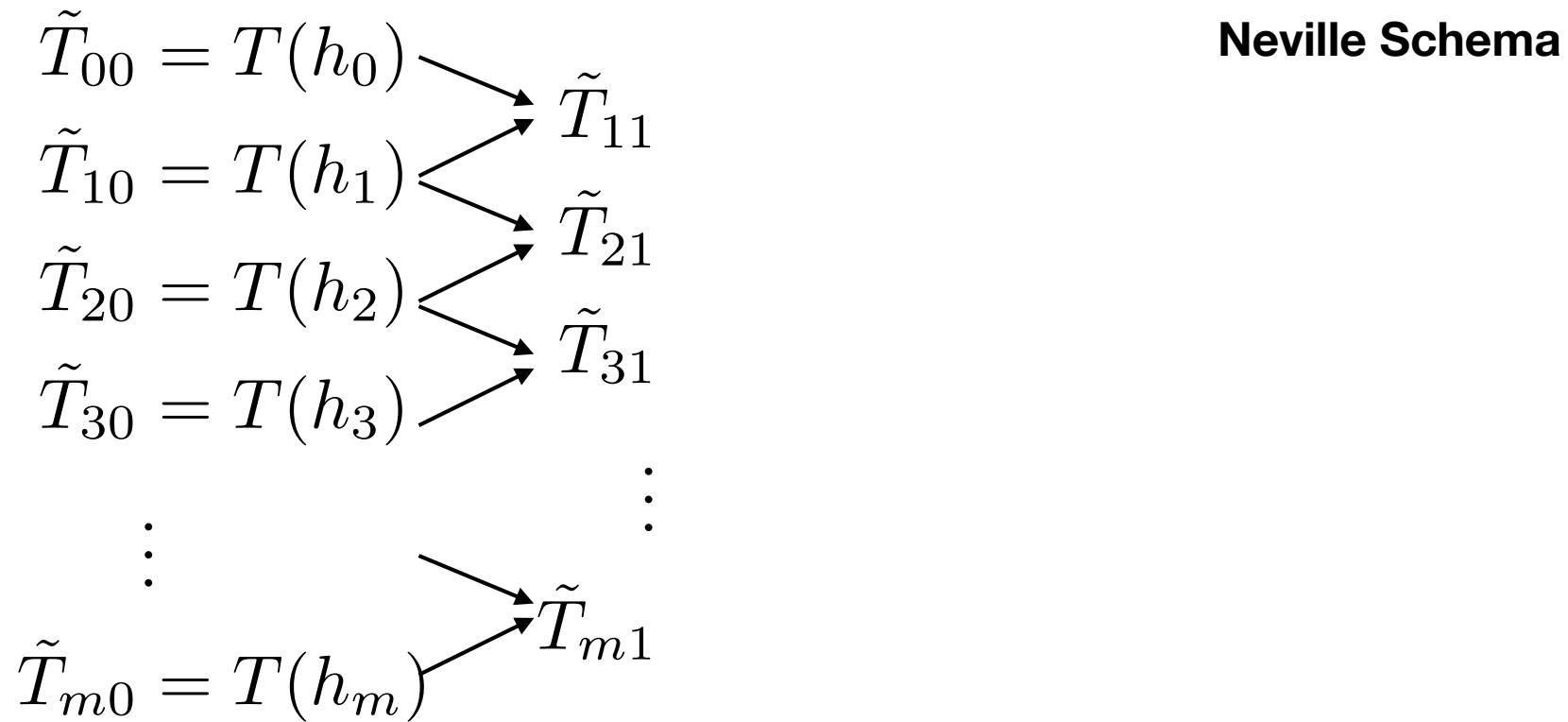
$$\tilde{T}_{30} = T(h_3)$$

•
⋮

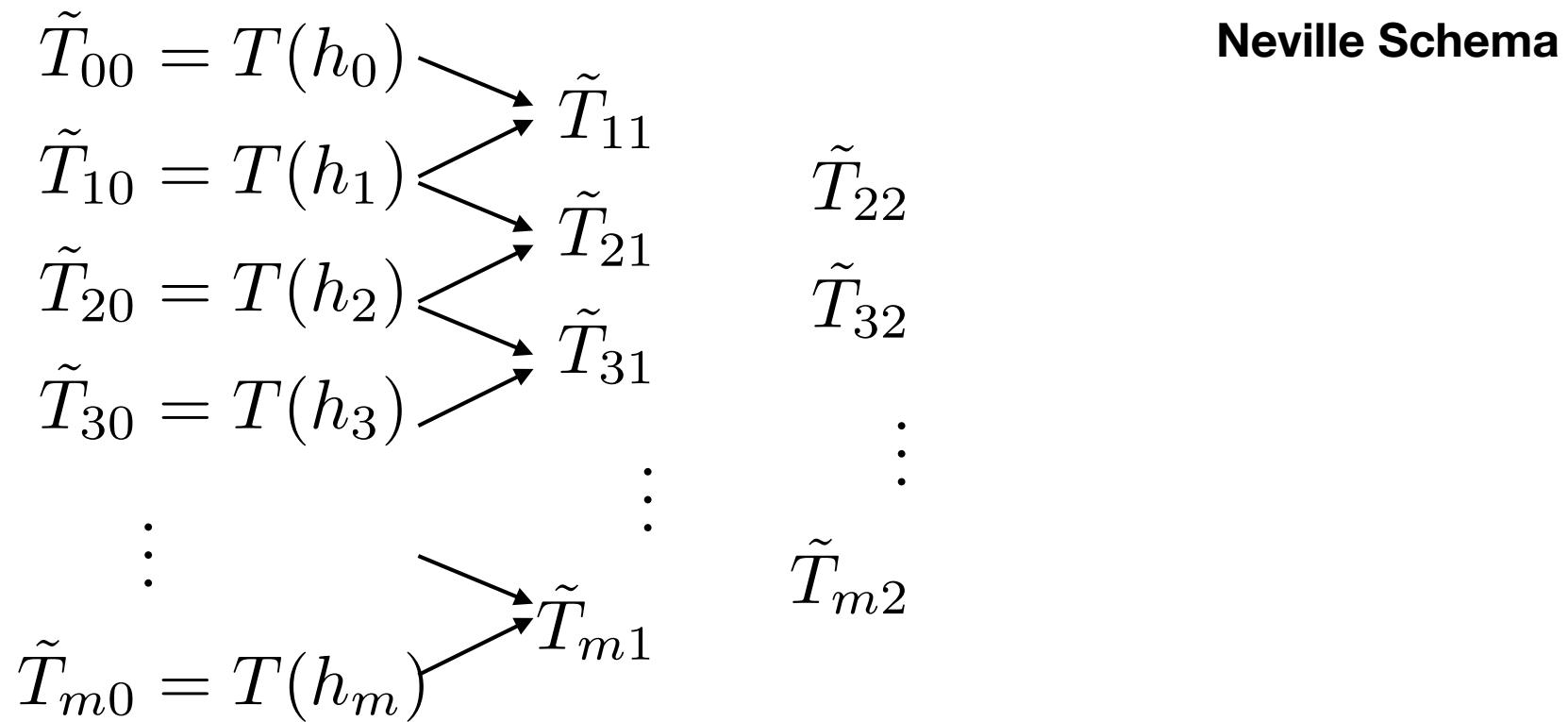
$$\tilde{T}_{m0} = T(h_m)$$

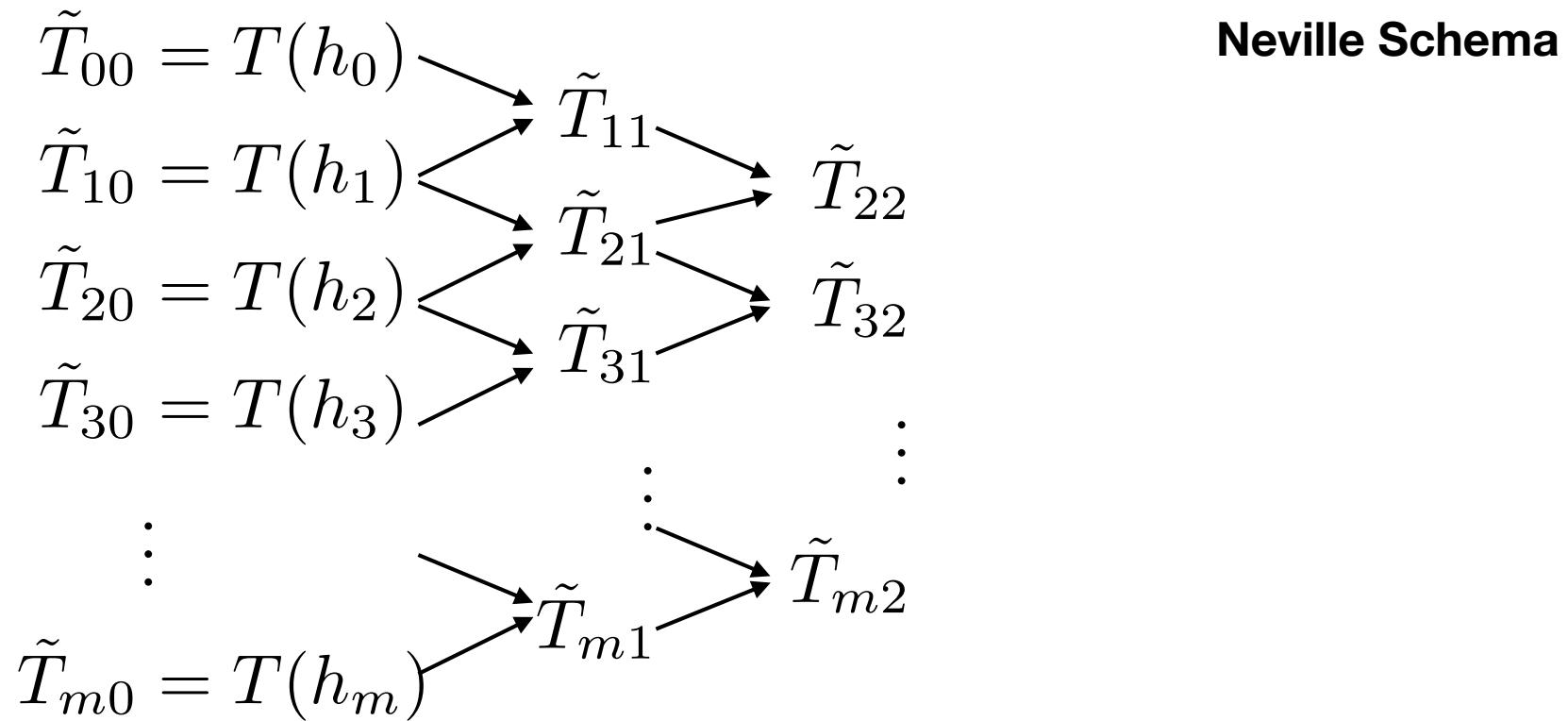
$$\begin{array}{ll} \tilde{T}_{00} = T(h_0) & \textbf{Neville Schema} \\ \tilde{T}_{10} = T(h_1) & \tilde{T}_{11} \\ \tilde{T}_{20} = T(h_2) & \tilde{T}_{21} \\ \tilde{T}_{30} = T(h_3) & \tilde{T}_{31} \\ \vdots & \vdots \\ \tilde{T}_{m0} = T(h_m) & \tilde{T}_{m1} \end{array}$$

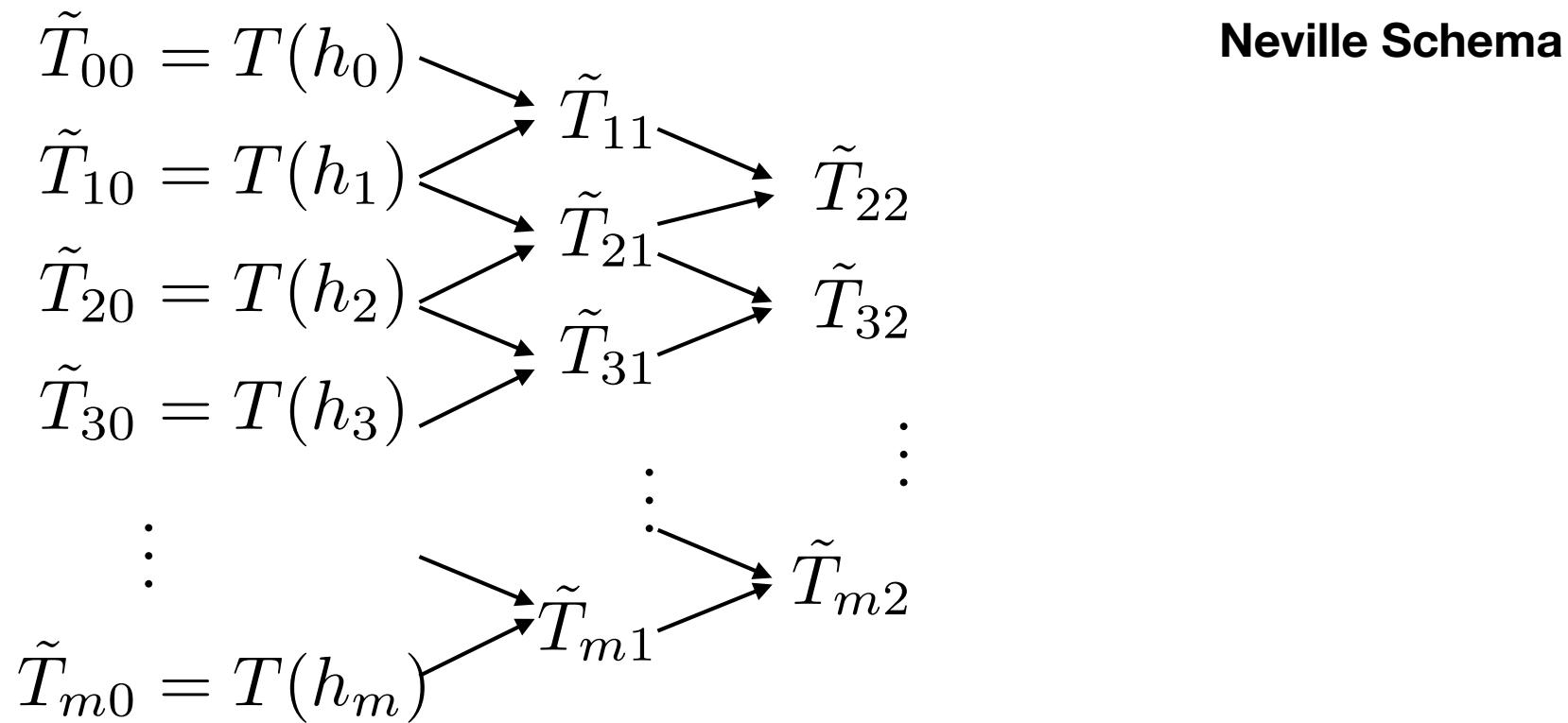




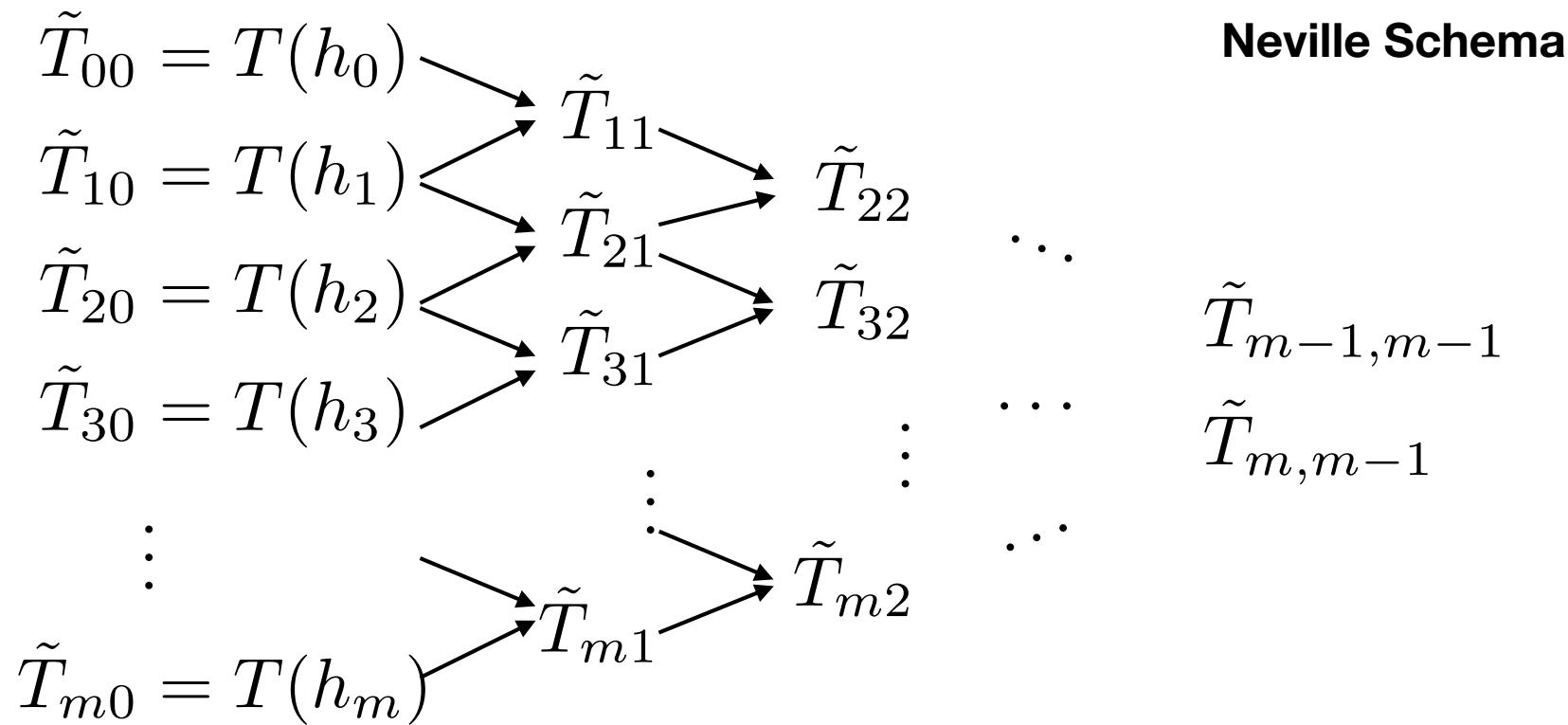
$$\tilde{T}_{11}(h) = \frac{h^2 - h_0^2}{h_1^2 - h_0^2} \tilde{T}_{10} - \frac{h^2 - h_1^2}{h_1^2 - h_0^2} \tilde{T}_{00}$$

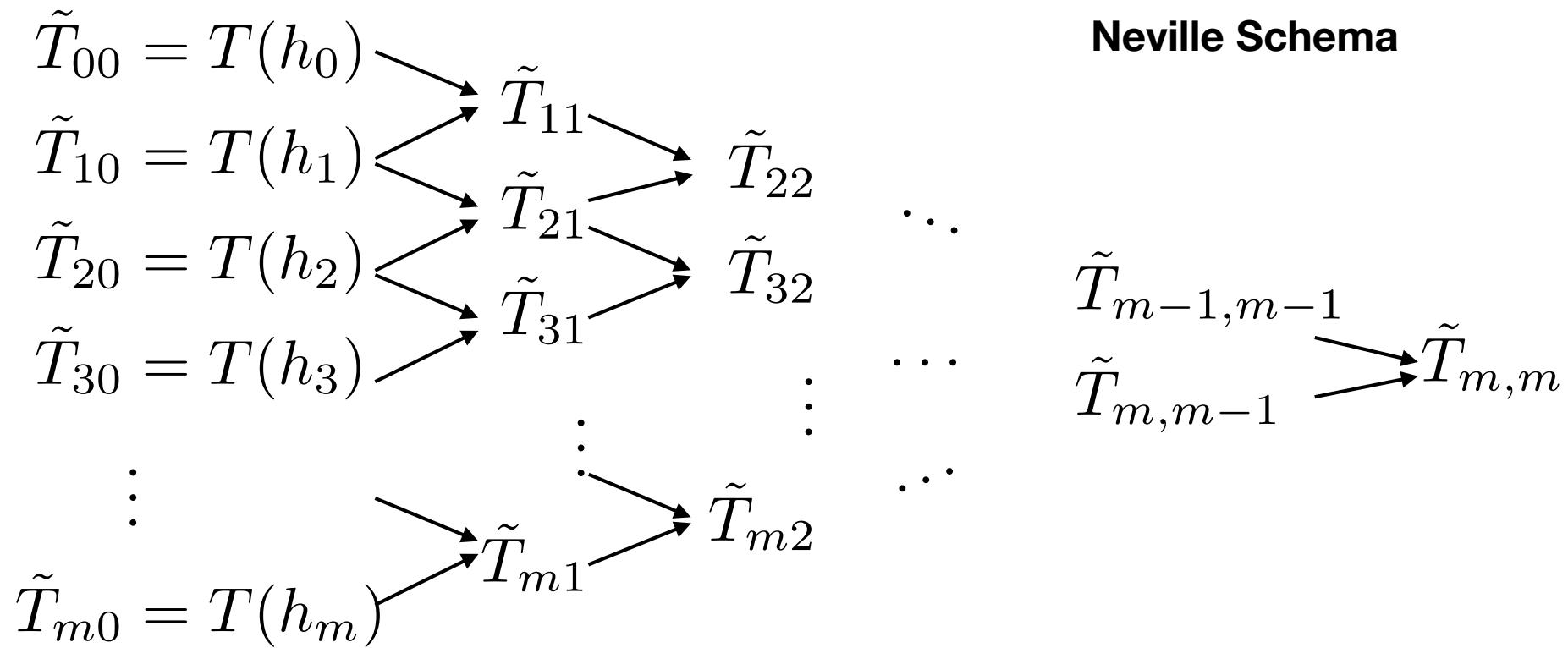


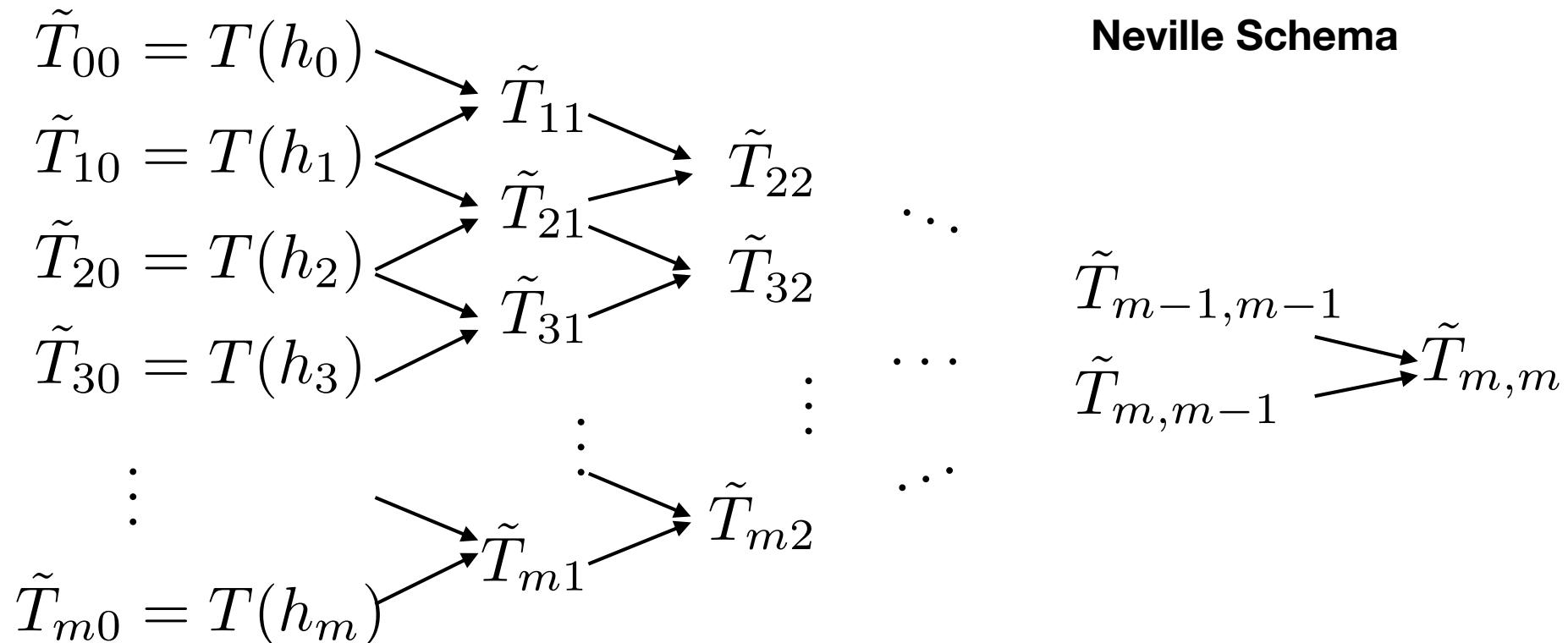




$$\tilde{T}_{32}(h) = \frac{h^2 - h_1^2}{h_3^2 - h_1^2} \tilde{T}_{31}(h) - \frac{h^2 - h_3^2}{h_3^2 - h_1^2} \tilde{T}_{21}(h)$$

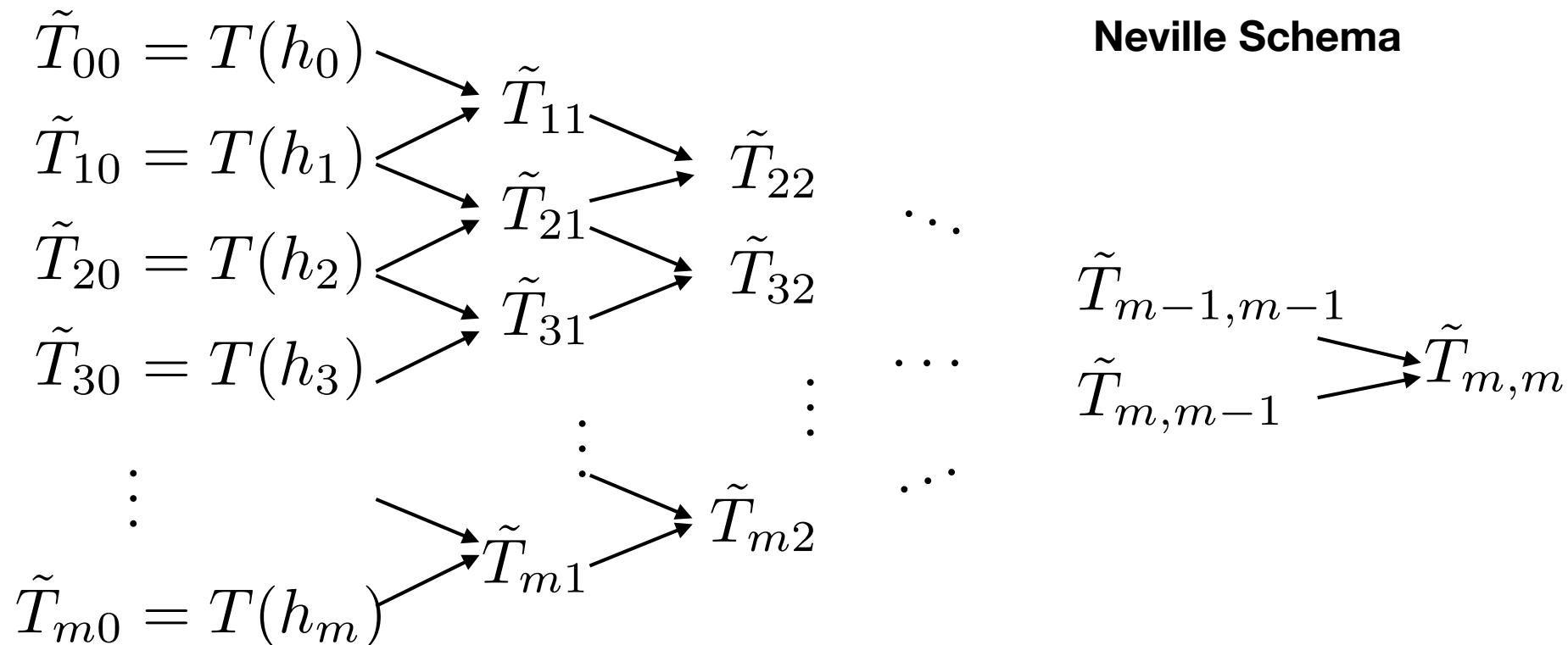




**Neville Schema**

Dazu definieren wir Polynome $\tilde{T}_{ik}(h)$ vom Grad $2k$, die $T(h_j)$ für $j=i-k, \dots, i$ reproduzieren.
Das macht natürlich nur Sinn für $i \geq k$ und $k \leq m$.

(deswegen auch die Notation $\tilde{T}_{mm}(h)$ auf der letzten Folie)

**Neville Schema**

Dazu definieren wir Polynome $\tilde{T}_{ik}(h)$ vom Grad $2k$, die $T(h_j)$ für $j=i-k, \dots, i$ reproduzieren.
Das macht natürlich nur Sinn für $i \geq k$ und $k \leq m$.

(deswegen auch die Notation $\tilde{T}_{mm}(h)$ auf der letzten Folie)

Diese Polynome kann man durch Rekursion erhalten.

Angenommen $\tilde{T}_{i,k-1}(h)$ und $\tilde{T}_{i-1,k-1}(h)$ sind bereits bekannt. Dann definiert man

$$\tilde{T}_{i,k}(h) = \frac{h^2 - h_{i-k}^2}{h_i^2 - h_{i-k}^2} \tilde{T}_{i,k-1}(h) - \frac{h^2 - h_i^2}{h_i^2 - h_{i-k}^2} \tilde{T}_{i-1,k-1}(h)$$

Verlauf der Romberg Integration:

- Berechnen Sie $T(h_j)$ $\left(= \tilde{T}_{j,0}(h_j) \right)$ für $j = 1, \dots, m$ mit Trapezregel
- Bestimmen Sie die Polynomwerte $\tilde{T}_{ij}(0)$
 - Benutzen Sie $h=0$ in der Rekursionsformel (Neville Schema)
- $\tilde{T}_{m,m}(0)$ ist die gesuchte Annäherung unseres Integrals
- Stellen Sie die Schritte oben in eine **for**-Schleife
 - Die Auswertung mit **$h=0$** ergibt dann jeweils eine **Extrapolation zum gesuchten Integral.**

Für jedes **neue h** kann man eine Ordnung höher gehen ohne jedesmal alles komplett neu zu berechnen (im Beispiel nicht realisiert)

```
/* 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); }
```

```
/* 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**

```
/* 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.

```
/* 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

```
/* 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})**

```
/* 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 */

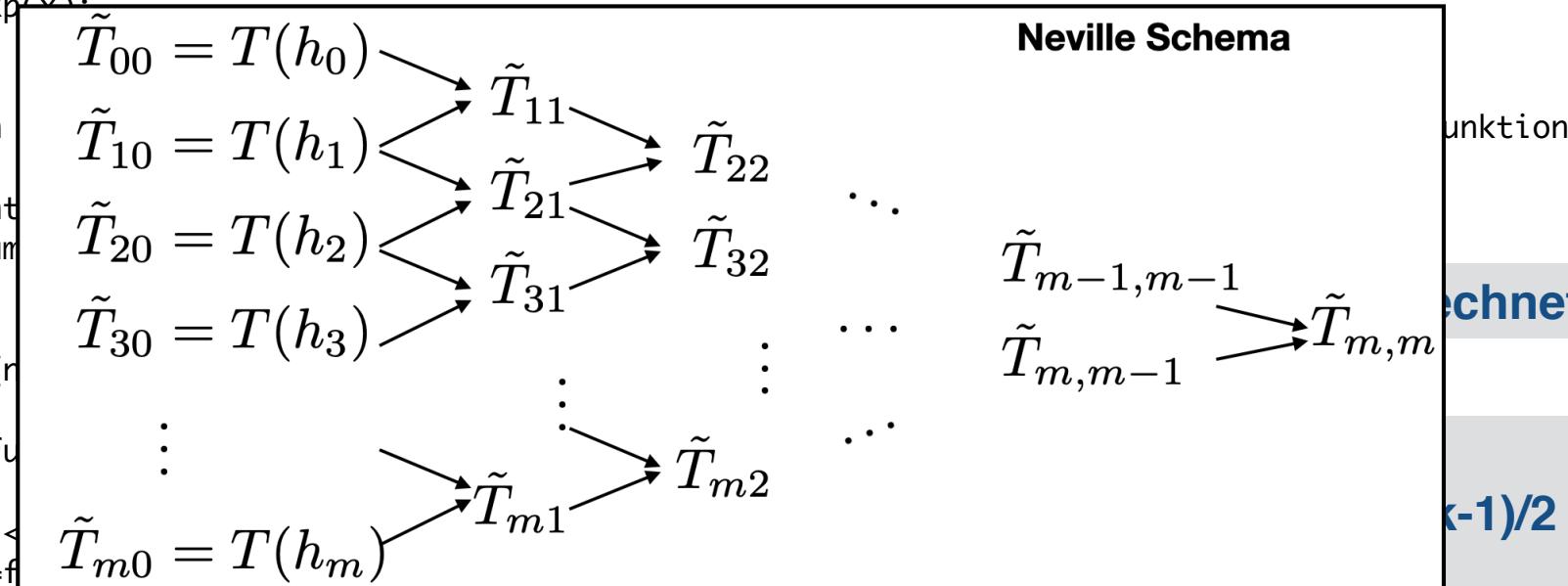
/* Definition der zu integrierenden Funktion */

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

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

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

Funktion, die zu integrieren ist.



(eindeutig, zum Speichern vom T_{ik})

```
/* 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); }
```

```
/* 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 durchfhrt.
n0 Anzahl der Sttzstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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)/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]); /* just to observe convergence */
    if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
  }

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

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchfhrt.
n0 Anzahl der Sttzstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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)/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]); /* just to observe convergence */
    if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
  }

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

```

**Deklarationen:
Felder fr h,T(h), T_{ik}(0)**

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchfhrt.
n0 Anzahl der Sttzstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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 */
  Tsum=(double *)malloc((mmax+1)*sizeof(double)); /* und T fuer diese h */
  tildeT=(double *)malloc((mmax+1)*(mmax+2)/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]); /* just to observe convergence */
    if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
  }

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

```

**Deklarationen:
Felder fr h,T(h), T_{ik}(0)**

**Alloziere Speicher
fr h,T(h), T_{ik}(0)**

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchfhrt.
n0 Anzahl der Sttzstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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 */
  Tsum=(double *)malloc((mmax+1)*sizeof(double)); /* und T fuer diese h */
  tildeT=(double *)malloc((mmax+1)*(mmax+2)/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]); /* just to observe convergence */
    if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
  }

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

```

Deklarationen:
Felder fr h,T(h), T_{ik}(0)

Alloziere Speicher
fr h,T(h), T_{ik}(0)

Vorbereitung fr m=0
fr h=h₀,T(h₀), T₀₀(0)

$$\tilde{T}_{00} = T(h_0)$$

$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

 \vdots

$$\tilde{T}_{m0} = T(h_m)$$

$$\tilde{T}_{11}$$

$$\tilde{T}_{21}$$

$$\tilde{T}_{31}$$

$$\vdots$$

$$\tilde{T}_{m1}$$

$$\tilde{T}_{22}$$

$$\tilde{T}_{32}$$

$$\vdots$$

$$\tilde{T}_{m2}$$

Neville Schema

$$\begin{array}{c} \tilde{T}_{m-1,m-1} \\ \tilde{T}_{m,m-1} \end{array} \rightarrow \tilde{T}_{m,m}$$

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

Bromberg Routine

43

• Stuetzstellen

• Daten:

$h, T(h), T_{ik}(0)$

• Schritt 1: Alloziere Speicher
für $h, T(h), T_{ik}(0)$

• Schritt 2: Neville Schema

```

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]); /* just to observe convergence */
  if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
}

printf("Final result: %15.6e \n",tildeT[index_ik(m,m)]);

```

```

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

```

$$\tilde{T}_{00} = T(h_0)$$

$$\tilde{T}_{10} = T(h_1) \rightarrow \tilde{T}_{11}$$

$$\tilde{T}_{20} = T(h_2) \rightarrow \tilde{T}_{21} \rightarrow \tilde{T}_{22}$$

$$\tilde{T}_{30} = T(h_3) \rightarrow \tilde{T}_{31} \rightarrow \tilde{T}_{32}$$

$$\vdots \quad \vdots \quad \vdots$$

$$\tilde{T}_{m0} = T(h_m) \rightarrow \tilde{T}_{m1} \rightarrow \tilde{T}_{m2}$$

Neville Schema

Bomberg Routine

43

Stuetzstellen

Werte:

$h, T(h), T_{ik}(0)$

Schritt 1: Alloziere Speicher für $h, T(h), T_{ik}(0)$

Schritt 2: Neville Schema

```
n=*n0;  
Tsum[0]=T(n,a,b,func);  
tildeT[index_ik(0,0)]=Tsum[0];
```

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

```
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]); /* just to observe convergence */  
  if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */  
}
```

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

$$\tilde{T}_{00} = T(h_0)$$

$$\tilde{T}_{10} = T(h_1) \rightarrow \tilde{T}_{11}$$

$$\tilde{T}_{20} = T(h_2) \rightarrow \tilde{T}_{21} \rightarrow \tilde{T}_{22}$$

$$\tilde{T}_{30} = T(h_3) \rightarrow \tilde{T}_{31} \rightarrow \tilde{T}_{32}$$

$$\vdots \quad \vdots \quad \vdots$$

$$\tilde{T}_{m0} = T(h_m) \rightarrow \tilde{T}_{m1} \rightarrow \tilde{T}_{m2}$$

Neville Schema

Bromberg Routine

43

Stuetzstellen

Werte:

$h, T(h), T_{ik}(0)$

Schritt 1: Alloziere Speicher für $h, T(h), T_{ik}(0)$

Schritt 2: Neville Schema

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

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

```
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]); /* just to observe convergence */
  if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
}

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

$$\tilde{T}_{00} = T(h_0)$$

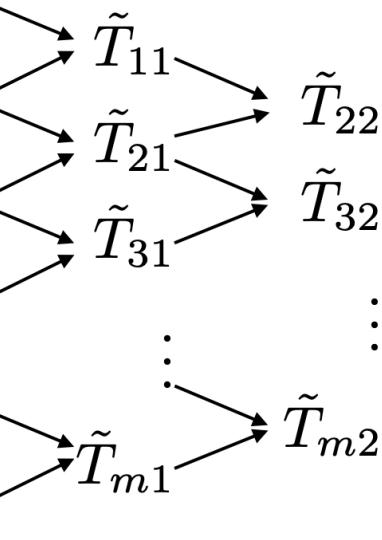
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

:

$$\tilde{T}_{m0} = T(h_m)$$



Neville Schema

Bromberg Routine

43

Stuetzstellen

Werte:

$h, T(h), T_{ik}(0)$

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

Neville Schema

`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]); /* just to observe convergence */
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */`

`*n0=n;`

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

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

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

$$\tilde{T}_{00} = T(h_0)$$

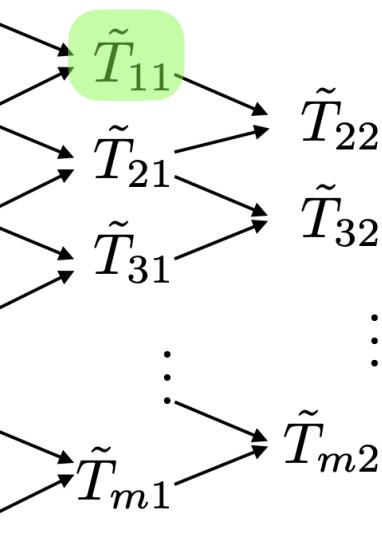
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

:

$$\tilde{T}_{m0} = T(h_m)$$



Neville Schema

Bromberg Routine

43

Stuetzstellen

Koeffizienten:

$h, T(h), T_{ik}(0)$

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

Neville Schema

`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]); /* just to observe convergence */
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */`

`}`

`*n0=n;`

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

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

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

$$\tilde{T}_{00} = T(h_0)$$

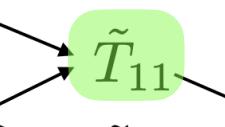
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

 \vdots

$$\tilde{T}_{m0} = T(h_m)$$



```
n=*n0;
Tsum[0]=T(n,a,b,func);
tildeT[index_ik(0,0)]=Tsum[0];
```

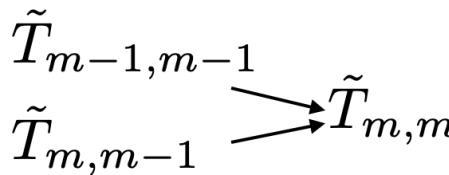
```
for(m=1;m<=mmax;m++)
{ h[m]=h[m-1]/2;           /* Trapezsumme fuer halbiertes h */
  n=2*n;
  Tsum[m]=T(n,a,b,func);
  tildeT[index_ik(m,0)]=Tsum[m]; /* fuer i=m und k=0 */
```

```
for(k=1;k<=m;k++)
{ 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)];
}
```

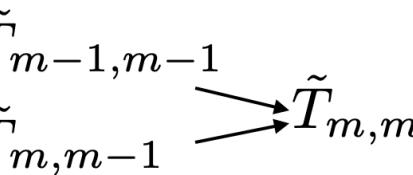
```
printf("%5d %15.6e %15.6e \n",m,tildeT[index_ik(m,m)],Tsum[m]); /* just to observe convergence */
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
}
```

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

Neville Schema



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



Bromberg Routine

43

1 Stuetzstellen

2 Daten:

$h, T(h), T_{ik}(0)$

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

4 Neville Schema

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

bestimme $T_{mk}(0)$

```
    /* 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]); /* just to observe convergence */
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break; /* stop when accuracy reached */
```

```
}
```

```
*n0=n;
```

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

$$\tilde{T}_{00} = T(h_0)$$

$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

 \vdots

$$\tilde{T}_{m0} = T(h_m)$$

$$\tilde{T}_{11}$$

$$\tilde{T}_{21}$$

$$\tilde{T}_{31}$$

$$\vdots$$

$$\tilde{T}_{m1}$$

$$\tilde{T}_{22}$$

$$\tilde{T}_{32}$$

$$\vdots$$

$$\tilde{T}_{m2}$$

 \dots \vdots \dots

Neville Schema

$$\begin{array}{c} \tilde{T}_{m-1,m-1} \\ \tilde{T}_{m,m-1} \end{array} \rightarrow \tilde{T}_{m,m}$$

Bomberg Routine

43

Stuetzstellen

Konnen:

 $h, T(h), T_{ik}(0)$

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

Neville Schema

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]); /*

if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break;

}

*n0=n;

result=tildeT[index_ik(m,m)];

free(tildeT); free(Tsum); free(h);

return result; }

**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
Abbruch mit "break"**

$$\tilde{T}_{00} = T(h_0)$$

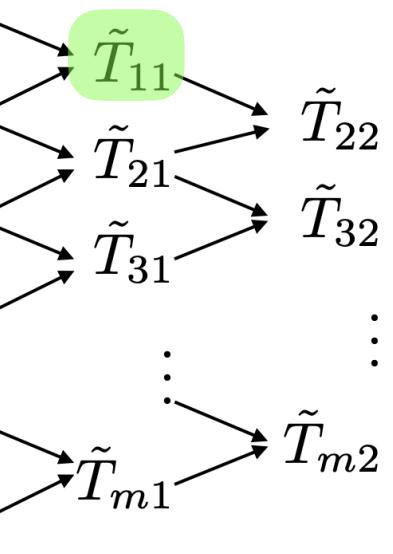
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

:

$$\tilde{T}_{m0} = T(h_m)$$



Neville Schema

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]); /*

 if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break;

}

*n0=n;

result=tildeT[index_ik(m,m)];

free(tildeT); free(Tsum); free(h);

return result; }

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

$\tilde{T}_{m-1,m-1}$
 $\tilde{T}_{m,m-1}$ $\rightarrow \tilde{T}_{m,m}$

Bomberg Routine

43

Stuetzstellen

Werte:

$h, T(h), T_{ik}(0)$

Schritt 1: Alloziere Speicher
für $h, T(h), T_{ik}(0)$

Neville Schema

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

bestimme $T_{mk}(0)$

Ausdruck um Konvergenz
zu beobachten
Abbruch mit "break"

$$\tilde{T}_{00} = T(h_0)$$

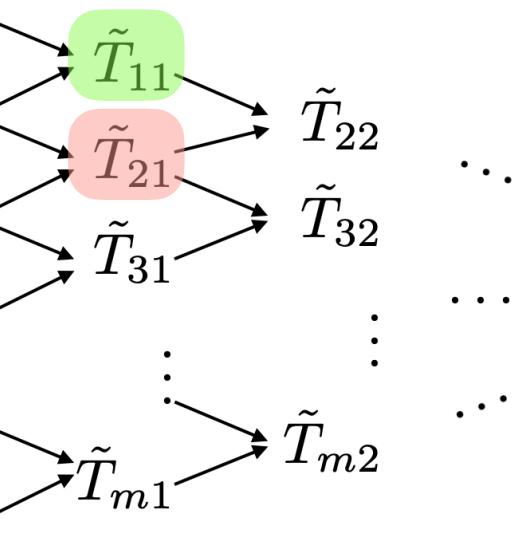
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

:

$$\tilde{T}_{m0} = T(h_m)$$



Neville Schema

Bomberg Routine

43

Stuetzstellen

Werte:

$h, T(h), T_{ik}(0)$

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

Neville Schema

```
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]); /*
```

```
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break;
```

```
}
```

```
*n0=n;
```

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

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
Abbruch mit "break"

$$\tilde{T}_{00} = T(h_0)$$

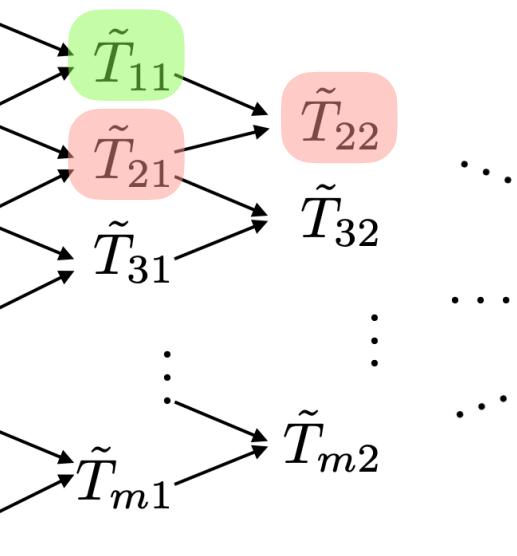
$$\tilde{T}_{10} = T(h_1)$$

$$\tilde{T}_{20} = T(h_2)$$

$$\tilde{T}_{30} = T(h_3)$$

:

$$\tilde{T}_{m0} = T(h_m)$$



Neville Schema

Bomberg Routine

43

Stuetzstellen

Koeffizienten:

$h, T(h), T_{ik}(0)$

Schritt 1: Alloziere Speicher für $h, T(h), T_{ik}(0)$

Schritt 2: Neville Schema

```
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]); /*
```

```
if(fabs(tildeT[index_ik(m,m)]-tildeT[index_ik(m-1,m-1)])<=eps) break;
```

```
}
```

```
*n0=n;
```

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

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
Abbruch mit "break"

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchföhrt.
n0 Anzahl der Stützstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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 */
  Tsum=(double *)malloc((mmax+1)*sizeof(double)); /* und T fuer diese h */
  tildeT=(double *)malloc((mmax+1)*(mmax+2)/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
    } Abbruch 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
Abbruch mit "break"**

```

/* Routine, die Romberg Integration bis zu einer Genauigkeit eps durchfhrt.
n0 Anzahl der Sttzstellen im ersten Schritt, maximale Anzahl der benutzten Stuetzstellen
a, b die Integralgrenzen und f die zu integrierend 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 */
  Tsum=(double *)malloc((mmax+1)*sizeof(double)); /* und T fuer diese h */
  tildeT=(double *)malloc((mmax+1)*(mmax+2)/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
  } Abbruch mit "break"

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

```

**Deklarationen:
Felder fr h,T(h), T_{ik}(0)**

**Alloziere Speicher
fr h,T(h), T_{ik}(0)**

**Vorbereitung fr m=0
fr h=h₀,T(h₀), T₀₀(0)**

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

bestimme T_{mk}(0)

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

dealloziere den Speicher fr 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;
}
```

Hauptprogramm

```
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;
}
```

Hauptprogramm

Aufruf der Funktion “romberg”

```
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;
}
```

Hauptprogramm

Aufruf der Funktion “romberg”

Vergleich mit bekannten Ergebnis

```

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);

```

Hauptprogramm

```

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);

```

Aufruf der Funktion “romberg”

```

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

```

Nmax	romberg	exact	diff
128	1.718282e+00	1.718282e+00	8.316105e-08 (Trapezregel benoetigt etwa 1200 Punkte)*,

Vergleich mit bekannten Ergebnis

```

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);

```

Hauptprogramm

```

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);

```

Aufruf der Funktion “romberg”

```

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

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)**

128 1.718282e+00 1.718282e+00

8.316105e-08 (Trapezregel benoetigt etwa 1200 Punkte)*,