Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
31.05.2022

MATTHIAS HANAUSKE

FRANKFURT INSTITUTE FOR ADVANCED STUDIES
JOHANN WOLFGANG GOETHE UNIVERSITÄT
INSTITUT FÜR THEORETISCHE PHYSIK
ARBEITSGRUPPE RELATIVISTISCHE ASTROPHYSIK
D-60438 FRANKFURT AM MAIN
GERMANY

7. Vorlesung

Plan für die heutige Vorlesung

- Kurze Wiederholung der Vorlesung 6
- Objekt-orientierte Programmierung und C++ Klassen
- Theorie: Numerische Integration
- Anwendungsbeispiel: Numerische Integration
- Übungsaufgaben: Übungsblatt Nr.7

Wiederholung der Vorlesung 6

- Mehrdimensionale C++ Arrays
- Jupyter Notebooks und das Rechnen mit symbolischen Ausdrücken
- Numerische Differentiation
- Übungsaufgaben: Übungsblatt Nr.6

Mehrdimensionale C++ Arrays

Dieser Unterpunkt baut auf dem Thema C++ Arrays, Zeiger und Referenzen der vorigen Vorlesung auf und betrachtet mehrdimensionale C++ Arrays. *Mehrdimensionale C++ Arrays* werden im Prinzip als ein Array von Arrays dargestellt und sie beschreiben einen Matrix-ähnlichen integrierten Datentyp. Für einen Typ T entspricht "T Name[m][n];" der Deklaration einer $(m \times n)$-Matrix, wobei die einzelnen Elemente vom gleichen Typ T sein müssen. Möchten wir z.B. eine $(m \times n)$-Matrix $\mathcal{A}$, bestehend aus Gleitkommazahlen vom Typ double in einem C++ Programm als ein integriertes Array deklarieren, so würden wir die folgende Anweisung in den Quelltext schreiben: double A[m][n];

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} = (a_{ij})_{i=1,2,\dots,m\,;\,j=1,2,\dots,n} \quad \Longrightarrow \quad \text{Deklaration z.B. mit: double A[m][n];}$$

Mehrdimensionale C++ Arrays und Zeiger

In gleicher Weise wie auch bei eindimensionalen Arrays sind mehrdimensionale C++ Arrays mittels des Konstruktes des Zeigers implementiert. Die einzelnen Elemente eines Arrays (z.B. "double A[m][n]") sind in Form von Zeigern auf die eindimensionalen Zeilen-Unterarrays geordnet. Der Zugriff auf den Wert eines Array-Elementes kann entweder durch die Angabe des entsprechenden Matrix-Indexes erfolgen (A[i][j]), oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen (*(A[i] + j) bzw. *(zeiger_A + i*n + j)). Dies Eigenschaften von mehrdimensionale C++ Arrays wollen wir nun, am Beispiel von int-Arrays näher betrachten.

In dem folgenden C++ Programm werden zwei int-Arrays definiert, die den folgenden zwei Matrizen $\mathcal{A}$ und $\mathcal{B}$ entsprechen:

$$(3 \times 5)\text{-Matrix} \quad \mathcal{A} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 21 & 22 & 23 & 24 & 25 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix} \quad , \qquad (5 \times 3)\text{-Matrix} \quad \mathcal{B} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix}$$

```cpp
#include <iostream>                     // Ein- und Ausgabebibliothek

int main(){                             // Hauptfunktion
    const int m = 3;
    const int n = 5;
    int A[m][n] = { { 1, 2, 3, 4, 5},
                    {21,22,23,24,25},
                    {51,52,53,54,55} }; // Definition eines (3x5)-Integer-Arrays ((3x5)-Matrix)
    int B[n][m] = { { 1, 2, 3},
                    { 4, 5, 6},
                    { 7, 8, 9},
                    {10,11,12},
                    {13,14,15} };       // Definition eines (5x3)-Integer-Arrays ((5x3)-Matrix)

    int* zeiger_A = &A[0][0];           // Definition des Zeigers auf das Integer-Array A

    constexpr int anz_elem_A = sizeof(A)/sizeof(A[0][0]); // Gesamte Anzahl der Elemente im Array A
    constexpr int dim_m = sizeof(A)/sizeof(A[0]);         // Dimension m des Arrays (Anzahl der Zeilen der Matrix A)
    constexpr int dim_n = sizeof(A[0])/sizeof(A[0][0]);   // Dimension n des Arrays (Anzahl der Spalten der Matrix A)

    printf("Matrix A: \n");
    for(int i=0; i < m; ++i){           // Anfang Schleife 1 (Zeilen der Matrix)
        printf("| ");
        for(int j=0; j < n; ++j){       // Anfang Schleife 2 (Spalten der Matrix)
            printf("%3i ", A[i][j]); // Ausgabe des Matrixwertes A_{ij}
        }                               // Ende der 1.Schleife
        printf("| \n");
    }                                   // Ende der 1.Schleife

    printf("\nMatrix B: \n");
    for(int i=0; i < n; ++i){           // Anfang Schleife 1 (Zeilen der Matrix)
        printf("| ");
        for(int j=0; j < m; ++j){       // Anfang Schleife 2 (Spalten der Matrix)
            printf("%3i ", B[i][j]); // Ausgabe des Matrixwertes B_{ij}
        }                               // Ende der 1.Schleife
        printf("| \n");
    }                                   // Ende der 1.Schleife

    printf("\nDie Adresse des mehrdimensionalen Arrays A entspricht der Referenz seines ersten Elementes:\n");
    printf("&A[0][0] = %p , A[0] = %p und zeiger_A = %p\n\n", &A[0][0], A[0], zeiger_A);
    printf("Der Ausdruck A[i] ist der Zeiger auf die i-te Zeile der Matrix A:\nA[0] = %p , A[1] = %p und A[2] = %p \n\n", A[0], A[1], A[2]);
    printf("Die gesamte Anzahl der Elemente im Array A ist %i \n\n", anz_elem_A);
    printf("Das Array A entspricht einer (m x n)-Matrix mit m = %i und n = %i \n\n", dim_m, dim_n);

    printf("Navigieren in Arrays:\n");
    printf("Der Zugriff auf ein Element des Arrays kann entweder ... \n");
    printf("... durch die Angabe des Matrix-Indexes erfolgen A[i][j]: z.B. A[2][3] = %i \n", A[2][3]);
    printf("... oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen *(A[i] + j): z.B. *(A[2] + 3) = %i \n", *(A[2] + 3));
    printf("    dies ist gleichbedeutend mit *(zeiger_A + i*n + j): z.B. *(zeiger_A + 2*5 + 3) = %i \n\n", *(zeiger_A + 2*5 + 3));
}
```

```cpp
#include <iostream>

int main(){
    const int m = 3;
    const int n = 5;
    int A[m][n] = { { 1, 2, 3, 4, 5},
                    {21,22,23,24,25},
                    {51,52,53,54,55} };
    int B[n][m] = { { 1, 2, 3},
                    { 4, 5, 6},
                    { 7, 8, 9},
                    {10,11,12},
                    {13,14,15} };

    int* zeiger_A = &A[0][0];

    constexpr int anz_elem_A = sizeof
    constexpr int dim_m = sizeof(A)/s
    constexpr int dim_n = sizeof(A[0]

    printf("Matrix A: \n");
    for(int i=0; i < m; ++i){
        printf("| ");
        for(int j=0; j < n; ++j){
            printf("%3i ", A[i][j]);
        }
        printf("| \n");
    }

    printf("\nMatrix B: \n");
    for(int i=0; i < n; ++i){
        printf("| ");
        for(int j=0; j < m; ++j){
            printf("%3i ", B[i][j]);
        }
        printf("| \n");
    }

    printf("\nDie Adresse des mehrdimensionalen Arrays A entspricht der Referenz seines ersten Elementes:\n");
    printf("&A[0][0] = %p , A[0] = %p und zeiger_A = %p\n\n", &A[0][0], A[0], zeiger_A);
    printf("Der Ausdruck A[i] ist der Zeiger auf die i-te Zeile der Matrix A:\nA[0] = %p , A[1] = %p und A[2] = %p \n\n", A[0], A[1], A[2]);
    printf("Die gesamte Anzahl der Elemente im Array A ist %i \n\n", anz_elem_A);
    printf("Das Array A entspricht einer (m x n)-Matrix mit m = %i und n = %i \n\n", dim_m, dim_n);

    printf("Navigieren in Arrays:\n");
    printf("Der Zugriff auf ein Element des Arrays kann entweder ... \n");
    printf("... durch die Angabe des Matrix-Indexes erfolgen A[i][j]: z.B. A[2][3] = %i \n", A[2][3]);
    printf("... oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen *(A[i] + j): z.B. *(A[2] + 3) = %i \n", *(A[2] + 3));
    printf("    dies ist gleichbedeutend mit *(zeiger_A + i*n + j): z.B. *(zeiger_A + 2*5 + 3) = %i \n\n", *(zeiger_A + 2*5 + 3));
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays/MehrdimArray$ g++ Array_mehrdim_zeiger.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays/MehrdimArray$ ./a.out
Matrix A:
|   1   2   3   4   5 |
|  21  22  23  24  25 |
|  51  52  53  54  55 |

Matrix B:
|   1   2   3 |
|   4   5   6 |
|   7   8   9 |
|  10  11  12 |
|  13  14  15 |

Die Adresse des mehrdimensionalen Arrays A entspricht der Referenz seines ersten Elementes:
&A[0][0] = 0x7ffe865214a0 , A[0] = 0x7ffe865214a0 und zeiger_A = 0x7ffe865214a0

Der Ausdruck A[i] ist der Zeiger auf die i-te Zeile der Matrix A:
A[0] = 0x7ffe865214a0 , A[1] = 0x7ffe865214b4 und A[2] = 0x7ffe865214c8

Die gesamte Anzahl der Elemente im Array A ist 15

Das Array A entspricht einer (m x n)-Matrix mit m = 3 und n = 5

Navigieren in Arrays:
Der Zugriff auf ein Element des Arrays kann entweder ...
... durch die Angabe des Matrix-Indexes erfolgen A[i][j]: z.B. A[2][3] = 54
... oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen *(A[i] + j): z.B. *(A[2] + 3) = 54
    dies ist gleichbedeutend mit *(zeiger_A + i*n + j): z.B. *(zeiger_A + 2*5 + 3) = 54
```

Anwendung: Multiplikation zweier Matrizen

Wir möchten nun die beiden Matrizen $\mathcal{A}$ und $\mathcal{B}$ miteinander multiplizieren. Allgemein gilt für die Multiplikation einer $(m \times n)$-Matrix $\mathcal{A}$ mit einer $(n \times l)$-Matrix $\mathcal{B}$ die folgende Formel: $\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$ mit

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj} \quad , \quad i = 1, 2, \ldots, m\,;\; j = 1, 2, \ldots, l \quad ,$$

wobei die entstehende Produktmatrix $\mathcal{C}$ eine $(m \times l)$-Matrix ist. Für unsere speziellen Matrizen gilt

$$\mathcal{C} = \mathcal{A} \cdot \mathcal{B} = \underbrace{\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 21 & 22 & 23 & 24 & 25 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix}}_{(3 \times 5)\text{-Matrix}} \cdot \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix}}_{(5 \times 3)\text{-Matrix}} = \underbrace{\begin{pmatrix} 135 & 150 & 165 \\ 835 & 950 & 1065 \\ 1885 & 2150 & 2415 \end{pmatrix}}_{(3 \times 3)\text{-Matrix}} \quad .$$

Array_mehrdim_mult.cpp

```cpp
/** Matrix Multiplikation
 * (m x l)-Matrix = (m x n)-Matrix * (n x l)-Matrix
 * Spezialfall
 * (m x m)-Matrix = (m x n)-Matrix * (n x m)-Matrix
 * C[m][m] = A[m][n] * B[n][m]
 * hier speziell eine (3 x 5)-Matrix A und eine (5 x 3)-Matrix B
 **/
#include <iostream>                     // Ein- und Ausgabebibliothek

int main(){                             // Hauptfunktion
    const int m = 3;                    // Dimension der Zeilen der Matrix A
    const int n = 5;                    // Dimension der Spalten der Matrix A

    int A[m][n] = { { 1, 2, 3, 4, 5},
                    {21,22,23,24,25},
                    {51,52,53,54,55} }; // Definition eines (3x5)-Integer-Arrays, (3x5)-Matrix

    int B[n][m] = { { 1, 2, 3},
                    { 4, 5, 6},
                    { 7, 8, 9},
                    {10,11,12},
                    {13,14,15} };       // Definition eines (5x3)-Integer-Arrays, (5x3)-Matrix

    int C[m][m];                        // Deklaration eines (3x3)-Integer-Arrays, (3x3)-Matrix

    // Matrix Multiplikation C = A * B
    for(int i=0; i < m; ++i){                       // Anfang Schleife 1 (Zeilen der Matrix C)
        for(int j=0; j < m; ++j){                   // Anfang Schleife 2 (Spalten der Matrix C)
            C[i][j] = 0;                            // Anfangs-Initialisierung von C_{ij}
            for(int k=0; k < n; ++k){               // Anfang Schleife 3 (Summationsschleife)
                C[i][j] = C[i][j] + A[i][k] * B[k][j]; // Matrix Multiplikation
            }                                       // Ende der 3.Schleife
        }                                           // Ende der 2.Schleife
    }                                               // Ende der 1.Schleife

    // Terminal Ausgabe der Matrix C
    printf("Die Matrix Multiplikation ergibt C = A * B : \n");
    for(int i=0; i < m; ++i){           //
        printf("| ");
        for(int j=0; j < m; ++j){       //
            printf("%3i ", C[i][j]);    //
        }                               //
        printf("| \n");
    }                                   //
}
```

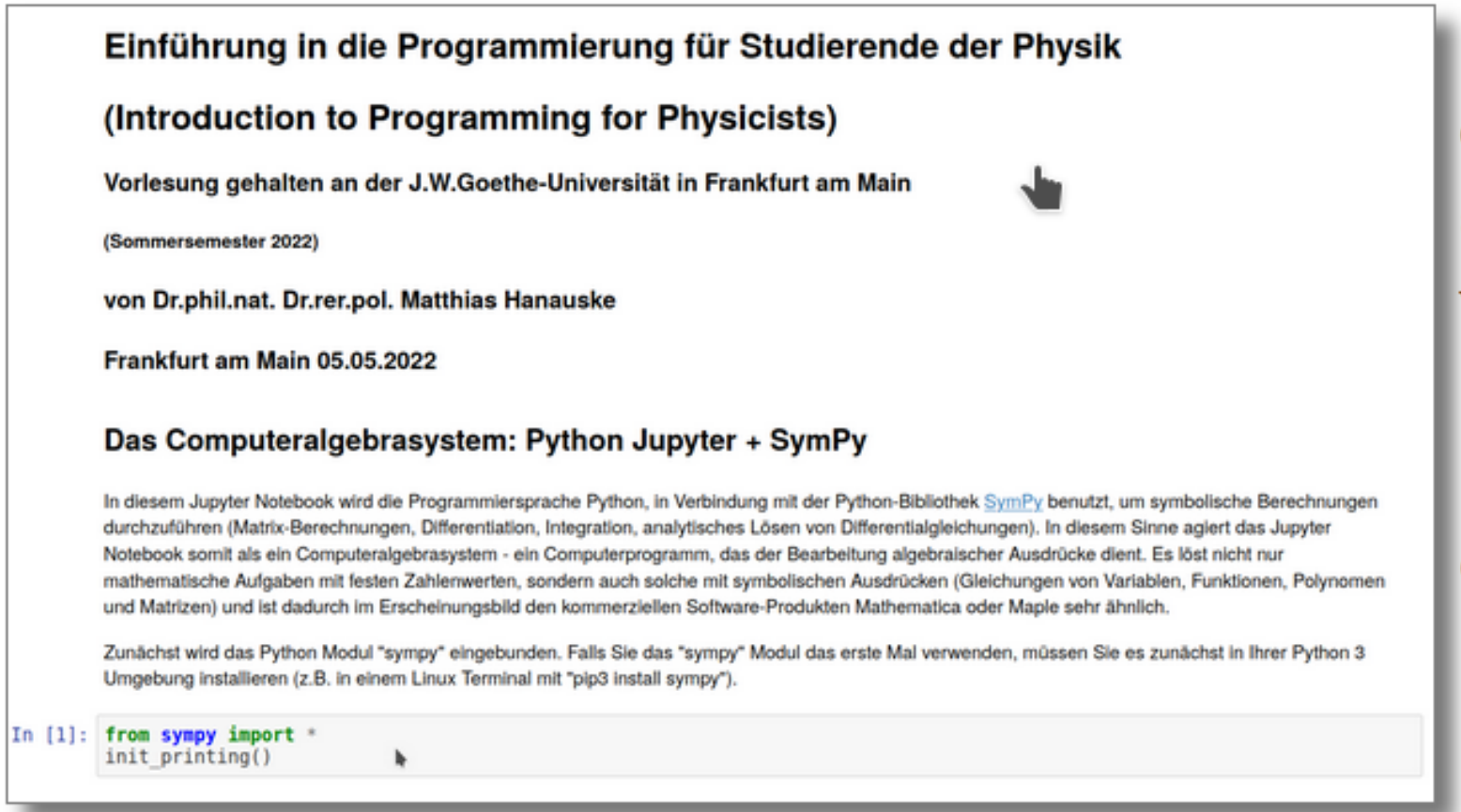
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigPr
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigPr

Matrix A:
|   1   2   3   4   5 |
|  21  22  23  24  25 |
|  51  52  53  54  55 |

Matrix B:
|   1   2   3 |
|   4   5   6 |
|   7   8   9 |
|  10  11  12 |
|  13  14  15 |

Die Matrix Multiplikation ergibt C = A * B :
| 135 150 165 |
| 835 950 1065 |
| 1885 2150 2415 |
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigPr
```

Die Implementierung der eigentlichen Multiplikation im Quelltext wurde mittels der Index-Schreibweise realisiert, da diese dem mathematischen Formalismus sehr ähnelt:

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj} \quad \Longleftrightarrow \quad \textbf{for}(\text{int k=0; k < n; ++k)\{ C[i][j] = C[i][j] + A[i][k] * B[k][j]; \}}$$

Nach der Berechnung der einzelnen Elemente des Arrays wird die Produktmatrix $\mathcal{C}$ schließlich im Terminal ausgegeben.

Jupyter Notebooks und das Rechnen mit symbolischen Ausdrücken



Einführung in die Programmierung für Studierende der Physik

(Introduction to Programming for Physicists)

Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Sommersemester 2022)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 05.05.2022

Das Computeralgebrasystem: Python Jupyter + SymPy

In diesem Jupyter Notebook wird die Programmiersprache Python, in Verbindung mit der Python-Bibliothek SymPy benutzt, um symbolische Berechnungen durchzuführen (Matrix-Berechnungen, Differentiation, Integration, analytisches Lösen von Differentialgleichungen). In diesem Sinne agiert das Jupyter Notebook somit als ein Computeralgebrasystem - ein Computerprogramm, das der Bearbeitung algebraischer Ausdrücke dient. Es löst nicht nur mathematische Aufgaben mit festen Zahlenwerten, sondern auch solche mit symbolischen Ausdrücken (Gleichungen von Variablen, Funktionen, Polynomen und Matrizen) und ist dadurch im Erscheinungsbild den kommerziellen Software-Produkten Mathematica oder Maple sehr ähnlich.

Zunächst wird das Python Modul "sympy" eingebunden. Falls Sie das "sympy" Modul das erste Mal verwenden, müssen Sie es zunächst in Ihrer Python 3 Umgebung installieren (z.B. in einem Linux Terminal mit "pip3 install sympy").

```
In [1]: from sympy import *
        init_printing()
```

Ein Computeralgebrasystem ist ein Computerprogramm, das der Bearbeitung algebraischer Ausdrücke dient. Es löst nicht nur mathematische Aufgaben mit festen Zahlenwerten, sondern auch solche mit symbolischen Ausdrücken (Gleichungen von Variablen, Funktionen, Polynomen und Matrizen). In diesem Unterkapitel wird die Programmiersprache Python, in Verbindung mit der Python-Bibliothek SymPy benutzt, um symbolische Berechnungen durchzuführen (Matrix-Berechnungen, Differentiation, Integration, analytisches Lösen von Differentialgleichungen). In diesem Sinne agiert das Jupyter Notebook als ein Computeralgebrasystem und es löst nicht nur mathematische Aufgaben mit festen Zahlenwerten, sondern auch solche mit symbolischen Ausdrücken (Gleichungen von Variablen, Funktionen, Polynomen und Matrizen). Beim Klicken auf das nebenstehende Bild gelangt man zu dem Jupyter Notebook Das Computeralgebrasystem: Python Jupyter + SymPy ('Jupyter_sympy.ipynb' , View Notebook, Download Notebook). Am Ende des Notebooks wird unter anderen auch der Anwendungsfall der *Herleitung der Differentiationsregeln mittels der Methode der Lagrange Polynome* behandelt, die dann im nächsten Unterpunkt bei der *Numerische Differentiation* benutzt werden.

Einführung in die Programmierung für Studierende der Physik

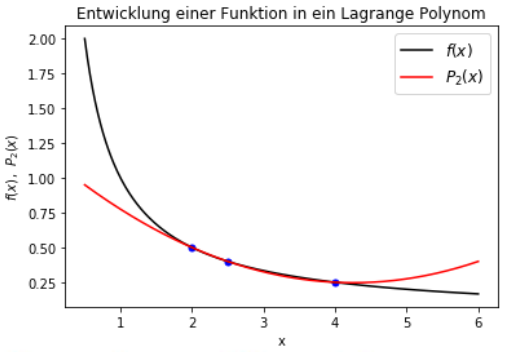
(Introduction to Programming for Physicists)

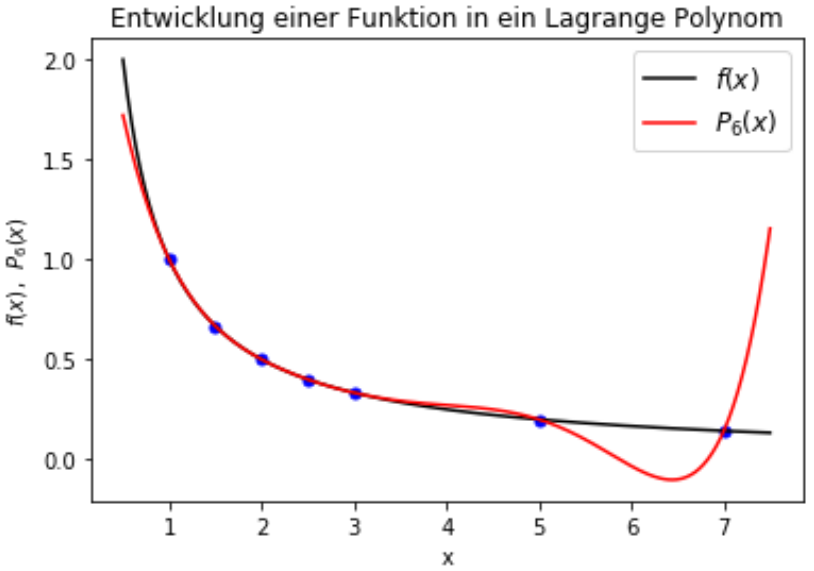
Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(Sommersemester 2022)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 05.05.2022





Das Computeralgebrasystem: Python Jupyter + SymPy

In diesem Jupyter Notebook wird die Programmiersprache Python, in Verbindung mit der Python-Bibliothek SymPy benutzt, um symbolische Berechnungen durchzuführen (Matrix-Berechnungen, Differentiation, Integration, analytisches Lösen von Differentialgleichungen). In diesem Sinne agiert das Jupyter Notebook somit als ein Computeralgebrasystem - ein Computerprogramm, das der Bearbeitung algebraischer Ausdrücke dient. Es löst nicht nur mathematische Aufgaben mit festen Zahlenwerten, sondern auch solche mit symbolischen Ausdrücken (Gleichungen von Variablen, Funktionen, Polynomen und Matrizen) und ist dadurch im Erscheinungsbild den kommerziellen Software-Produkten Mathematica oder Maple sehr ähnlich.

Zunächst wird das Python Modul "sympy" eingebunden. Falls Sie das "sympy" Modul das erste Mal verwenden, müssen Sie es zunächst in Ihrer Python 3 Umgebung installieren (z.B. in einem Linux Terminal mit "pip3 install sympy").

In [1]:
```
from sympy import *
init_printing()
```

Anwendungsbeispiel:
Herleitung der Differentiationsregeln mittels der Methode der Lagrange Polynome

Numerische Differentiation

In diesem Unterpunkt werden wir die im Jupyter Notebook Das Computeralgebrasystem: Python Jupyter + SymPy ('Jupyter_sympy.ipynb' , View Notebook, Download Notebook) hergeleiteten *Differentiationsregeln der numerischen Mathematik* in einem C++ Programm anwenden. Im Speziellen werden wir die numerische Ableitung der Funktion $f(x) = 10\,e^{-x/10} \cdot \sin(x)$ mittels der unterschiedlichen Differentiationsformel in einem C++ Programm bestimmen und die Ergebnisse mit der wirklichen, analytisch bestimmbaren Ableitung $f'(x)$ vergleichen.
Die mathematische Definition der Ableitung einer Funktion $f(x)$ an der Stelle x_0, der sogenannte Differentialquotient, benutzt die folgende Grenzwert-Formulierung ($h \to 0$):

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \quad .$$

In der numerischen Mathematik hingegen werden mehrere *Differentiationsregeln* formuliert, die diesen wirklichen Wert der Ableitung approximieren. Die analytische Herleitung dieser Regeln benutzt die *Methode der Lagrange Polynome* (siehe Anwendungsbeispiel: Interpolation und Polynomapproximation).

Die Differentiationsregeln der numerischen Mathematik

Die im vorigen Unterpunkt hergeleiteten *Differentiationsregeln der numerischen Mathematik* (siehe Das Computeralgebrasystem: Python Jupyter + SymPy) sind in der folgenden Box nochmals zusammenfassend dargestellt:

Zweipunkteformel (n=1):	$f'(x_0) \approx \frac{1}{h}[f(x_0 + h) - f(x_0)]$	, Stützstellen: $(x_0, x_0 + h)$
Dreipunkte-Endpunkt-Formel (n=2):	$f'(x_0) \approx \frac{1}{2h}[-3\,f(x_0) + 4\,f(x_0 + h) - f(x_0 + 2\,h)]$	, Stützstellen: $(x_0, x_0 + h, x_0 + 2h)$
Dreipunkte-Mittelpunkt-Formel (n=2):	$f'(x_0) \approx \frac{1}{2h}[f(x_0 + h) - f(x_0 - h)]$	, Stützstellen: $(x_0 - h, x_0, x_0 + h)$
Fünfpunkte-Mittelpunkt-Formel (n=4):	$f'(x_0) \approx \frac{1}{12\,h}[f(x_0 - 2\,h) - 8\,f(x_0 - h) + 8\,f(x_0 + h) - f(x_0 + 2\,h)]$	, Stützstellen: $(x_0 - 2h, x_0 - h, x_0, x_0 + h, x_0 + 2h)$

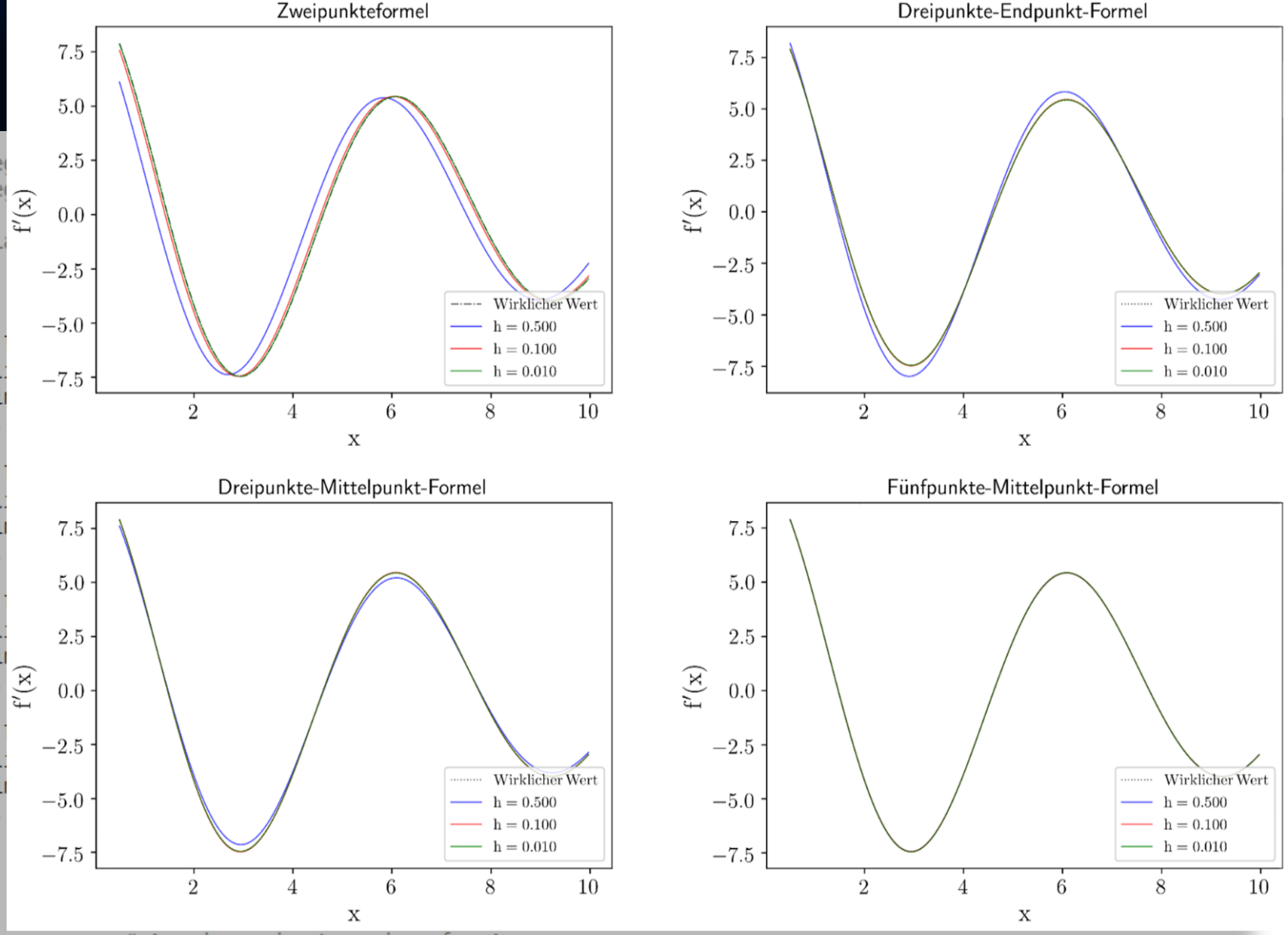
In dem folgenden C++ Programm wird die Ableitung der Funktion $f(x) = 10\,e^{-x/10} \cdot \sin(x)$ an der Stelle $x_0 = 3$ numerisch approximativ berechnet.

Numerical_Differentiation_1

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V6$ g++ Numerical_Differentiation_1.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V6$ ./a.out
h-Wert:                          h =   0.0100000000
Wirklicher Wert:                 f'(x=3.000) =  -7.4385890715
Zweipunkteformel:                f'(x=3.000) =  -7.4363062720 , absoluter Fehler:  -0.0022827995
Dreipunkte-Endpunkt-Formel:      f'(x=3.000) =  -7.4388361365 , absoluter Fehler:   0.0002470650
Dreipunkte-Mittelpunkt-Formel:   f'(x=3.000) =  -7.4384652952 , absoluter Fehler:  -0.0001237763
Fuenfpunkte-Mittelpunkt-Formel:  f'(x=3.000) =  -7.4385890691 , absoluter Fehler:  -0.0000000024
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V6$
```

```cpp
/* Berechnung der Ableitung f'(x
 * mittels unterschiedlich genau
 * (Zweipunkteformel, Dreipunkte
#include <stdio.h>
#include <cmath>

double f(double x){
    double wert;
    wert=10*sin(x)*exp(-x/10);
    return wert;
}                                                   // Ende der Funktion f(x)

double f_strich(double x){                          // Deklaration und Definition der Ableitung f'(x) der Funktion f(x)
    double wert;                                    // (f'(x) wird nur zum Vergleich mit den Ergebnissen benoetigt)
    wert=10*cos(x)*exp(-x/10) - sin(x)*exp(-x/10);  // Eigentliche Definition der Ableitung
    return wert;                                    // Rueckgabewert der Funktion f'(x)
}                                                   // Ende der Funktion f'(x)

int main(){                                         // Hauptfunktion
    double x = 3;                                   // Aktueller x-Wert an dem die Ableitung berechnet wird
    double h=0.01;                                  // Aequidistanter Abstand zwischen den x-Werten die zur numerischen Differentation benutzt werden
    double f1_strich_p;                             // Deklaration der f'(x)-Ausgabe-Punkte (Zweipunkteformel)
    double f3a_strich_p;                            // Deklaration der f'(x)-Ausgabe-Punkte (Dreipunkte-Endpunkt-Formel)
    double f3b_strich_p;                            // Deklaration der f'(x)-Ausgabe-Punkte (Dreipunkte-Mittelpunkt-Formel)
    double f5_strich_p;                             // Deklaration der f'(x)-Ausgabe-Punkte (Fünfpunkte-Mittelpunkt-Formel)

    f1_strich_p = (f(x+h) - f(x))/h;                // Zweipunkteformel
    f3a_strich_p = (-3*f(x) + 4*f(x+h) - f(x+2*h))/(2*h);           // Dreipunkte-Endpunkt-Formel
    f3b_strich_p = (f(x+h) - f(x-h))/(2*h);                         // Dreipunkte-Mittelpunkt-Formel
    f5_strich_p = (f(x-2*h) - 8*f(x-h) + 8*f(x+h) - f(x+2*h))/(12*h); // Fünfpunkte-Mittelpunkt-Formel

    printf("h-Wert:                          h = %14.10f \n", h);
    printf("Wirklicher Wert:                 f'(x=%5.3f) = %14.10f \n", x, f_strich(x));                                                  // Ausgabe der berechneten Werte
    printf("Zweipunkteformel:                f'(x=%5.3f) = %14.10f , absoluter Fehler: %14.10f \n", x, f1_strich_p, f_strich(x) - f1_strich_p);   // Ausgabe der berechneten Werte
    printf("Dreipunkte-Endpunkt-Formel:      f'(x=%5.3f) = %14.10f , absoluter Fehler: %14.10f \n", x, f3a_strich_p, f_strich(x) - f3a_strich_p); // Ausgabe der berechneten Werte
    printf("Dreipunkte-Mittelpunkt-Formel:   f'(x=%5.3f) = %14.10f , absoluter Fehler: %14.10f \n", x, f3b_strich_p, f_strich(x) - f3b_strich_p); // Ausgabe der berechneten Werte
    printf("Fuenfpunkte-Mittelpunkt-Formel:  f'(x=%5.3f) = %14.10f , absoluter Fehler: %14.10f \n", x, f5_strich_p, f_strich(x) - f5_strich_p);   // Ausgabe der berechneten Werte

}                                                   // Ende der Hauptfunktion
```

```
l_width=0.8                                                    # Festle
alp=0.7                                                        # Festle

label_0=r'$\rm h='+'{0:.3f}'.format(h_list[0])+'$'             # Plot-L
label_1=r'$\rm h='+'{0:.3f}'.format(h_list[1])+'$'
label_2=r'$\rm h='+'{0:.3f}'.format(h_list[2])+'$'

ax1.plot(data[:,1],data[:,2], color="black", linewidth=l_width,
ax1.plot(data[:,1],data[:,3], color="blue", linewidth=l_width, l
ax1.plot(data[:,1],data[:,7], color="red", linewidth=l_width, li
ax1.plot(data[:,1],data[:,11], color="green", linewidth=l_width,

ax2.plot(data[:,1],data[:,2], color="black", linewidth=l_width,
ax2.plot(data[:,1],data[:,4], color="blue", linewidth=l_width, l
ax2.plot(data[:,1],data[:,8], color="red", linewidth=l_width, li
ax2.plot(data[:,1],data[:,12], color="green", linewidth=l_width,

ax3.plot(data[:,1],data[:,2], color="black", linewidth=l_width,
ax3.plot(data[:,1],data[:,5], color="blue", linewidth=l_width, l
ax3.plot(data[:,1],data[:,8], color="red", linewidth=l_width, li
ax3.plot(data[:,1],data[:,12], color="green", linewidth=l_width,

ax4.plot(data[:,1],data[:,2], color="black", linewidth=l_width,
ax4.plot(data[:,1],data[:,6], color="blue", linewidth=l_width, l
ax4.plot(data[:,1],data[:,9], color="red", linewidth=l_width, li
ax4.plot(data[:,1],data[:,13], color="green", linewidth=l_width,

ax1.legend(frameon=True, loc="lower right",fontsize=10)
ax2.legend(frameon=True, loc="lower right",fontsize=10)
ax3.legend(frameon=True, loc="lower right",fontsize=10)
ax4.legend(frameon=True, loc="lower right",fontsize=10)

plt.savefig("Numerical_Differentiation_2.png", dpi=400, bbox_inches="tight", pad_inches=0.05, format="png") # Speichern der Abbildung als Bild
plt.show()                                                     # Zusaetzliches Darstellen der Abbildung in einem separaten Fenster
```



Anordnung der Legende auf ax4

Übungsblatt Nr. 6

Aufgabe 1 *(6.5 Punkte)*

In der Aufgabe 4 des Übungsblattes Nr. 3 hatten wir die ersten 40 Zahlenwerte der Fibonacci-Folge mittels einer for-Schleife berechnet. Die Fibonacci-Folge $f_n, n = [1, 2, 3, \dots]$ wurde dabei durch folgendes rekursives Bildungsgesetz definiert:

$$f_n = f_{n-1} + f_{n-2} \ , \quad \text{mit den Anfangswerten: } f_1 = f_2 = 1$$

Die Implementierung dieses Bildungsgesetzes in einem C++ Programm können wir nun eleganter, mittels eines eindimensionalen C++ Arrays formulieren. Dies hätte zusätzlich den Vorteil hätte, dass die Indexschreibweise der mathematischen Formulierung des Bildungsgesetzes in einer äquivalenten Darstellung im C++ Quelltext stehen würde, und somit die Verständlichkeit des Quelltextes erleichtert. Deklarieren Sie ein eindimensionales Array der Folgenglieder f_n und speichern Sie sich die ersten 40 Fibonacci-Zahlen in diesem Array. Benutzen Sie bei der Implementierung des rekursiven Bildungsgesetzes einen Indexbasierten Zugriff auf die Elemente des Arrays. Zeigen Sie, dass das Verhältnis zweier aufeinanderfolgender Zahlen der Fibonacci-Folge $(\frac{f_n}{f_{n-1}})$ im Grenzwert $\lim_{n\to\infty}$ gegen die irrationale Zahl des Goldenen Schnitts $\Phi \approx 1.6180339887498948482045868 34$ konvergiert. Bemerkung: Der Goldene Schnitt ist in vielen Bereichen der Mathematik, Kunst, Architektur und Biologie von Bedeutung (näheres siehe z.B. Wikipedia: Goldene Schnitt).

Aufgabe 2 *(7 Punkte)*

Berechnen Sie für die unten angegebenen Vektoren $\vec{a}$ und $\vec{b}$ die folgenden Größen:

$$\vec{a} = \begin{pmatrix} 2.3 \\ -1.36 \\ 6.91 \end{pmatrix} \quad , \quad \vec{b} = \begin{pmatrix} 10.3 \\ -4.34 \\ 5.3 \end{pmatrix} \quad , \quad \text{Skalarprodukt: } s = \vec{a} \cdot \vec{b} \quad , \quad \text{Kreuzprodukt: } \vec{c} = \vec{a} \times \vec{b}$$

Definieren Sie dazu zwei eindimensionale C++ Arrays für die Vektoren $\vec{a}$ und $\vec{b}$ und berechnen Sie die das Skalar- und Kreuzprodukt der Vektoren möglichst mittels der folgenden Indexbasierten Ausdrücke

$$\text{Skalarprodukt: } s = \vec{a} \cdot \vec{b} = \sum_{i=1}^{3} a_i \, b_i, \quad , \quad \text{Kreuzprodukt: } \vec{c} = \vec{a} \times \vec{b} = \sum_{i,j,k=1}^{3} \epsilon_{ijk} \, a_i \, b_j \, \vec{e}_k \quad ,$$

wobei ϵ_{ijk} der total antisymmetrischer Tensor (auch Epsilon-Tensor bzw. Levi-Civita-Symbol) ist. Definieren Sie das Levi-Civita-Symbol bitte als ein mehrdimensionales C++ Array (der Epsilon-Tensor entspricht einer $(3 \times 3 \times 3)$-Matrix). Verwenden Sie bei der Implementierung des Skalar- und Kreuzproduktes sowohl eine Formulierung unter Zuhilfenahme eines Index-Zugriff auf die einzelnen Elemente, als auch eine Zeiger-basierte Zugriffsweise. Lassen Sie sich die berechneten Größen im Terminal ausgeben.

Aufgabe 3 *(6.5 Punkte)*

In der Vorlesung 4 im Unterpunkt Anwendungsbeispiel: Nullstellensuche einer Funktion hatten wir die *Methode der Bisektion* (das *Intervallhalbierungsverfahren*) und die *Newton-Raphson Methode* der Nullstellenermittlung vorgestellt. Die *Newton-Raphson Methode* stellte hierbei ein sehr effektives Verfahren zur Ermittlung einer Nullstelle dar, hatte jedoch den Nachteil, dass man neben der Funktion selbst auch noch die Ableitung $f'(x)$ der Funktion benötigte. Da wir nun die Ableitung einer Funktion auch numerisch bestimmen können (siehe Teilkapitel Numerische Differentiation), können wir das *Newton-Raphson Verfahren* insofern abändern, dass der Benutzer lediglich die Funktion $f(x)$ definieren muss und die benötigte Ableitung, mittels der hergeleiteten *Differentiationsregeln der numerischen Mathematik* berechnet werden.

Schreiben Sie ein C++ Programm, welches die *Newton-Raphson Methode* zur Ermittlung einer Nullstelle benutzt und lediglich den Ausdruck der Funktion $f(x)$ benötigt (benutzen Sie speziell $f(x) = e^x - 20$). Für den approximativen Wert der Ableitung verwenden Sie bitte einerseits die 'Dreipunkte-Mittelpunkt-Formel' und die 'Fünfpunkte-Mittelpunkt-Formel' und zusätzlich vergleichen Sie die Ergebnisse mit einer Berechnung, die den analytischen Ausdruck für $f'(x)$ benutzt. Lassen Sie sich die approximierten Nullstellenwerte der ersten 10 Iterationen dieser modifizierten Newton-Raphson Methode für die drei Varianten im Terminal ausgeben und geben Sie zusätzlich den relativen Fehler zum wirklichen Wert an. Benutzen Sie bei allen Varianten den geratenen Startwert $p_0 = 2$ bei der Nullstellenberechnung und einen h-Wert von $h = 0.1$ bei der approximativen Bestimmung der Ableitung.

Übungsblatt Nr. 7

Aufgabe 1 *(10 Punkte)*

Im Folgenden soll ein Teil einer Kurvendiskussion mittels eines eigenen Jupyter Notebooks in Verbindung mit einem C++ Programm angefertigt werden. Gegeben sei die Funktion $f(x)$

$$f(x) = e^{-x^2/4} \cdot \sin(x/10) \cdot \left(x^4 - q\,x^2 - 5\right) \quad .$$

Nullstellensuche mit Jupyter und Sympy

Betrachten Sie die Funktion in ihrem gesamten Definitionsbereich ($x \in \mathbb{R}$) und berechnen Sie für beliebige Parameter $q \in \mathbb{R}$ die Nullstellen der Funktion $f(x) = 0$. Wie lauten die Nullstellenwerte für $q = 9$?

Darstellung von $f(x)$, $f'(x)$ und $f''(x)$ mittels Jupyter

Stellen Sie nun die Funktion $f(x)$ und die erste und zweite Ableitung der Funktion im Teilintervall $x \in [-7, 7]$ grafisch in drei x-y-Diagrammen dar (benutzen Sie dafür wieder den festen Parameterwert $q = 9$). Versuchen Sie die Maxima der Funktion (mit $q = 9$) mittels Sympy zu bestimmen.

C++ Programm zur Bestimmung der Hochpunkte der Funktion

Berechnen Sie die x-Werte der beiden Hochpunkte der Funktion $f(x)$ mittels der Methode der Bisektion (Intervallhalbierungsverfahren) unter Verwendung eines C++ Programms.

Aufgabe 2 *(10 Punkte)*

Diese Aufgabe ist angelehnt an das Kapitel 23 "Der gedämpfte harmonische Oszillator" des Buches von Prof. Walter Greiner, Mechanik (Teil 1) [5. Auflage, 1989, siehe Seite 226- 237]. Siehe auch Vorlesungsskript von Prof. Rischke auf Seite 117- 126 http://www.th.physik.uni-frankfurt.de/~drischke/Skript_MI_WiSe2016-2017.pdf). Wir betrachten im Folgenden den gedämpften harmonischen Oszillator am Beispiel eines reibungsfrei gelagerten Wagens (Masse=M) auf den eine Rückstellkraft einwirkt (die proportional zu seiner Auslenkung x ist (Proportionalitätskonstante k)), wobei zusätzlich eine geschwindigkeitsabhängige Reibungskraft auf den Wagen einwirkt (z.B. verursacht durch den auf den Wagen einwirkende Luftwiderstand, Stokesscher Ansatz: Proportionalitätskonstante α). Aufgrund der Rückstellkraft, besitzt das zugrundeliegende Potential $V(x)$ die Form einer Parabel $V(x) = \frac{k\,x^2}{2}$. Die Differentialgleichung des linearen harmonischen Oszillators mit Dämpfung wird mittels der folgenden Differentialgleichung zweiter Ordnung beschrieben (wir setzen $\omega_0^2 = \frac{k}{M}$ und $\beta = \frac{\alpha}{2M}$):

$$\ddot{x}(t) = -\omega_0^2\, x(t) - 2\beta\, \dot{x}(t)$$

Die Anfangsbedingungen seien zunächst noch allgemein gehalten: $x(0) = \alpha_1$, $\dot{x}(0) = \alpha_2$. Bestimmen Sie die allgemeine Lösung der Differentialgleichung mittels eines eigenen Jupyter Notebooks. Geben Sie dann die spezielle Lösung der Differentialgleichung bei festgelegten Parameterwerten ($\omega_0^2 = 3$ und $\beta = 0.25$) und Anfangsbedingungen ($\alpha_1 = 0$ und $\alpha_2 = 40$) an und visualisieren Sie diese in einem x-t Diagramm. An welchem Ort befindet sich der Wagen zur Zeit $t = 10$ ($x(10)$)?

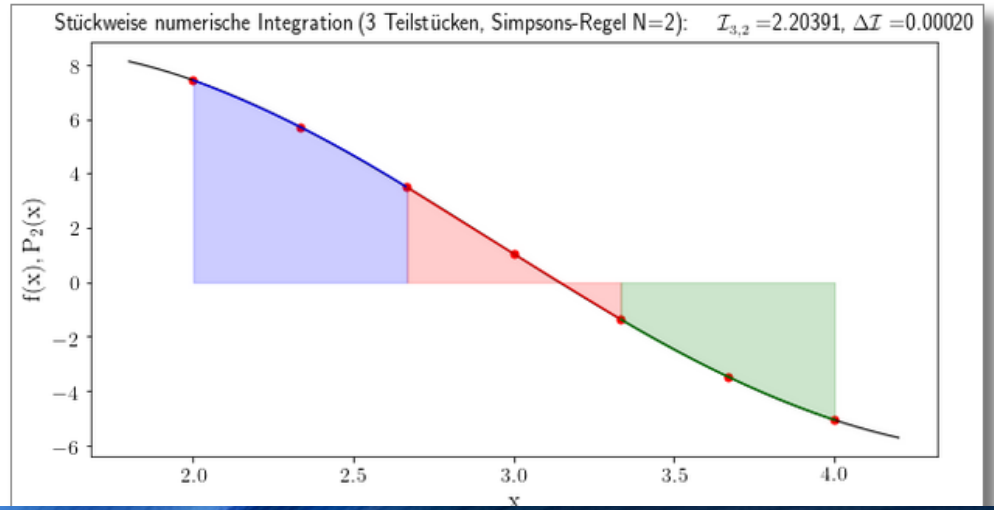
Vorlesung 7

In dieser Vorlesung werden wir den Programmier- und Programmentwurfstil der *objektorientierten Programmierung* kennenlernen. Die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf dem Konzept der *Klasse*. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird. Außerdem werden wir, nachdem wir in einem Jupyter Notebook die Integrationsregeln hergeleitet haben, den Anwendungsfall der numerischen Integration betrachten.

Objekt-orientierte Programmierung und C++ Klassen

Die meisten Programmiertechniken, die wir bis jetzt kennengelernt haben, verwendeten den Programmentwurfstil der *prozeduralen Programmierung*. Wir werden nun den Fokus auf die Strukturierung von Programmen legen (das Programmierparadigma der objektorientierten Programmierung) und auf das in C++ integrierte Klassenkonzept eingehen. Das Konzept der objektorientierten Programmierung beruht auf der alltäglichen Erfahrung, dass man Objekte nach zwei Maßstäben beurteilt: Ein Objekt besitzt einerseits messbare Eigenschaften und ist aber auch andererseits über seine Verhaltensweisen definiert. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird und die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf diesem Konzept der *Klasse*. In einer C++ Klasse werden die messbaren Eigenschaften des Objektes in Instanzvariablen (Daten-Member) gespeichert und durch Konstruktoren werden diese Daten-Member dann initialisiert. Die Verhaltensweisen des Objektes werden durch klasseninterne Funktionen, die sogenannten Member-Funktionen beschrieben (näheres siehe Objekt-orientierte Programmierung und C++ Klassen).

Theorie: Numerische Integration



Wir betrachten in diesem Unterpunkt die Methode der numerischen Integration mittels der "Geschlossenen Newton-Cotes Gleichungen" (closed Newton-Cotes formulars). Die Vorgehensweise der Herleitung dieser Gleichungen erfolgt, indem man die zu integrierende Funktion $f(x)$ in ein Lagrange Polynom vom Grade N ($P_N(x)$) entwickelt ((siehe Anwendungsbeispiel: Interpolation und Polynomapproximation)) und dann durch analytische Integration zur Approximation gelangt. Beim Klicken auf die nebenstehende Abbildung gelangen Sie zu einem

Vorlesung 7

Mittels der bisher erlernten Programmierkonzepte können wir bereits viele umfangreiche Berechnungen durchführen und Sie werden auch bald an Ihrem eigenen Programmier-Projekt arbeiten. Als Programmierer eines umfangreichen Programmes kommt man sich manchmal wie ein Schöpfer einer neuen fiktiven, virtuellen Kreatur vor und in dieser Vorlesung werden wir eine ganz neue Art von Herangehensweise bei der Erschaffung eines Programmes erlernen. Mittels eigener, dem Problem angepasster Programmierobjekte ist es durch einen Abstraktionsmechanismus möglich, eine geordnete Struktur in den Ideenreichtum eines C++ Quelltextes bringen. Eine Klasse stellt dabei den Bauplan für das zu konstruierende Objekt bereit und die wirkliche Realisierung des Objektes (die Instanzbildung) findet dann im Hauptprogramm zur Laufzeit statt. Eine Klasse stellt somit eine formale Beschreibung dar, wie das Objekt beschaffen ist, d.h. welche Merkmale (Instanzvariablen bzw. Daten-Member der Klasse) und Verhaltensweisen (Methoden der Klasse bzw. Member-Funktionen) das zu beschreibende Objekt hat. Eine Klasse ist also eine Vorlage, eine abstrakte Idee, die ein Grundgerüst von Eigenschaften und Methoden vorgibt. Die Erzeugung eines Objektes dieser Klasse entspricht der Materialisierung dieser Idee im Programm. Bei der Erzeugung (Materialisierung) des Objektes wird der sogenannte *Konstruktor* der Klasse aufgerufen, und verlässt das Objekt den Gültigkeitsbereich seines Teilbereiches des Programms, wird es durch den sogenannten *Destruktor* wieder zerstört. Das Grundgerüst einer Klasse besitzt die folgende Form (siehe untere Box), wobei im Anweisungsblock der Klasse nicht alle der aufgezählten Größen deklariert bzw. definiert werden müssen.

```
class Klassenname {
    Private Instanzvariablen (Daten-Member)

    public:
        Konstruktoren
        Member-Funktionen
        (Destruktor)
};
```

Objekt-orientierte Programmierung und C++ Klassen

Einführung

Die C++ Typen, die wir bisher kennengelernt hatten (z.B. int i, double a, int v[3], double A[4][5]), die sogenannten *integrierten Typen*, werden wir nun mittels eines *Abstraktionsmechanismus* erweitern, um eigenen, benutzerdefinierte Typen zu erstellen. Ein *benutzerdefinierter Typ*, wie z.B. die C++ Struktur '**struct**' oder die C++ Klasse '**class**', ist ein Abstraktionskonzept, das den Quelltext eines C+ Programms übersichtlicher macht, indem es das Programm in voneinander separierbare Teilbereiche aufteilt. Große Programme bestehen oft aus einzelnen Teilaufgaben, die man mittels einer sinnvollen Klassenstruktur voneinander trennen und ordnen kann. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird und die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf diesem Konzept der *Klasse*.

Benutzerdefinierte Typen und Abstraktionsmechanismen in C++

Das Konzept der *objektorientierten Programmierung* beruht auf der alltäglichen Erfahrung, dass man Objekte nach zwei Maßstäben beurteilt: Ein Objekt besitzt einerseits messbare Eigenschaften (z.B. Farbe, Gewicht, ...) und ist aber auch andererseits über seine Verhaltensweisen (z.B. zeitliches Verhalten, Interaktionsverhalten, Bewegungsverhalten, ...) definiert. Eine Klasse ist ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von realen/fiktiven Objekten (Klassifizierung). Mittels des Konzeptes der Klasse lassen sich solche Objekte im Programm realisieren. Eine Klasse stellt dabei den Bauplan für das zu beschreibende Objekt bereit und die wirkliche Realisierung des Objektes (die Instanzbildung) findet dann im Hauptprogramm zur Laufzeit statt. Eine Klasse stellt somit eine formale Beschreibung dar, wie das Objekt beschaffen ist, d.h. welche Merkmale (Instanzvariablen bzw. Daten-Member der Klasse) und Verhaltensweisen (Methoden der Klasse bzw. Member-Funktionen) das zu beschreibende Objekt hat. Eine Klasse ist also eine Vorlage, eine abstrakte Idee, die ein Grundgerüst von Eigenschaften und Methoden vorgibt. Die Erzeugung eines Objektes dieser Klasse entspricht der Materialisierung dieser Idee im Programm. Bei der Erzeugung des Objektes wird der sogenannte *Konstruktor* der Klasse aufgerufen, und verlässt das Objekt den Gültigkeitsbereich seines Teilbereiches des Programms, wird es durch den sogenannten *Destruktor* wieder zerstört. Das Grundgerüst einer Klasse besitzt die folgende Form, wobei im Anweisungsblock der Klasse nicht alle der aufgezählten Größen definiert werden müssen.

Objekt-orientierte Programmierung und C++ Klassen

Die C++ Typen, die wir bisher kennengelernt hatten (z.B. int i, double a, int v[3], double A[4][5]), die sogenannten *integrierten Typen*, werden wir nun mittels eines *Abstraktionsmechanismus* erweitern, um eigenen, benutzerdefinierte Typen zu erstellen. Ein *benutzerdefinierter Typ*, wie z.B. die C++ Struktur '**struct**' oder die C++ Klasse '**class**', ist ein Abstraktionskonzept, das den Quelltext eines C+ Programms übersichtlicher macht, indem es das Programm in voneinander separierbare Teilbereiche aufteilt. Große Programme bestehen oft aus einzelnen Teilaufgaben, die man mittels einer sinnvollen Klassenstruktur voneinander trennen und ordnen kann. Eine *C++ Klasse* ist ein benutzerdefinierter neuer Datentyp, der durch das Schlüsselwort '**class**' gekennzeichnet wird und die gesamte Idee der objektorientierten Programmierung beruht gänzlich auf diesem Konzept der *Klasse*.

Das Konzept der *objektorientierten Programmierung* beruht auf der alltäglichen Erfahrung, dass man Objekte nach zwei Maßstäben beurteilt: Ein Objekt besitzt einerseits messbare Eigenschaften (z.B. Farbe, Gewicht, ...) und ist aber auch andererseits über seine Verhaltensweisen (z.B. zeitliches Verhalten, Interaktionsverhalten, Bewegungsverhalten, ...) definiert. Eine Klasse ist ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von realen/fiktiven Objekten (Klassifizierung). Mittels des Konzeptes der Klasse lassen sich solche Objekte im Programm realisieren. Eine Klasse stellt dabei den Bauplan für das zu beschreibende Objekt bereit und die wirkliche Realisierung des Objektes (die Instanzbildung) findet dann im Hauptprogramm zur Laufzeit statt.

Benutzerdefinierte Typen und Abstraktionsmechanismen in C++

Eine Klasse stellt somit eine formale Beschreibung dar, wie das Objekt beschaffen ist, d.h. welche Merkmale (Instanzvariablen bzw. Daten-Member der Klasse) und Verhaltensweisen (Methoden der Klasse bzw. Member-Funktionen) das zu beschreibende Objekt hat. Eine Klasse ist also eine Vorlage, eine abstrakte Idee, die ein Grundgerüst von Eigenschaften und Methoden vorgibt. Die Erzeugung eines Objektes dieser Klasse entspricht der Materialisierung dieser Idee im Programm. Bei der Erzeugung des Objektes wird der sogenannte *Konstruktor* der Klasse aufgerufen, und verlässt das Objekt den Gültigkeitsbereich seines Teilbereiches des Programms, wird es durch den sogenannten *Destruktor* wieder zerstört. Das Grundgerüst einer Klasse besitzt die folgende Form, wobei im Anweisungsblock der Klasse nicht alle der aufgezählten Größen definiert werden müssen.

```
class Klassenname { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };
```

C++ Klassen: Zugriffskontrolle und die öffentlich zugänglichen Bereichen eines Objekte

Eine weitere wichtige Klassen-Terminologie ist die Kennzeichnung von privaten und öffentlich zugänglichen Bereichen des Objektes. In einer Klasse werden die Daten-Member und Member-Funktionen nach außen gekapselt, sodass der Benutzer der Klasse sie nicht manipulieren kann (**private**-Bereiche der Klasse). Kennzeichnet man einen Bereich der Klasse jedoch als **public**, so kann man von außen auf die Daten und Methoden zugreifen und sie auch verändern.

```
class Klassenname {
    // Private Instanzvariablen (Daten-Member) der Klasse
    ...

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Standard-Konstruktor und überladene Konstruktoren der Klasse
        ...

        // Member-Funktionen der Klasse
        ...
};
```

Neben diesen beiden Klassifizierungsbegriffen gibt es zusätzlich die Kennzeichnung **protected**. Besitzt eine Klasse keine explizite Kennzeichnung von privaten und öffentlich zugänglichen Bereichen, so sind alle Merkmale der Klasse privat. Bei der Verwendung der C++ Struktur '**struct**' sind hingegen alle Merkmale öffentlich und man kann '**struct**' somit als eine öffentliche '**class**' ansehen. Die nebenstehende Abbildung veranschaulicht die Schreibweise einer C++ Klasse im Quellcode, wobei gewöhnlicherweise zunächst die privaten und dann die als öffentlich gekennzeichneten Definitionen und Anweisungen folgen.

class Klassenname { 'Anweisungsblock: Instanzvariablen (Daten-Member), Konstruktoren, Member-Funktionen, Destruktor' };

Merkmale von C++ Klassen: Daten-Member und Member-Funktionen

Daten und Funktionen, die in einer Klassendefinition deklariert werden, bezeichnet man als *Daten-Member* (Instanzvariablen) und *Member-Funktionen* (Klassen-interne Funktionen). Durch die Bezeichner private, protected und public findet eine Kapselung der Klassen-internen Merkmale von den anderen Bereichen des C++ Programmes statt. Der Zugriff auf die privaten Eigenschaften des Objektes kann jedoch mittels konstanter, öffentlicher Zugriffsmethoden (Member-Funktionen) erfolgen. Durch diese Kapselung findet eine Art 'Information Hiding' statt ('Geheimnisprinzip' der Klasse).

Konstruktoren und Destruktoren

Möchte man ein Objekt der Klasse im Hauptprogramm erzeugen, so deklariert man es mit dem Klassennamen, gibt dem Objekt einen eigenen Namen und initialisiert am besten gleichzeitig die Instanzvariablen des Objektes. In einer Klasse gibt es dafür eine besondere Member-Funktion, der sogenannte Konstruktor. In einer Klasse kann es mehrere überladene Konstruktoren geben, die z.B. unterschiedliche Initialisierungsvarianten beschreiben. Ein Konstruktor ist eine öffentlich zugängliche Member-Funktion der Klasse, die im Gegensatz zu den anderen Klassen-Funktionen keinen Rückgabetyp besitzt und der Funktionsname des Konstruktors ist identisch mit dem Namen der Klasse. Verlässt das Objekt den Gültigkeitsbereich seiner Deklaration, bzw. spätestens bei Beendigung der main()-Funktion, wird das Objekt mittels des Destruktors zerstört. Manche Klassen benötigen die explizite Angabe eines Destruktors, um z.B. reservierte und benötigte Speicherbereiche freizugeben. Der Name des Destruktors besteht aus einer 'Tilde' (~) gefolgt von seinem Klassennamen.

Arten von Klassen: Konkrete und abstrakte Klassen

Allgemein ist eine Klasse ein benutzerdefinierter Typ mit der Aufgabe, ein Konzept im Code eines Programms darzustellen. Ein auf objektorientierten Prinzipien aufgebauter Quelltext, in welchem die wesentlichen Kernkonzepte und Ideen des Programms durch einen gut ausgewählten Satz von Klassen formuliert sind, ist wesentlich einfacher zu verstehen. Man unterscheidet hierbei grob die folgenden Arten von Klassen: die konkreten und abstrakte Klassen. *Konkrete Klassen* sind Klassen, die eine spezielle, konkrete Idee des Programms bzw. eine getrennt beschreibbare Teil-Entität des Quelltextes in einer Klasse zusammenfassend definiert. Bei konkrete Klassen ist die Darstellung der Kernidee in Form von C++ Anweisungen Teil der Klasse selbst und die Objekte der Klasse können sofort und vollständig von der Klasse initialisiert werden. *Abstrakte Klassen* hingegen stellen eine Art von Schnittstelle dar und sie entkoppeln die eigentliche Darstellung der konkreten Umsetzung der Idee von der Klasse. Oft werden abstrakte Klassen benutzt, um eine generelle Überstruktur des Programms zu definieren und mehrere Klassen miteinander in Verbindung zu bringen. Der Begriff der *virtuellen Funktion* hat bei der Formulierung von abstrakten Klassen einen wichtigen Stellenwert und solche Funktionen werden mit dem Zusatz 'virtual' gekennzeichnet; was besagt, dass sie in einer von dieser abstrakten Klasse abgeleiteten Unterklasse redefiniert werden.

Klassenhierarchien und die Vererbung von Klassenmerkmalen

C++ Klassen stehen häufig in Beziehung zueinander. Man hat beispielsweise eine Oberklasse (z.B. Kuchen), und aus dieser leitet sich eine andere Klasse (z.B. Brombeerkuchen) ab. Diese abgeleitete Klasse erbt dann bestimmte Eigenschaften der Daten-Member und Member-Funktionen der Oberklasse. In umfangreichen C++ Programmen baut sich somit eine Art von Klassenhierarchie auf und oft werden dabei auch sogenannte *Templates* eingesetzt, auf die wir erst in einer späteren Vorlesung eingehen werden.

Ein einfaches Beispiel für eine konkrete Klasse

Wir möchten nun eine einfache Klasse von Objekten/Dingen erstellen, wobei jedes Objekt eine ganzzahlige positive Nummer n und eine Positionsangabe im Raum (eindimensionaler Raum mit Koordinate x) erhält. Diese Merkmale stellen die Instanzvariablen (Daten-Member) der Klasse dar und wir werden diese als private Daten deklarieren. Wir wählen als Klassenname 'Ding' und die Erzeugung der Objekte erfolgt mittels eines der drei überladenen Konstruktoren der Klasse:

```
Ding(unsigned int set_n, double set_x) : n{set_n}, x{set_x} { ...}
Ding(unsigned int set_n) : n{set_n}, x{0} { ... }
Ding() : n{0}, x{0} { ... }
```

Die einzelnen Konstruktoren folgen der Schreibweise
'Name der Klasse' ('Argumentenliste') : 'Initialisierung der Instanzvariablen mittels der Argumentenliste' { 'Anweisungsblock' }

und unterscheiden sich lediglich in der 'Argumentenliste'. Die Auswahl, welcher der Konstruktoren bei der Erzeugung des Objektes benutzt wird, ist dem Benutzer überlassen.

```cpp
/* Beispiel einer einfachen Klasse
 * Zwei private Instanzvariablen,
 * drei ueberladene Konstruktoren
 * Zwei oeffentliche Member-Funktionen
*/
#include <iostream>                    // Ein- und Ausgabebibliothek

//Definition der Klasse 'Ding'
class Ding{
    // Private Instanzvariablen (Daten-Member) der Klasse
    unsigned int n;
    double x;

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Drei ueberladene Konstruktoren der Klasse
        // Konstruktor mit zwei Argumenten
        Ding(unsigned int set_n, double set_x) : n{set_n}, x{set_x} {
            printf("Konstruktor(n,x) erzeugt ein neues Ding \n");
        }
        // Konstruktor mit einem Argument
        Ding(unsigned int set_n) : n{set_n}, x{0} {
            printf("Konstruktor(n) erzeugt ein neues Ding \n");
        }
        // Konstruktor ohne Argument (Standard-Konstruktor)
        Ding() : n{0}, x{0} {
            printf("Konstruktor() erzeugt ein neues Ding \n");
        }

        // Member-Funktionen der Klasse
        // als const deklariert, da sie die privaten Instanzvariablen nicht veraendern
        unsigned int get_Nummer() const {return n;}
        double get_Ort() const {return x;}

        // Destruktor der Klasse
        ~Ding(){
            printf("Destruktor, zerstört ein Ding \n");
        }
};
```

Die Klasse „Ding“

Das folgende Programm zeigt die Implementierung der Klasse im Programm:

Obwohl die Instanzvariablen n und x private Größen repräsentieren, kann man mittels öffentlicher Member-Funktionen auch auf die Werte dieser Member-Daten zugreifen. Solche Klasseninterne Funktionen sollten stets mit dem Zusatz const vor dem Anweisungsblock gekennzeichnet sein (hier z.B.
double get_Ort() const {**return** x;}
).

Anwendung der Klasse „Ding“ im Hauptprogramm

Im Hauptprogramm werden die Konstruktoren dann benutzt, um vier verschiedene Dinge mit den Namen 'Teilchen_A', 'Teilchen_B', 'Teilchen_C' und 'Teilchen_D' zu erstellen. Es werden hierbei unterschiedliche Formulierungen des Konstruktoraufrufs benutzt (z.B. Ding Teilchen_C = Ding(2,9.8); vs. Ding Teilchen_D {3,5.1};), wobei die letztere Variante wohl die zu Präferierende darstellt, da sie den Initialisierungscharakter des Konstruktor Aufrufs am besten gerecht wird.

```
int main(){                                 // Hauptfunktion
    Ding Teilchen_A = Ding();               // Benutzt Konstruktor Ding(), n=0, x=0
    Ding Teilchen_B = Ding(1);              // Benutzt Konstruktor Ding(n), n=1, x=0
    Ding Teilchen_C = Ding(2,9.8);          // Benutzt Konstruktor Ding(n,x), n=2, x=9.8
    Ding Teilchen_D {3,5.1};                // Benutzt Konstruktor Ding(n,x), n=3, x=5.1, andere Schreibweise der Initialisierung

    printf("\n");
    printf("Das Teilchen A hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_A.get_Nummer(), Teilchen_A.get_Ort());
    printf("Das Teilchen B hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_B.get_Nummer(), Teilchen_B.get_Ort());
    printf("Das Teilchen C hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_C.get_Nummer(), Teilchen_C.get_Ort());
    printf("Das Teilchen D hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_D.get_Nummer(), Teilchen_D.get_Ort());
    printf("\n");
}
```

Am Ende des Hauptprogramms werden diese Member-Funktionen dann mittels des Punktoperators aufgerufen, um die Nummer und den Ort des Objektes im Terminal auszugeben (siehe nebenstehende Abbildung). Bei der Beendigung des Programms werden die Destruktoren für alle vier erzeugten Teilchen aufgerufen.

```cpp
int main(){                                 // Hauptfunktion
    Ding Teilchen_A = Ding();               // Benutzt Konstruktor Ding(), n=0, x=0
    Ding Teilchen_B = Ding(1);              // Benutzt Konstruktor Ding(n), n=1, x=0
    Ding Teilchen_C = Ding(2,9.8);          // Benutzt Konstruktor Ding(n,x), n=2, x=9.8
    Ding Teilchen_D {3,5.1};                // Benutzt Konstruktor Ding(n,x), n=3, x=5.1, andere Schreibweise der Initialisierung

    printf("\n");
    printf("Das Teilchen A hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_A.get_Nummer(), Teilchen_A.get_Ort());
    printf("Das Teilchen B hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_B.get_Nummer(), Teilchen_B.get_Ort());
    printf("Das Teilchen C hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_C.get_Nummer(), Teilchen_C.get_Ort());
    printf("Das Teilchen D hat die Nummer %i und befindet sich an der Stelle %5.2f \n", Teilchen_D.get_Nummer(), Teilchen_D.get_Ort());
    printf("\n");
}
```

```cpp
//Definition der Klasse 'Ding'
class Ding{
    // Private Instanzvariablen (Daten-Member) der Klasse
    unsigned int n;
    double x;

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Drei ueberladene Konstruktoren der Klasse
        // Konstruktor mit zwei Argumenten
        Ding(unsigned int set_n, double set_x) : n{set_n}, x{set_x} {
            printf("Konstruktor(n,x) erzeugt ein neues Ding \n");
        }
        // Konstruktor mit einem Argument
        Ding(unsigned int set_n) : n{set_n}, x{0} {
            printf("Konstruktor(n) erzeugt ein neues Ding \n");
        }
        // Konstruktor ohne Argument (Standard-Konstruktor)
        Ding() : n{0}, x{0} {
            printf("Konstruktor() erzeugt ein neues Ding \n");
        }

        // Member-Funktionen der Klasse
        // als const deklariert, da sie die privaten Instanzvariablen
        unsigned int get_Nummer() const {return n;}
        double get_Ort() const {return x;}

        // Destruktor der Klasse
        ~Ding(){
            printf("Destruktor, zerstört ein Ding \n");
        }
};
```

```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Klassen/Ding$ g++
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Klassen/Ding$ ./a
Konstruktor() erzeugt ein neues Ding
Konstruktor(n) erzeugt ein neues Ding
Konstruktor(n,x) erzeugt ein neues Ding
Konstruktor(n,x) erzeugt ein neues Ding

Das Teilchen A hat die Nummer 0 und befindet sich an der Stelle  0.00
Das Teilchen B hat die Nummer 1 und befindet sich an der Stelle  0.00
Das Teilchen C hat die Nummer 2 und befindet sich an der Stelle  9.80
Das Teilchen D hat die Nummer 3 und befindet sich an der Stelle  5.10

Destruktor, zerstört ein Ding
Destruktor, zerstört ein Ding
Destruktor, zerstört ein Ding
Destruktor, zerstört ein Ding
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Klassen/Ding$
```

Die Klasse der Lagrange Polynom Methode

Die Frage, ob es sinnvoll ist ein Programm in eine Klassenstruktur zu bringen ist nicht leicht zu beantworten. Es ist hierbei wichtig, das gesamte Programm auf wiederverwertbare Algorithmen und kapselbare Konzepte zu untersuchen und diese zusammenhängenden Code-Fragmente in einer Klasse sinnvoll und möglichst benutzerfreundlich zu implementieren. Wir hatten in der Vorlesung 5, im Unterpunkt Anwendungsbeispiel: Interpolation und Polynomapproximation die Methode der Lagrange Polynome kennengelernt und diese in einem C++ Programm implementiert (siehe auch Übungsblatt Nr.5, Aufgabe 1). Das Konzept der Lagrange Polynome stellt ein kapselbares, inhaltlich zusammenhängendes Konzept im Programm dar und wir wollen deshalb eine Klasse konstruieren, die der Benutzer anwenden kann, um den Polynomwert an der Stelle x bei einem vorgegebenen Stützstellenarray zu erhalten. Der unten abgebildete C++ Quelltext stellt eine Implementierung einer solchen Klasse dar.

Lagrange_Polynom_Klasse.cpp

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom (mit ausgelagerter 'LagrangePoly'-Klasse)
 * Mittels der Methode der Lagrange Polynome entwickelt man eine Funktion ( hier speziell f(x)=1/x )
 * durch Angabe von N+1 vorgegebener Punkte in ein Lagrange Polynom vom Grade N.
 * Hier speziell 7 Punkte
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_Klasse.dat" */
#include <iostream>                                                 // Ein- und Ausgabebibliothek

//Definition der Klasse 'LagrangePoly'
class LagrangePoly {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double* points;                                                 // Zeigervariable der Stuetzstellenpunkte
    unsigned int N_points;                                          // Anzahl der Stuetzstellenpunkte

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Konstruktor mit zwei Argumenten
        LagrangePoly(double* set_points, unsigned int set_N_points) : points{set_points}, N_points{set_N_points} {}

        double rechne(double x) {                                   // Member-Funktionen der Klasse zur Berechnung des approximierten Polynomwertes
            double Pfp = 0;                                         // Deklaration und Initialisierung des Funktionswertes des approximierten Polynoms
            double Lk = 0;                                          // Deklaration und Initialisierungeiner Zusatzvariable
            for(int k = 0; k < N_points; ++k){                      // For-Schleife der Summation in der Lagrange Polynom Methode
                Lk=1;                                               // Initialisierung der Produktvariable Lk mit 1
                for(int i = 0; i < N_points; ++i){                  // For-Schleife der Produktbildung in der Lagrange Polynom Methode
                    if(i != k){                                     // Die Produktbildung soll nur fuer (i ungleich k) erfolgen
                        Lk = Lk * (x - points[i])/(points[k] - points[i]); // Berechnung der Lk-Werte in der Lagrange Polynom Methode
                    }                                               // Ende if-Bedingung
                }                                                   // Ende for-Schleife der Produktbildung
                Pfp = Pfp + f(points[k])*Lk;                        // Kern-Gleichung in der Lagrange Polynom Methode
            }                                                       // Ende for-Schleife der Summenbildung
            return Pfp;                                             // Rueckgabe des berechneten, approximierten Polynomwertes
        }                                                           // Ende der Member-Funktion rechne(double x)
```

Die Klasse „Lagrangepoly“

Lagrange_Polynom_Klasse.cpp

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom (mit ausgelagerter 'LagrangePoly'-Klasse)
 * Mittels der Methode der Lagrange Polynome entwickelt man eine Funktion ( hier speziell f(x)=1/x )
 * durch Angabe von N+1 vorgegebener Punkte in ein Lagrange Polynom vom Grade N.
 * Hier speziell 7 Punkte
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_Klasse.dat" */
#include <iostream>                                                    // Ein- und Ausgabebibliothek

//Definition der Klasse 'LagrangePoly'
class LagrangePoly {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double* points;                                                    // Zeigervariable der Stuetzstellenpunkte
    unsigned int N_points;                                             // Anzahl der Stuetzstellenpunkte

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Konstruktor mit zwei Argumenten
        LagrangePoly(double* set_points, unsigned int set_N_points) : points{set_points}, N_points{set_N_points} {}

        double rechne(double x) {                                      // Member-Funktionen der Klasse zur Berechnung des approximierten Polynomwertes
            double Pfp = 0;                                            // Deklaration und Initialisierung des Funktionswertes des approximierten Polynoms
            double Lk = 0;                                             // Deklaration und Initialisierungeiner Zusatzvariable
            for(int k = 0; k < N_points; ++k){                         // For-Schleife der Summation in der Lagrange Polynom Methode
                Lk=1;                                                  // Initialisierung der Produktvariable Lk mit 1
                for(int i = 0; i < N_points; ++i){                     // For-Schleife der Produktbildung in der Lagrange Polynom Methode
                    if(i != k){                                        // Die Produktbildung soll nur fuer (i ungleich k) erfolgen
                        Lk = Lk * (x - points[i])/(points[k] - points[i]); // Berechnung der Lk-Werte in der Lagrange Polynom Methode
                    }                                                  // Ende if-Bedingung
                }                                                      // Ende for-Schleife der Produktbildung
                Pfp = Pfp + f(points[k])*Lk;                           // Kern-Gleichung in der Lagrange Polynom Methode
            }                                                          // Ende for-Schleife der Summenbildung
            return Pfp;                                                // Rueckgabe des berechneten, approximierten Polynomwertes
        }                                                              // Ende der Member-Funktion rechne(double x)

        double f(double x){                                            // Deklaration und Definition der Funktion f(x) die approximiert werden soll
            double wert;
            wert = 1.0/x;                                              // Eigentliche Definition der Funktion
            return wert;                                               // Rueckgabewert der Funktion f(x)
        }                                                              // Ende der Funktion f(x)
};                                                                     // Ende der Klassendefinition
```

Anwendung der Klasse „Lagrangepoly“ im Hauptprogramm

```
int main(){                                                                    // Hauptfunktion
    double points[] = { 1, 1.5, 2, 2.5, 3, 5, 7 };                             // Deklaration und Initialisierung der Punkte als double-Datenfeld (Array)
    unsigned int N_points = sizeof(points)/sizeof(points[0]);                  // Anzahl der Punkte die zur Approximation verwendet werden
    double plot_a = 0.5;                                                       // Untergrenze des x-Intervalls in dem die Ergebnisse ausgegeben werden sollen
    double plot_b = 6;                                                         // Obergrenze des x-Intervalls in dem die Ergebnisse ausgegeben werden sollen
    const unsigned int N_xp=300;                                               // Anzahl der Punkte in die das x-Intervall aufgeteilt wird
    double dx = (plot_b - plot_a)/N_xp;                                        // Abstand dx zwischen den aequidistanten Punkten des x-Intervalls
    double x = plot_a-dx;                                                      // Aktueller x-Wert
    double xp[N_xp+1];                                                         // Deklaration der x-Ausgabe-Punkte als double-Array
    double fp[N_xp+1];                                                         // Deklaration der f(x)-Ausgabe-Punkte als double-Array
    double Pfp[N_xp+1];                                                        // Deklaration der Ausgabe-Punkte des approximierten Polynoms als double-Array

    LagrangePoly Poly1 {points,N_points};                                      // Aufruf des Konstruktors der Klasse LagrangePoly (Erzeugung des Objektes 'Poly1')

    printf("# x-Werte der %3d Stuetzstellen-Punkte: \n", N_points);            // Beschreibung der ausgegebenen Groessen
    for(int k = 0; k < N_points; ++k){                                         // For-Schleife der Ausgabe der Stuetzstellen x-Werte
        printf("%10.5f",points[k]);                                            // Ausgabe der Stuetzpunkte
    }                                                                          // Ende for-Schleife der Ausgabe
    printf("\n");                                                              // Zeilenumbruch

    printf("# 0: Index j \n# 1: x-Wert \n# 2: f(x)-Wert \n");                  // Beschreibung der ausgegebenen Groessen
    printf("# 3: Approximierter Wert des Lagrange Polynoms P(x) \n");          // Beschreibung der ausgegebenen Groessen
    printf("# 4: Fehler zum wirklichen Wert f(x)-P(x) \n");                    // Beschreibung der ausgegebenen Groessen

    for(int j = 0; j <= N_xp; ++j){                                            // For-Schleife die ueber die einzelnen Punkte des x-Intervalls geht
        x = x + dx;                                                            // Aktueller x-Wert
        xp[j] = x;                                                             // Eintrag des aktuellen x-Wertes in das x-Array
        fp[j] = Poly1.f(x);                                                    // Eintrag des aktuellen f(x)-Wertes in das fp-Array (Aufruf der Member-Funktion f(x))
        Pfp[j] = Poly1.rechne(x);                                              // Eintrag des aktuellen approximierten Polynom-Wertes in das Pfp-Array (Aufruf der Member-Funktion rechne(x))
    }                                                                          // Ende der for-Schleife ueber die einzelnen Punkte des x-Intervalls

    for(int j = 0; j <= N_xp; ++j){                                                                  // For-Schleife der separaten Ausgabe der berechneten Werte
        printf("%3d %14.10f %14.10f %14.10f %14.10f \n",j, xp[j], fp[j], Pfp[j], (fp[j] - Pfp[j])); // Ausgabe der berechneten Werte
    }                                                                                                // Ende for-Schleife der Ausgabe
}                                                                                                    // Ende der Hauptfunktion
```

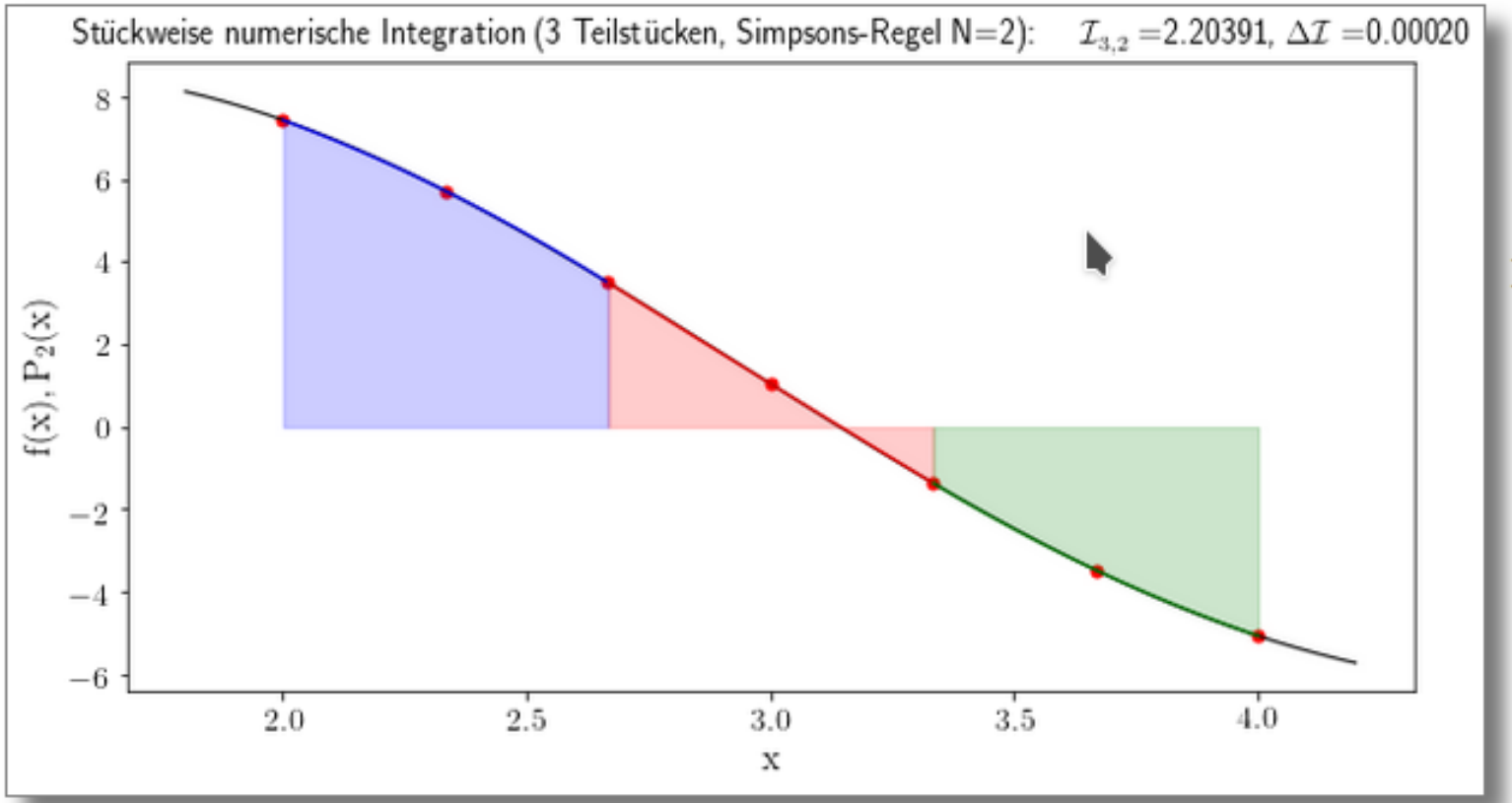
Lagrange_Polynom_Klasse_a.cpp

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom (mit ausgelagerter 'LagrangePoly'-Klasse und inline-Member Funktion f(x))
 * Mittels der Methode der Lagrange Polynome entwickelt man eine Funktion ( hier speziell f(x)=1/x )
 * durch Angabe von N+1 vorgegebener Punkte in ein Lagrange Polynom vom Grade N.
 * Hier speziell 7 Punkte
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_Klassea.dat" */
#include <iostream>                                          // Ein- und Ausgabebibliothek

//Definition der Klasse 'LagrangePoly'
class LagrangePoly {
    // Private Instanzvariablen (Daten-Member) der Klasse
    double* points;                                          // Zeigervariable der Stuetzstellenpunkte
    unsigned int N_points;                                   // Anzahl der Stuetzstellenpunkte

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Konstruktor mit zwei Argumenten
        LagrangePoly(double* set_points, unsigned int set_N_points) : points{set_points}, N_points{set_N_points} {}

        double f(double x);                                  // Deklaration der Member-Funktion f(x) (Definition findet ausserhalb der Klasse statt)

        double rechne(double x) {                            // Member-Funktion der Klasse zur Berechnung des approximierten Polynomwertes
            double Pfp = 0;                                  // Deklaration und Initialisierung des Funktionswertes des approximierten Polynoms
            double Lk = 0;                                   // Deklaration und Initialisierungeiner Zusatzvariable
            for(int k = 0; k < N_points; ++k){               // For-Schleife der Summation in der Lagrange Polynom Methode
                Lk=1;                                        // Initialisierung der Produktvariable Lk mit 1
                for(int i = 0; i < N_points; ++i){           // For-Schleife der Produktbildung in der Lagrange Polynom Methode
                    if(i != k){                              //                                                  erfolgen
                        Lk = Lk * (x - points[i])/(points[k] - points[i]); //                                    Methode
                    }                                        //
                }                                            //
                Pfp = Pfp + f(points[k])*Lk;                 //
            }                                                //
            return Pfp;                                      //                                                  mwertes
        }                                                    // Ende der Member-Funktion rechne(double x)
};                                                           // Ende der Klassendefinition

inline double LagrangePoly::f(double x){                     //  Definition der Funktion f(x) als inline-Methode der Klasse LagrangePoly
    double wert;
    wert = 1.0/x;                                            // Eigentliche Definition der Funktion
    return wert;                                             // Rueckgabewert der Funktion f(x)
}                                                            // Ende der Funktion f(x)
```

Die Klasse „Lagrangepoly“

Ausgelagerte „inline“-Funktion Der Klasse

Theorie: Numerische Integration



Wir betrachten in diesem Unterpunkt die Methode der numerischen Integration mittels der "Geschlossenen Newton-Cotes Gleichungen" (closed Newton-Cotes formulars). Die Vorgehensweise der Herleitung dieser Gleichungen erfolgt, indem man die zu integrierende Funktion $f(x)$ in ein Lagrange Polynom vom Grade N ($P_N(x)$) entwickelt ((siehe Anwendungsbeispiel: Interpolation und Polynomapproximation)) und dann durch analytische Integration zur Approximation gelangt. Beim Klicken auf die nebenstehende Abbildung gelangen Sie zu einem Jupyter Notebook, das die Herleitung der einzelnen Integrationsregeln veranschaulicht und speziell die Trapez-Regel ($N = 1$), die Simpson's-Regel ($N = 2$), die Simpson's-3/8-Regel ($N = 3$) und die $N = 4$-Regel behandelt. Möchte man eine Funktion jedoch über ein großes Intervall integrieren, so liefern auch die höheren N-Regeln keine genaue Approximation. Man sollte dann eine iterative, stückweise numerische Integration verwenden, wobei man innerhalb der einzelnen Teilstücke eine der oberen Integrationsregeln verwendet. Die nebenstehende Abbildung veranschaulicht diese stückweise numerische Integration, wobei das Integrationsintervall $[2, 4]$ in drei Teilintervalle unterteilt wurde und in den jeweiligen Teilintervallen die Simpson's-Regel verwendet wurde (näheres siehe Theorie: Numerische Integration).

Theorie: Numerische Integration

Wir betrachten in diesem Unterpunkt die Methode der numerischen Integration mittels der "Geschlossenen Newton-Cotes Gleichungen" (closed Newton-Cotes formulars). Die Vorgehensweise der Herleitung dieser Gleichungen erfolgt, indem man die zu integrierende Funktion $f(x)$ in ein Lagrange Polynom vom Grade N ($P_n(x)$) entwickelt (siehe Anwendungsbeispiel: Interpolation und Polynomapproximation) und dann durch analytische Integration zur Approximation gelangt, die für $N = 1$ in die Trapez-Regel, für $N = 2$ in die Simpson's-Regel und für $N = 3$ in die Simpson's-3/8-Regel übergehen.
Wir möchten das folgende bestimmte Integral numerisch approximieren:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{N} c_i\, f(x_i) \quad ,$$

wobei man das Integrationsintervall in N äquidistante Punkte unterteilt: $x_0 = a$, $x_N = b$ und $x_i = a + i \cdot (b-a)/N = a + i \cdot h$ mit $(i = 0, 1, 2, .., N)$. Im Folgenden wollen wir die jeweiligen Gleichungen mittels der Methode der Lagrange Polynome ($P_N(x)$) herleiten. Das N-te Lagrange Polynom $P_N(x)$ einer Funktion ist wie folgt definiert:

$$P_N(x) = \sum_{k=0}^{N} f(x_k) \cdot L_{N,k}(x) \quad ,$$

mit

$$L_{N,k}(x) = \prod_{i=0,\, i\neq k}^{N} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

Setzt man das Lagrange Polynom $P_N(x)$ in das bestimmte Integral ein, erhält man:

$$\int_a^b f(x)\,dx \approx \int_a^b \sum_{k=0}^{N} f(x_k) \cdot L_{N,k}(x)\,dx = \sum_{k=0}^{N} f(x_k) \cdot \underbrace{\int_a^b L_{N,k}(x)\,dx}_{c_k} := \sum_{k=0}^{N} f(x_k)\, c_k \quad .$$

Anwendungsbeispiel: Interpolation und Polynomapproximation) und dann durch analytische Integration zur App
$N = 1$ in die Trapez-Regel, für $N = 2$ in die Simpson's-Regel und für $N = 3$ in die Simpson's-3/8-Regel übergehe
Wir möchten das folgende bestimmte Integral numerisch approximieren:

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{N} c_i\, f(x_i) \quad ,$$

ationsintervall in N äquidistante Punkte unterteilt: $x_0 = a$, $x_N = b$ und $x_i = a + i \cdot (b-a)/N = a + i \cdot h$ mit $(i = 0,$
en Gleichungen mittels der Methode der Lagrange Polynome ($P_N(x)$) herleiten. Das N-te Lagrange Polynom $P_N($
folgt definiert:

$$P_N(x) = \sum_{k=0}^{N} f(x_k) \cdot L_{N,k}(x) \quad ,$$

mit

$$L_{N,k}(x) = \prod_{i=0,\, i\neq k}^{N} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

Setzt man das Lagrange Polynom $P_N(x)$ in das bestimmte Integral ein, erhält man:

$$\int_a^b f(x)\,dx \approx \int_a^b \sum_{k=0}^{N} f(x_k) \cdot L_{N,k}(x)\,dx = \sum_{k=0}^{N} f(x_k) \cdot \underbrace{\int_a^b L_{N,k}(x)\,dx}_{c_k} := \sum_{k=0}^{N} f(x_k)\, c_k \quad .$$

Die Trapez-Regel (N=1)

Die einfachste Approximation erhält man mittels eines linearen Lagrange Polynoms $P_1(x)$. Man benutzt dabei lediglich zwei Punkte, nämlich die Grenzen des Intervalls ($x_0 = a$ und $x_1 = b = x_0 + h$ mit $h = (b - a)$). Mittels $P_1(x)$ berechnen wir das folgende bestimmte Integral

$$\int_a^b f(x)\,dx \approx \mathcal{I}_1 := \int_{x_0}^{x_1} \sum_{k=0}^{1} f(x_k) \cdot L_{1,k}(x)\,dx = \int_{x_0}^{x_1} \sum_{k=0}^{1} f(x_k) \cdot \prod_{i=0,\, i\neq k}^{1} \frac{(x - x_i)}{(x_k - x_i)}\,dx \quad ,$$

und durch analytische Integration erhält man die folgende Trapez-Regel:

$$\mathcal{I}_1 = \frac{(x_1 - x_0)}{2} \cdot (f(x_0) + f(x_1)) = \frac{h}{2} \cdot (f(x_0) + f(x_1)) \quad .$$

Die Simpson's-Regel (N=2)

Die Simpson's Approximation erhält man mittels eines quadratischen Lagrange Polynoms $P_2(x)$. Man benutzt dabei drei Punkte, nämlich die Grenzen des Intervalls ($x_0 = a$ und $x_2 = b = a + 2h$) und einen Wert in der Mitte des Intervalls ($x_1 = a + h$) mit $h = (b - a)/2$. Wieder erhält man durch analytische Integration die folgende Simpson's-Regel:

$$\mathcal{I}_2 = \frac{h}{3} \cdot (f(x_0) + 4f(x_1) + f(x_2))$$

Die Simpson's-3/8-Regel (N=3)

Die Simpson's-3/8 Approximation erhält man mittels des Lagrange Polynoms $P_3(x)$. Man benutzt dabei vier Punkte, nämlich die Grenzen des Intervalls ($x_0 = a$ und $x_3 = b = a + 3h$) und zwei zusätzliche Werte ($x_1 = a + h$ und $x_2 = a + 2h$) mit $h = (b - a)/3$. Man erhält:

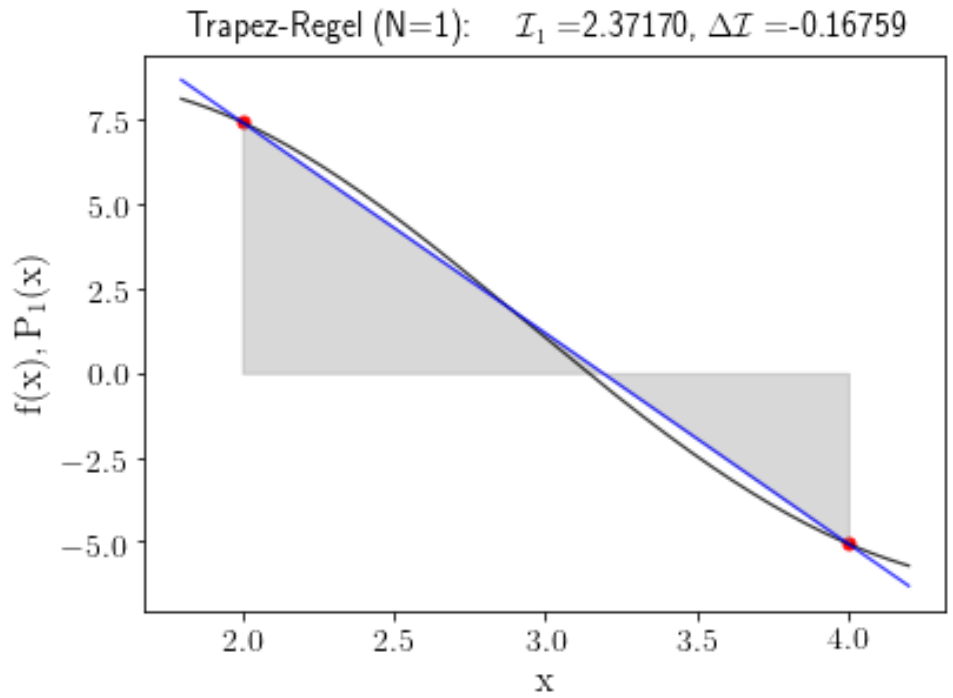
$$\mathcal{I}_3 = \frac{3\,h}{8} \cdot (f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$$

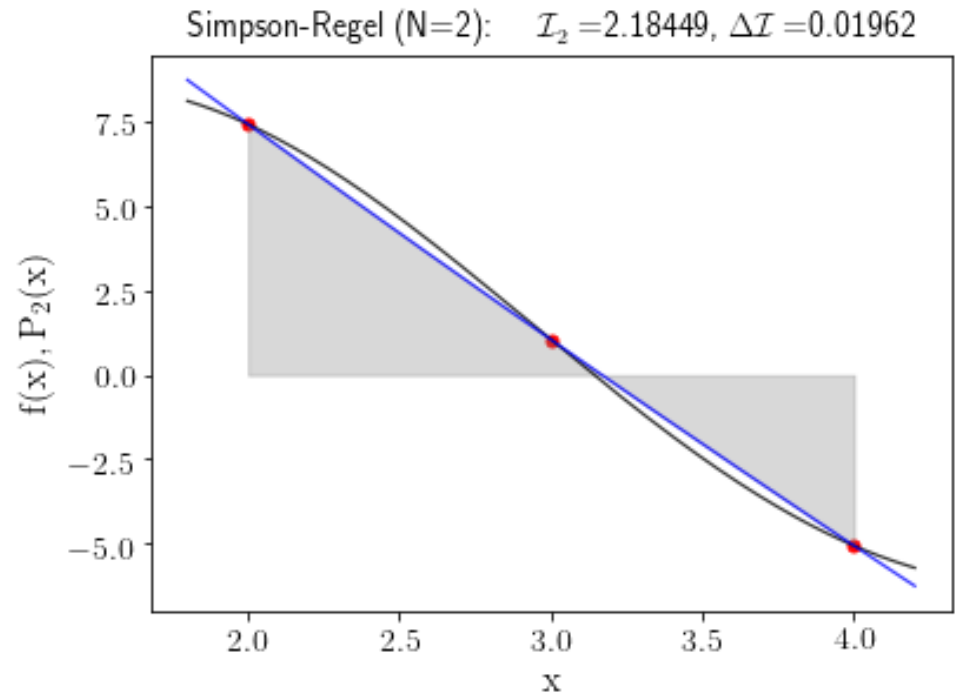
Die N=4 Regel

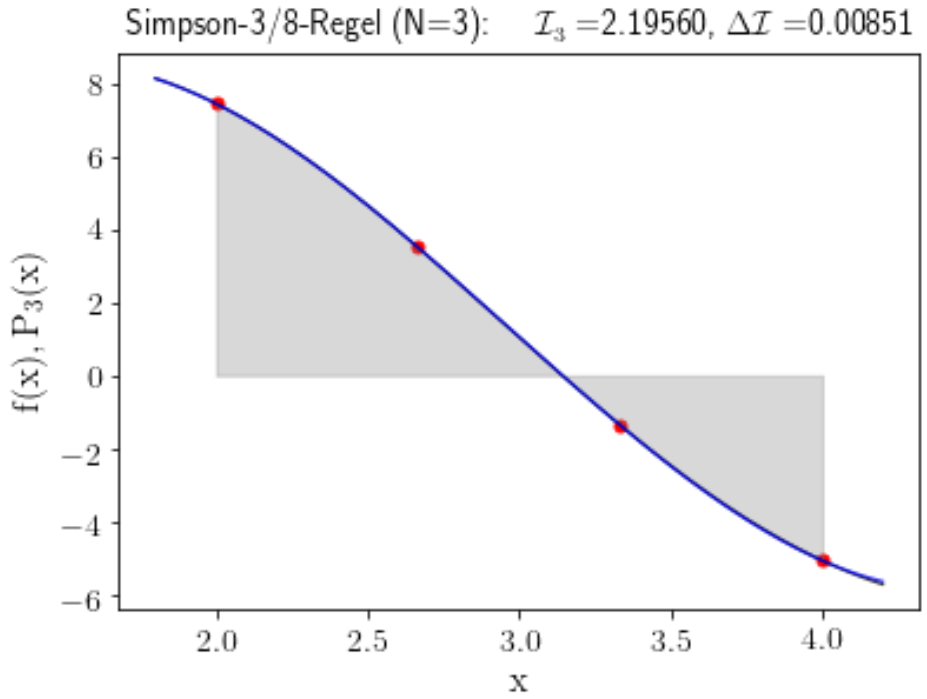
Die $N = 4$ Approximation erhält man mittels des Lagrange Polynoms $P_4(x)$. Man benutzt dabei fünf Punkte, nämlich die Grenzen des Intervalls ($x_0 = a$ und $x_4 = b = a + 4h$) und zwei zusätzliche Werte ($x_1 = a + h$, $x_2 = a + 2h$ und $x_3 = a + 3h$) mit $h = (b - a)/4$. Man erhält:

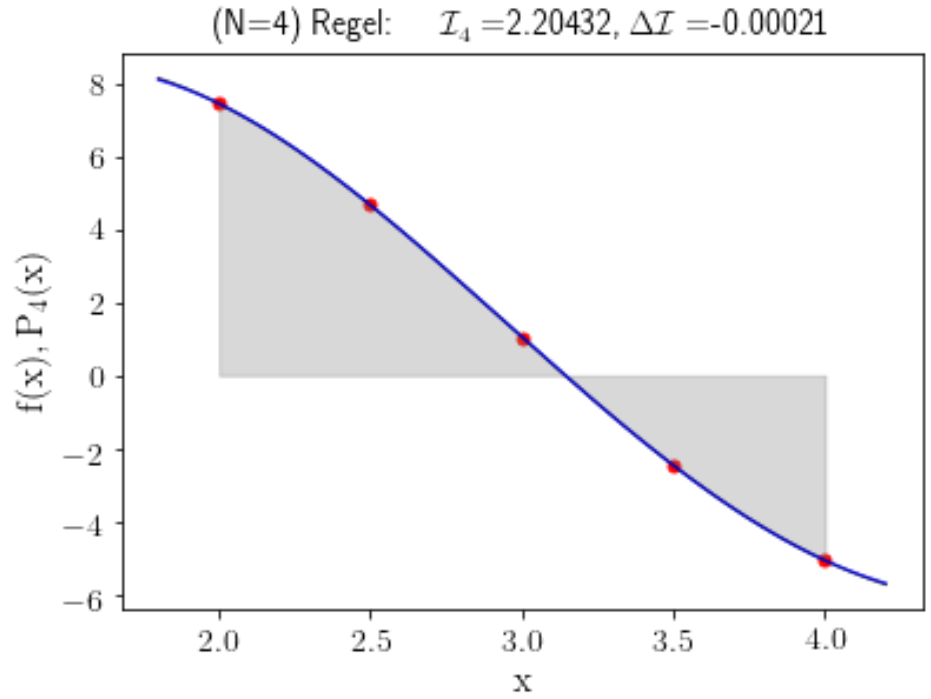
$$\mathcal{I}_4 = \frac{2\,h}{45} \cdot (7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4))$$

Wir betrachten nun z.B. die Funktion $f(x) = 10 \cdot \sin(x) \cdot e^{-\frac{x}{10}}$ und approximieren den Wert für das bestimmte Integral $\int_2^4 f(x)\,dx$. Die nebenstehende Abbildung veranschaulicht für dieses Beispiel die unterschiedlichen Integrationsregeln grafisch.


Trapez-Regel (N=1): $\mathcal{I}_1$ =2.37170, $\Delta\mathcal{I}$ =-0.16759
f(x), P$_1$(x)
x


Simpson-Regel (N=2): $\mathcal{I}_2$ =2.18449, $\Delta\mathcal{I}$ =0.01962
f(x), P$_2$(x)
x


Simpson-3/8-Regel (N=3): $\mathcal{I}_3$ =2.19560, $\Delta\mathcal{I}$ =0.00851
f(x), P$_3$(x)
x


(N=4) Regel: $\mathcal{I}_4$ =2.20432, $\Delta\mathcal{I}$ =-0.00021
f(x), P$_4$(x)
x

Zusammenfassung
Integrationsregeln der numerischen Mathematik

Anwendungsbeispiel: Numerische Integration

Die im vorigen Unterpunkt hergeleitete numerische Integrationsregeln werden nun in einem C++ Programm benutzt, um den Wert des bestimmten Integrals einer Funktion $\int_a^b f(x)\,dx$ approximativ zu bestimmen. Die Integrationsregeln lauteten:

Die Trapez-Regel (N=1): $$\int_a^b f(x)\,dx \approx \frac{(x_1 - x_0)}{2} \cdot (f(x_0) + f(x_1)) = \frac{h}{2} \cdot (f(x_0) + f(x_1))$$, Stützstellen: $(x_0 = a, x_0 + h = b)$

Die Simpson's-Regel (N=2): $$\int_a^b f(x)\,dx \approx \frac{h}{3} \cdot (f(x_0) + 4f(x_1) + f(x_2))$$, Stützstellen: $(x_0 = a, x_0 + h, x_0 + 2h = b)$

Die Simpson's-3/8-Regel (N=3): $$\int_a^b f(x)\,dx \approx \frac{3\,h}{8} \cdot (f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$$, Stützstellen: $(x_0 = a, x_0 + h, x_0 + 2h, x_0 + 3h = b)$

Die N=4 Regel: $$\int_a^b f(x)\,dx \approx \frac{2\,h}{45} \cdot (7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4))$$, Stützstellen: $(x_0 = a, x_0 + h, x_0 + 2h, x_0 + 3h, x_0 + 4h = b)$

Anwendung der Integrationsregeln in einem C++ Programm

Numerical_Integration_1.cpp

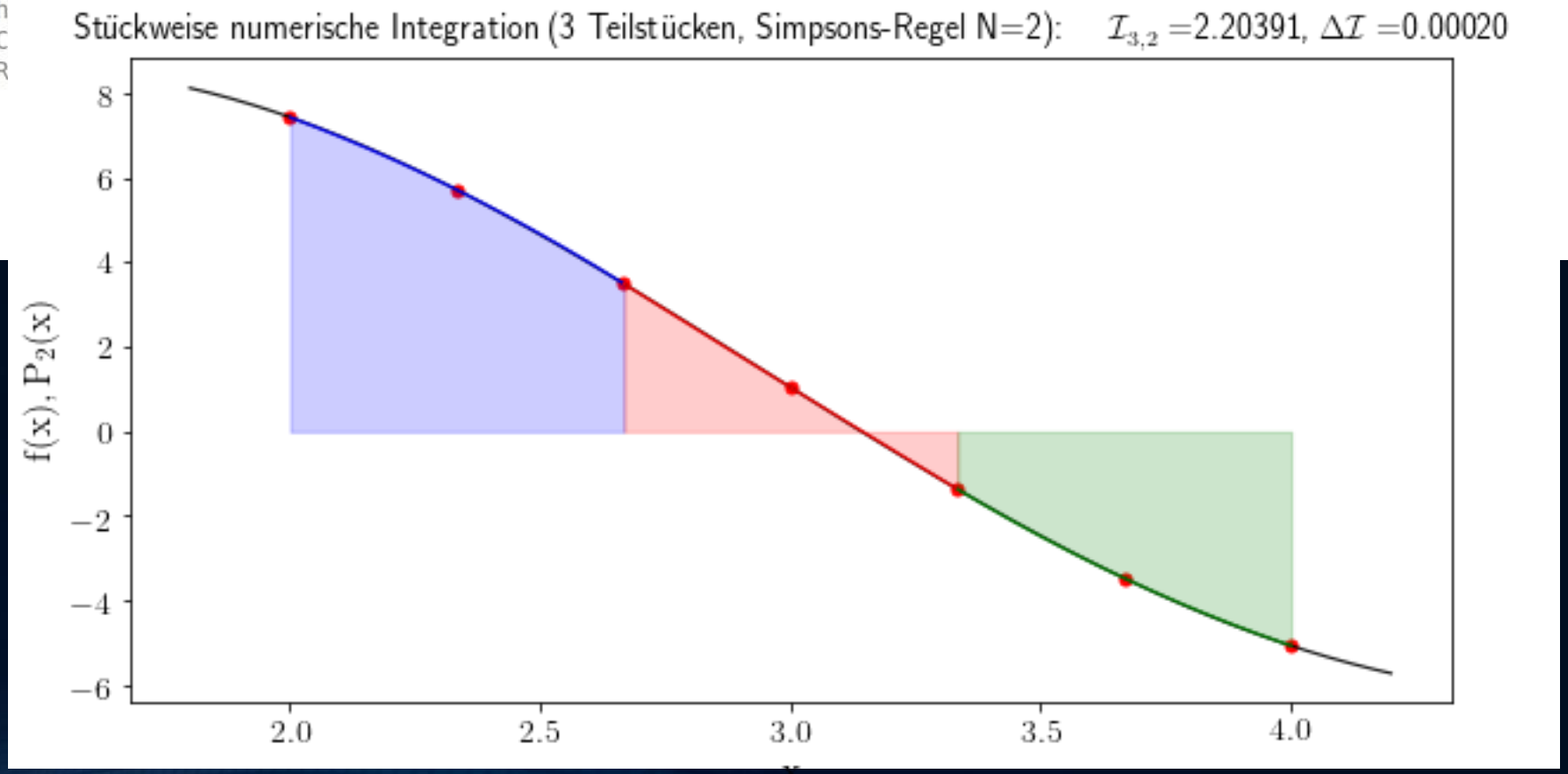
```cpp
/* Berechnung des bestimmten Integrals einer Funktion f(x) in den Grenzen [a,b]
 * mittels unterschiedlicher "closed Newton-Cotes formulars"
 * (N=1 Trapez-Regel, N=2 Simpson's-Regel, N=3 Simpson's-3/8-Regel und N=4 Regel)
 */
#include <iostream>                                          // Ein- und Ausgabebibliot
#include <cmath>                                             // Bibliothek für mathemat

double f(double x){                                          // Definition der Funktion f(x)
    double wert;
    wert = 10*sin(x)*exp(-x/10);                             // Eigentliche Definition der Funktion
    return wert;                                             // Rueckgabewert der Funktion f(x)
}                                                            // Ende der Funktion f(x)

double Int_f(double x){                                      // Definition der analytischen Stammfunktion von f(x)
    double wert;                                             // (Bem.: benoetigen wir nur zum Vergleich)
    wert = -100*sin(x)*exp(-x/10)/101 - 1000*cos(x)*exp(-x/10)/101;
    return wert;
}

int main(){                                                  // Hauptfunktion
    double a = 2;                                            // Untergrenze des bestimmten Integrals
    double b = 4;                                            // Obergrenze des bestimmten Integrals
    double h;                                                // Deklaration des h-Wertes
    double I_1, I_2, I_3, I_4;                               // Deklaration der Integrale der vier Approximationsregel

    h = (b-a)/1;
    I_1 = h/2 * (f(a) + f(b));                                       // N=1 Trapez-Regel
    h = (b-a)/2;
    I_2 = h/3 * (f(a) + 4*f(a+h) + f(b));                            // N=2 Simpson's-Regel
    h = (b-a)/3;
    I_3 = 3*h/8 * (f(a) + 3*f(a+h) + 3*f(a+2*h) + f(b));             // N=3 Simpson's-3/8-Regel
    h = (b-a)/4;
    I_4 = 2*h/45 * (7*f(a) + 32*f(a+h) + 12*f(a+2*h) + 32*f(a+3*h) + 7*f(b)); // N=4 Regel

    // Ausgaben der berechneten Groessen
    printf("Integral N=1: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_1, (Int_f(b)-Int_f(a)) - I_1);
    printf("Integral N=2: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_2, (Int_f(b)-Int_f(a)) - I_2);
    printf("Integral N=3: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_3, (Int_f(b)-Int_f(a)) - I_3);
    printf("Integral N=4: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_4, (Int_f(b)-Int_f(a)) - I_4);
}                                                                    // Ende der Hauptfunktion
```

Stückweise numerische Integration

Da man das betrachtete Integrationsintervall [a,b] ohne einen großen Aufwand in viele Teilintervalle unterglieder kann, ist die stückweise numerische Integration gerade dafür prädestiniert auf dem Computer berechnet zu werden. In dem folgenden C++ Programm wird wieder das bestimmte Integral der Funktion $f(x) = 10\,e^{-x/10} \cdot \sin(x)$ in den Grenzen [2,4] approximativ berechnet, jedoch wird das gesamte Intervall in 'ts' Teilintervalle unterteilt (hier speziell 'ts=3').

Numerical_Integration_2.cpp

```
/* Berechnung des bestimmten Integral
 * mittels einer stückweise numerisch
 * unterschiedlicher "closed Newton-C
 * (N=1 Trapez-Regel, N=2 Simpson's-R
 */
#include <iostream>
#include <cmath>

double f(double x){
    double wert;
```



Stückweise numerische Integration

Möchte man die Funktion jedoch über ein großes Intervall integrieren, so liefern auch die höheren N-Regeln keine genaue Approximation. Man sollte dann eine iterative, stückweise numerische Integration verwenden, wobei man innerhalb der einzelnen Teilstücke eine der oberen Integrationsregeln verwendet. Wir unterteilen dazu das Integrations-Intervall $[a, b]$ in n/N-Teilstücke, wobei N die Ordnung der Integrationsregel und n das kleinste gemeinsame Vielfache von N ist.

$$\int_a^b f(x)\,dx = \sum_{j=1}^{n/N} \int_{x_{N\cdot j-N}}^{x_{N\cdot j}} f(x)\,dx$$

$$= \int_{a=x_0}^{x_N} f(x)\,dx + \int_{x_N}^{x_{N\cdot 2}} f(x)\,dx + \ldots + \int_{x_{N\cdot(n/N-1)-N}}^{x_{N\cdot(n/N-1)}} f(x)\,dx + \int_{x_{N\cdot(n/N)-N}}^{x_{N\cdot(n/N)}} f(x)\,dx =$$

$$= \underbrace{\int_{a=x_0}^{x_N} f(x)\,dx}_{\text{Teilintervall } 1} + \underbrace{\int_{x_N}^{x_{N\cdot 2}} f(x)\,dx}_{\text{Teilintervall } 2} + \ldots + \underbrace{\int_{x_{n-2\cdot N}}^{x_{n-N}} f(x)\,dx}_{\text{Teilintervall } n/N-1} + \underbrace{\int_{x_{n-N}}^{x_n} f(x)\,dx}_{\text{Teilintervall } n/N},$$

und benutzen in den Teilintervallen eine der oberen Integrationsregeln, so geht man zu einer stückweisen numerischen Integration über.

Stückweise numerische Integration
Beispiel: 3 Teilintervalle und Simpsons-Regel

Unterteilen wir z.B. das Integrations-Intervall in drei Teilstücke und benutzen die Simpson's-Regel ($N = 2$, $n = 6 \rightarrow n/N = 3$ Teilstücke), so erhalten wir z.B. die folgenden Integrationsterme:

$$\int_a^b f(x)\,dx = \underbrace{\int_{a=x_0}^{x_2} f(x)\,dx}_{\text{Teilintervall 1}} + \underbrace{\int_{x_2}^{x_4} f(x)\,dx}_{\text{Teilintervall 2}} + \underbrace{\int_{x_4}^{x_6} f(x)\,dx}_{\text{Teilintervall 3}} =$$

$$= \int_{a=x_0}^{x_2} \sum_{k=0}^{2} f(x_k) \cdot L_{2,k}(x)\,dx + \int_{x_2}^{x_4} \sum_{k=2}^{4} f(x_k) \cdot L_{2,k}(x)\,dx + \int_{x_4}^{x_6} \sum_{k=4}^{6} f(x_k) \cdot L_{2,k}(x)\,dx =$$

$$= \underbrace{\frac{h}{3} \cdot (f(x_0) + 4f(x_1) + f(x_2))}_{\text{Teilintervall 1}} + \underbrace{\frac{h}{3} \cdot (f(x_2) + 4f(x_3) + f(x_4))}_{\text{Teilintervall 2}} + \underbrace{\frac{h}{3} \cdot (f(x_4) + 4f(x_5) + f(x_6))}_{\text{Teilintervall 3}} =$$

$$= \frac{h}{3} \cdot (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + 4f(x_5) + f(x_6)) \quad .$$

Stückweise numerische Intergration

Numerical_Integration_2.cpp

```cpp
/* Berechnung des bestimmten Integrals einer Funktion f(x) in den Grenzen [a,b]
 * mittels einer stückweise numerischen Integration und
 * unterschiedlicher "closed Newton-Cotes formulars"
 * (N=1 Trapez-Regel, N=2 Simpson's-Regel, N=3 Simpson's-3/8-Regel und N=4 Regel)
 */
#include <iostream>                                     // Ein- und Ausgabebibliothek
#include <cmath>                                        // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f(double x){                                     // Definition der Funktion f(x)
    double wert;
    wert = 10*sin(x)*exp(-x/10);                        // Eigentliche Definition der Funktion
    return wert;                                        // Rueckgabewert der Funktion f(x)
}                                                       // Ende der Funktion f(x)

double Int_f(double x){                                 // Definition der analytischen Stammfunktion von f(x)
    double wert;                                        // (Bem.: benoetigen wir nur zum Vergleich)
    wert = -100*sin(x)*exp(-x/10)/101 - 1000*cos(x)*exp(-x/10)/101;
    return wert;
}

int main(){                                                                     // Hauptfunktion
    unsigned ts = 3;                                                            // Anzahl der Teilintervalle
    double a = 2;                                                               // Untergrenze des bestimmten Integrals
    double b = 4;                                                               // Obergrenze des bestimmten Integrals
    double h;                                                                   // Deklaration des h-Wertes
    double x_u;                                                                 // x-Untergrenze der Teilintegration
    double I_1=0, I_2=0, I_3=0, I_4=0;                                          // Deklaration der Integrale der vier Approximationsregel

    for(int i = 0; i < ts; ++i){                                                    // Schleifen Anfang ueber die einzelnen Teilintegrationen
        h=(b-a)/(ts*1);                                                             // N=1 Trapez-Regel
        x_u = a + i*h;                                                              // Setzen der x-Untergrenze der aktuellen Teilintegration
        I_1 = I_1 + h/2*(f(x_u) + f(x_u+h));                                        // N=1 Trapez-Regel
        h=(b-a)/(ts*2);                                                             // N=2 Simpson's-Regel
        I_2 = I_2 + h/3*(f(x_u) + 4*f(x_u+h) + f(x_u+2*h));                         // N=2 Simpson's-Regel
        h=(b-a)/(ts*3);                                                             // N=3 Simpson's-3/8-Regel
        I_3 = I_3 + 3*h/8*(f(x_u) + 3*f(x_u+h) + 3*f(x_u+2*h) + f(x_u+3*h));        // N=3 Simpson's-3/8-Regel
        h=(b-a)/(ts*4);                                                             // N=4 Regel
        I_4 = I_4 + 2*h/45*(7*f(x_u) + 32*f(x_u+h) + 12*f(x_u+2*h) + 32*f(x_u+3*h) + 7*f(x_u+4*h)); // N=4 Regel
    }                                                                               // Ende for-Schleife

    // Ausgaben der berechneten Groessen
    printf("Integral N=1: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_1, (Int_f(b)-Int_f(a)) - I_1);
    printf("Integral N=2: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_2, (Int_f(b)-Int_f(a)) - I_2);
    printf("Integral N=3: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_3, (Int_f(b)-Int_f(a)) - I_3);
    printf("Integral N=4: %14.10f \t Abweichung zum wirklichen Wert: %14.10f \n", I_4, (Int_f(b)-Int_f(a)) - I_4);
}   // Ende der Hauptfunktion
```

Übungsblatt Nr. 8

Aufgabe 1 *(10 Punkte)*

Erstellen Sie eine Klasse, die das bestimmte Integral einer Funktion $f(x)$ in den Grenzen $[a, b]$ mittels einer stückweisen numerischen Integration berechnet (ts Teilintervalle). Benutzen Sie hierbei für in den einzelnen Teilintervallen die N=4 Integrationsregel. Die Klasse sollte drei überladene Konstruktoren besitzen. Der Standardkonstruktor berechnet hierbei das Integral in den Grenzen $[0, 1]$ mit $ts = 10$ Teilintervallen. Der Konstruktor mit zwei Argumenten lässt den Benutzer die Integrationsgrenzen $[a, b]$ festlegen und benutzt ebenfalls $ts = 10$ Teilintervalle und der Konstruktor mit drei Argumenten lässt den Benutzer auch die Anzahl der Teilintervalle ts frei wählen. Die zu integrierende Funktion lautet $f(x) = 10 \cdot e^{-x/5} \cdot \sin(3\,x)$ und soll als inline-Methode der Klasse definiert werden. Der Algorithmus der eigentlichen Integration soll als eine, innerhalb der Klasse definierte, öffentliche Member-Funktion definiert werden. Erzeugen Sie dann im Hauptprogramm vier unterschiedliche Objekte (Instanzen der Klasse), wobei alle eine Integration der Funktion in den Grenzen $[a, b] = [1, 2]$ berechnen sollten und die Unterschiedlichkeit lediglich in der Anzahl der Teilintervalle ts besteht (benutzen Sie hierbei $ts = 10$, $ts = 50$, $ts = 100$ und $ts = 1000000$). Lassen Sie sich den berechneten Integralwert und den absoluten Fehler des Wertes zum wirklichen, analytischen Wert im Terminal ausgeben und diskutieren Sie die Ergebnisse. Ist die hier benutzte Integrationsklasse sinnvoll, oder denken Sie, dass es vorteilhafter gewesen wäre, den Algorithmus der Integration lediglich in einer normalen C++ Funktion zu implementieren?

Übungsblatt Nr.8

Aufgabe 2 *(10 Punkte)*

Die im C++ Programm Lagrange_Polynom_Klasse_a.cpp implementierte Lagrange Polynom Klasse approximiert eine vorgegebene Funktion $f(x)$ (hier speziell $f(x) = 1/x$) durch ein Lagrangepolynom. Wir nehmen jedoch im Folgenden an, dass die wirkliche Funktion $f(x)$ unbekannt ist, und wir lediglich an den Stützstellen die Funktionswerte kennen. Gegeben seien die folgenden x- und y-Werte der Stützstellen: $\vec{x} = (1.1, 10.1, 12.9, 25.7, 40.5, 60.2, 95.1, 98.8)$ und $\vec{y} = (2.1, 41.5, 48.2, 35.2, 5.2, 10.6, 27.5, 15.2)$. Schreiben Sie die Lagrange Polynom Klasse um und berechnen Sie das Lagrange Polynom $P_7(x)$ im Teilintervall $[a, b] = [0, 100]$ mittels eines C++ Programms und stellen es grafisch mittels Python dar.