Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
19.05.2022

MATTHIAS HANAUSKE

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

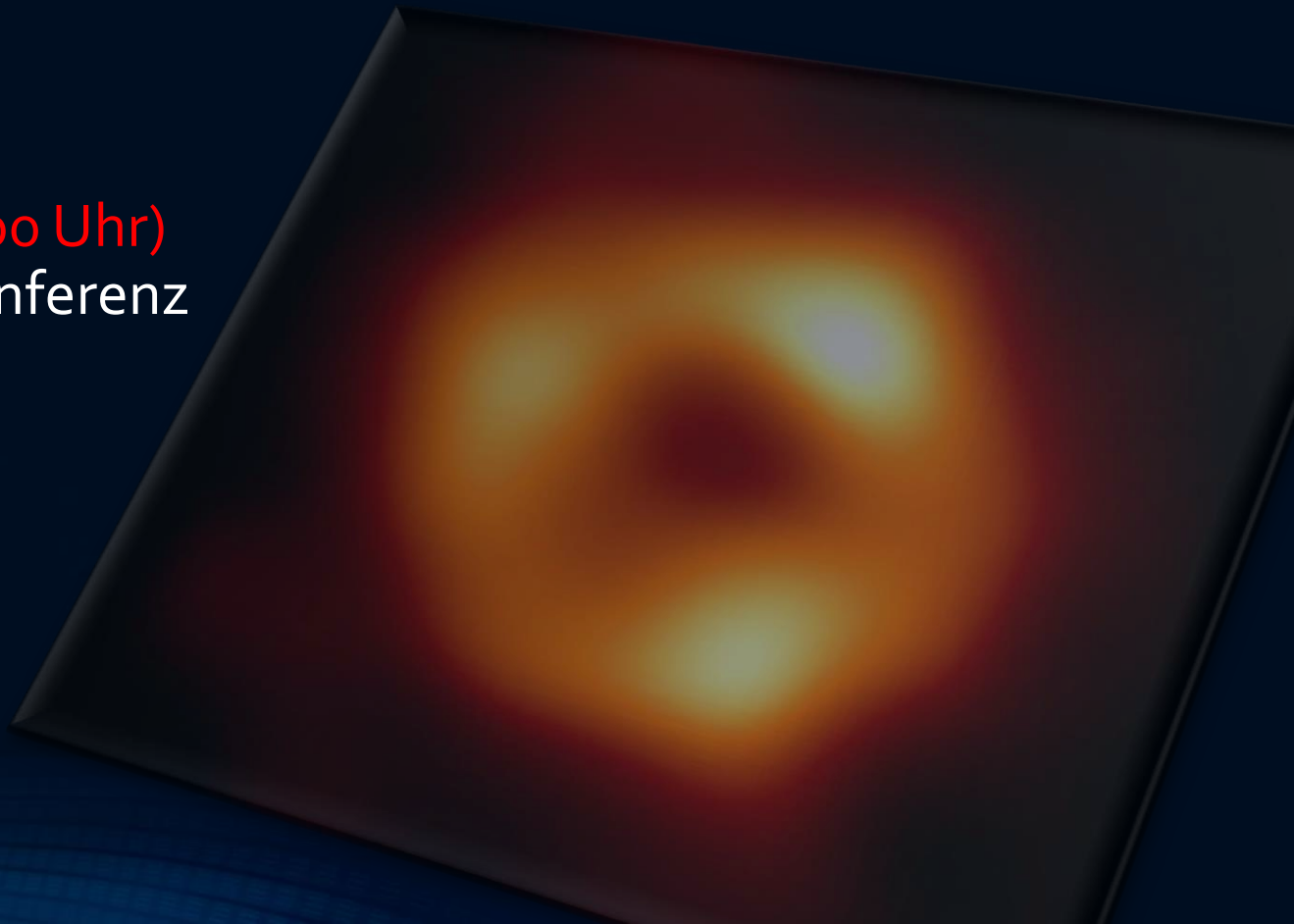
6. Vorlesung

Plan für die heutige Vorlesung

- Kurze Wiederholung der Vorlesung 5
- Mehrdimensionale C++ Arrays
- Jupyter Notebooks und das Rechnen mit symbolischen Ausdrücken
- Numerische Differentiation
- Übungsaufgaben: Übungsblatt Nr.6

Wiederholung der Vorlesung 5

- C++ Arrays, Zeiger und Referenzen
- Anwendungsbeispiel: Interpolation und Polynomapproximation
- Übungsaufgaben: Übungsblatt Nr.5
- Aus aktuellem Anlass
  (Donnerstag, der 12.05.2022 ab 15.00 Uhr)
  Liveübertragung der ESO-Pressekonferenz
  Neues vom Schwarzen Loch
  im Zentrum der Milchstrasse



Zeiger, Adressen und Referenzen

Betrachten wir z.B. eine Variable, die den Wert einer Gleitkommazahl in doppelter Maschinengenauigkeit speichern soll - eine sogenannte double-Variable mit dem Namen "zahl". Für den Wert dieser Variable wurde beim Deklarationsprozess ein Platz von 8 Bytes im Hauptspeicher reserviert. Die Adresse im Hauptspeicher, wo der Wert der Variable binär abgelegt ist, kann man mittels des Referenzoperators (&zahl: ein der Variable vorgestelltes &) ermitteln. In der Sprache C++, sind speziell für den Zweck des Hauptspeicherzugriffs, zwei eigene Datentypen definiert, mittels deren man die Adresse der Variable eines bestimmten Datentyps T speichern kann. Der Datentyp "Zeiger auf T" wird mit einem nachgestellten Sternsymbol gekennzeichnet (T*) und eine Variable dieses neuen Datentyps kann die Adresse eines Typs T speichern. Man sollte eine Zeiger-Variable des Typs T* am besten sofort beim Deklarationsprozess mit einer Adresse initialisieren, da es sonst geschehen kann, dass der Zeiger auf ein nicht existentes Objekt im Hauptspeicher zeigt. Möchte man dennoch einen nicht-intitialisierten Zeiger verwenden, so wird es angeraten diesen mit dem Nullzeiger "nullptr" zu initialisieren (z.B. int* a = nullptr;).

Die unten abgebildete Box zeigt die Verwendung von Zeigern, Adressen und Referenzen am Beispiel des Datentyps double.

Zeiger, Adresse und Referenz am Beispiel des Datentyps T → double

Deklaration einer double-Variable und Initialisierung mit einer Gleitkommazahl:

```
double zahl = 2.47654673;
```

Definition des Zeigers auf die Adresse der double-Variable:

```
double* zeiger_zahl = &zahl;
```

Definition der Referenz der double-Variable

```
double& ref_zahl = zahl;
```

Dereferenzierung des Zeigers (vorgestelltes Sternzeichen) liefert den Wert der double-Variable:

```
double stern_zeiger_zahl = *zeiger_zahl;
```

Zeiger, Adressen und Referenzen

Ein weiterer neuer Datentyp ist die "Referenz eines Typs T" und diese wird bei der Deklaration des Typennamens mit einem nachgestellten &-Symbol gekennzeichnet (T&). Eine Referenz ist im Prinzip gleichbedeutend mit der Adresse des Objektes, wobei im Unterschied zum Zeigerkonstrukt eine automatische Umwandlung (Dereferenzierung) der Adresse in den Wert der Variable geschieht. Eine Referenz ist demnach eine Art von Zeiger, der bei jeder Verwendung im Programm dereferenziert wird. Bei einer Zeigervariable Z des Typs T erhält man den Wert mittels eines vorgestellten Sternsymbols "*" (*Z: Dereferenzieren eines Zeigers, Inhaltsoperator "*" angewandt auf den Zeiger Z), wobei bei einer Referenz das Dereferenzieren automatisch geschieht. Eine Referenz bezieht sich immer auf das Objekt, mit dem sie initialisiert wurde und es gibt keine Null-Referenzen im Gegensatz zum Null-Zeiger.

Die unten abgebildete Box zeigt die Verwendung von Zeigern, Adressen und Referenzen am Beispiel des Datentyps double.

Zeiger, Adresse und Referenz am Beispiel des Datentyps T → double

Deklaration einer double-Variable und Initialisierung mit einer Gleitkommazahl:

```
double zahl = 2.47654673;
```

Definition des Zeigers auf die Adresse der double-Variable:

```
double* zeiger_zahl = &zahl;
```

Definition der Referenz der double-Variable

```
double& ref_zahl = zahl;
```

Dereferenzierung des Zeigers (vorgestelltes Sternzeichen) liefert den Wert der double-Variable:

```
double stern_zeiger_zahl = *zeiger_zahl;
```

Zeiger, Adressen und Referenzen

am Beispiel von char, int und double

Das folgende C++ Programm illustriert die V

Zeiger_1.cpp

```
#include <iostream>                                  // Ein- u

int main(){                                          // Haupt
    char zeichen = 'H';                              // Defini
    char* zeiger_zeichen = &zeichen;                 // Defini
    char& ref_zeichen = zeichen;                     // Defini
    char stern_zeiger_zeichen = *zeiger_zeichen; // char \

    int ganze_zahl = 234767;                                 // D
    int* zeiger_ganze_zahl = &ganze_zahl;                    // D
    int& ref_ganze_zahl = ganze_zahl;                        // D
    int stern_zeiger_ganze_zahl = *zeiger_ganze_zahl; // i

    double zahl = 2.47654673;                  // Definitior
    double* zeiger_zahl = &zahl;               // Definitior
    double& ref_zahl = zahl;                   // Definitior
    double stern_zeiger_zahl = *zeiger_zahl; // double Var

    printf("Wert der Variable 'zeichen' ist: %c \n", zeichen);                          // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_zeichen' ist: %p \n", zeiger_zeichen);            // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_zeichen' ist: %c \n", stern_zeiger_zeichen); // Ausgabe der Dereferenzierung des Zeigers
    printf("Wert der Variable 'ref_zeichen' ist: %c \n", ref_zeichen);                  // Ausgabe der Referenz (konstanter Zeiger, der direkt dereferenziert wird)
    printf("------------------------------\n");
    printf("Wert der Variable 'ganze_zahl' ist: %i \n", ganze_zahl);                    // Ausgabe des Wertes der Variable
    printf("Wert der Variable 'zeiger_ganze_zahl' ist: %p \n", zeiger_ganze_zahl);      // Ausgabe des Zeigers auf die Adresse der Variable
    printf("Wert der Variable 'stern_zeiger_ganze_zahl' ist: %i \n", stern_zeiger_ganze_zahl); // ....
    printf("Wert der Variable 'ref_ganze_zahl' ist: %i \n", ref_ganze_zahl);
    printf("------------------------------\n");
    printf("Wert der Variable 'zahl' ist: %f \n", zahl);
    printf("Wert der Variable 'zeiger_zahl' ist: %p \n", zeiger_zahl);
    printf("Wert der Variable 'stern_zeiger_zahl' ist: %f \n", stern_zeiger_zahl);
    printf("Wert der Variable 'ref_zahl' ist: %f \n", ref_zahl);
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ g++ Zeiger_1.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$ ./a.out
Wert der Variable 'zeichen' ist: H
Wert der Variable 'zeiger_zeichen' ist: 0x7ffc0bf541be
Wert der Variable 'stern_zeiger_zeichen' ist: H
Wert der Variable 'ref_zeichen' ist: H
------------------------------
Wert der Variable 'ganze_zahl' ist: 234767
Wert der Variable 'zeiger_ganze_zahl' ist: 0x7ffc0bf541c0
Wert der Variable 'stern_zeiger_ganze_zahl' ist: 234767
Wert der Variable 'ref_ganze_zahl' ist: 234767
------------------------------
Wert der Variable 'zahl' ist: 2.476547
Wert der Variable 'zeiger_zahl' ist: 0x7ffc0bf541c8
Wert der Variable 'stern_zeiger_zahl' ist: 2.476547
Wert der Variable 'ref_zahl' ist: 2.476547
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Arrays$
```

Eindimensionale integrierte C++ Arrays *(Vektoren)*

Navigieren in Arrays

Der Zugriff auf den Wert eines Array-Elementes kann entweder durch die Angabe des entsprechenden Vektor-Index (v[i]), oder durch den dereferenzierten Wert seiner Zeigerposition im Array erfolgen: *(zeiger_v+ i). Es gelten dabei die folgenden äquivalenten Formulierungen, um den Wert des i-ten Eintrages im Array zu erhalten: v[i] =*(v+i) = *(&v[o]+i) . Diese Eigenschaft wird in den vier letzten Zeilen der Terminalausgabe für i=3 überprüft.

Array_1_zeiger.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek
using namespace std;                        // Benutze den Namensraum std

int main(){                                 // Hauptfunktion
    int v[7] = {1,4,6,8,9,5,3};             // Definition eines Integer-Arrays mit sieben Einträgen
    int* zeiger_v = v;                      // Definition des Zeigers auf das Integer-Array v
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    cout << "Das Array v ist ein Zeiger auf sein erstes Element: v=" << v << " und &v[0]=" << &v[0] << endl;
    cout << "Die Dimension unseres Arrays v ist: dim(v)=" << dim_v << endl << endl;

    printf("%20s %20s %20s %25s %30s %30s \n", "Index i Array", "Wert v[i]", "Referenz &v[i]", "Zeiger zeiger_v+i", "Deref. Zeiger *(zeiger_v+i)", "Deref. Adresse *(&v[i])");

    for(int i=0; i<dim_v; ++i){         // Schleifen Anfang ueber alle
        printf("%20i ", i);             // Ausgabe des Indexes i
        printf("%20i ", v[i]);          // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &v[i]);         // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_v+i);    // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%30i ", *(zeiger_v+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%30i \n", *(&v[i]));    // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                   // Ende der Schleife
    printf("\nEs gilt z.B. für i=3:\nv[3] = %i \n*(v+3) = %i \n*(&v[0]+3) = %i \n", v[3], *(v+3), *(&v[0]+3));
}
```

Strings als eindimensionale Arrays von Zeichen

am Beispiel des „Hallo Welt" Programms

Strings als Arrays von Zeichen

Obwohl wir es im Großteil der Vorlesung mit Zahlen-Arrays zu tun haben werden, wollen wir die Verwendung von anderen C++ Array-Typen auch einmal am Beispiel eines Arrays bestehend aus Zeichen (ein String, ein eindimensionales Array vom Typ char) verdeutlichen. Ein String ist eine Abfolge (ein Array) von mehreren Zeichen. Nehmen wir z.B. den Satz "Hallo, du schoene Welt!". Dieser String stellt ein Array des Typs char dar, wobei seine Dimension N die Anzahl der Zeichen (hier speziell N=23) ist. Im folgenden Programm wurde dieses char-Array mittels der Anweisung char satz[] = {'H','a', ... ,'!'}; deklariert und sofort mit den einzelnen Zeichen initialisiert. Initialisiert man ein Array direkt bei seiner Deklaration, muss man die Dimension des Arrays (hier N=23) nicht explizit angeben. In gleicher Weise wie im vorigen Beispiel des int-Arrays, wird der Zeiger auf das Array definiert, die Dimension des Arrays berechnet und mehrere Größen der jeweiligen Elemente des Arrays ausgegeben.

HelloWorld_1_zeiger.cpp

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

int main(){                                 // Hauptfunktion
    char satz[] = {'H','a','l','l','o',',',' ','d','u',' ','s','c','h','o','e','n','e',' ','W','e','l','t','!'}; // Deklaration des eindimensionalen Arrays von Zeichentypen und Initialisierung
    char* zeiger_satz = satz;                       // Deklaration des Zeigers auf das eindimensionalen Arrays von Zeichentypen
    int anz_zeichen = sizeof(satz)/sizeof(satz[0]); // Dimension des eindimensionalen Arrays (dim= 23 = 92/4 = (Byte fuer den gesamten Satz)/(Byte für ein Zeichen)

    printf("%20s %20s %20s %25s %35s %35s \n", "Index i Array", "Zeichen satz[i]", "Referenz &satz[i]", "Zeiger zeiger_satz+i", "Deref. Zeiger *(zeiger_satz+i)", "Deref. Adresse *(&satz[i])");

    for(int i=0; i<anz_zeichen; ++i){       // Schleifen Anfang ueber alle Zeichen im Satz
        printf("%20i ", i);                 // Ausgabe des Indexes i
        printf("%20c ", satz[i]);           // Ausgabe des i-ten Wertes des Arrays
        printf("%20p ", &satz[i]);          // Ausgabe der Referenz (Addresse) des i-ten Eintrages im Array
        printf("%25p ", zeiger_satz+i);     // Ausgabe des Zeigers auf den i-ten Eintrag im Array
        printf("%35c ", *(zeiger_satz+i)); // Dereferenzierung des Zeigers auf den i-ten Eintrag im Array
        printf("%35c \n", *(&satz[i]));     // Dereferenzierung der Addresse des i-ten Eintrages im Array
    }                                       // Ende der Schleife
}
```

Array_funktionen_1.cpp

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

/** Funktion ohne Rueckgabewert, die als Argument den Zeiger auf
 * ein eindimensionales int-Array und seine Dimension hat.
 * Falls die Dimension groesser als 5 ist, vertauscht sie den vierten
 * Eintrag im Array (v[3]) mit dem sechsten (v[5])
 **/
void Tausche_35(int* pv, int dim){
    if( dim > 5 ){
        printf("Vertausche v[3] mit v[5] \n");
        int tmp = *(pv+3);                  // Speichert den Wert von v[3]
        *(pv+3) = *(pv+5);                  // Schreibt den Wert von v[5] in v[3]
        *(pv+5) = tmp;                      // Schreibt den alten Wert von v[3] in v[5]
    }
}

int main(){                                 // Hauptfunktion
    int v[] = {1,4,6,8,5,3,8,9,3};          // Definition eines Integer-Arrays mit neun Einträgen
    int dim_v = sizeof(v)/sizeof(v[0]);     // Dimension des Arrays

    //Ausgabe von v mittels einer Index-basierten for-Schleife
    printf("v = ( %2i", v[0]);
    for(int i=1; i<dim_v; ++i){
        printf(" ,%2i", v[i]);
    }
    printf(" )\n");

    Tausche_35(v, dim_v);

    //Ausgabe von v mittels einer Bereichsbasierten for-Schleife
    printf("v = ( ");
    for(int n : v){         // Man haette an dieser Stelle auch "for(auto n : v){" schreiben koennen
        printf("%2i, ", n);
    }
    printf("\b\b )\n");
}
```

Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array (dies gilt auch für mehrdimensionale Arrays) als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension.

C++ Arrays und Funktionen

Anwendungsbeispiel
Interpolation und Polynomapproximation

Anwendungsbeispiel: Interpolation und Polynomapproximation

Die Interpolation ist ein Teilgebiet der numerischen Mathematik und es befasst sich mit der Problematik, eine analytische Funktion mittels einer gegebenen Menge von Werten zu bestimmen. Wir werden in diesem Unterpunkt den wohl elementarsten Algorithmus einer Interpolation vorstellen, bei dem ein Polynom $P(x)$ mittels einer gegebenen Liste von Werten konstruiert wird. Aufgrund des *Weierstraßschen Approximationssatzes* gibt es für jede, auf einem Teilintervall $[a, b]$ definierte, stetige Funktion, $f(x)$ ein Polynom $P(x)$ mit der Eigenschaft, dass für jedes $\epsilon > 0$ die Differenz der Funktion mit dem Polynom verschwindend klein wird ($|f(x) - P(x)| < \epsilon \; \forall \, x \in [a, b]$). Die Taylorschen Polynome sind hierbei wohl das bekannteste Beispiel einer Entwicklung einer Funktion in ein Polynom. Leider haben die Taylorschen Polynome den Nachteil, dass sie von ihrer Konstruktion her nur eine Stützstelle $x_0 \in [a, b]$ verwenden und somit die Funktion $f(x)$ zwar gut in dem Bereich um die Stützstelle beschreiben, aber nicht im gesamten Teilintervall $[a, b]$. Bei einer Interpolation mittels der *Methode der Lagrange Polynome* verwendet man hingegen mehrere Stützstellenpunkte für die Konstruktion des approximierenden Polynoms. Dieses Verfahren ist Gegenstand dieses Anwendungsbeispiels.

Die Theorie der Entwicklung einer Funktion in ein Lagrange Polynom

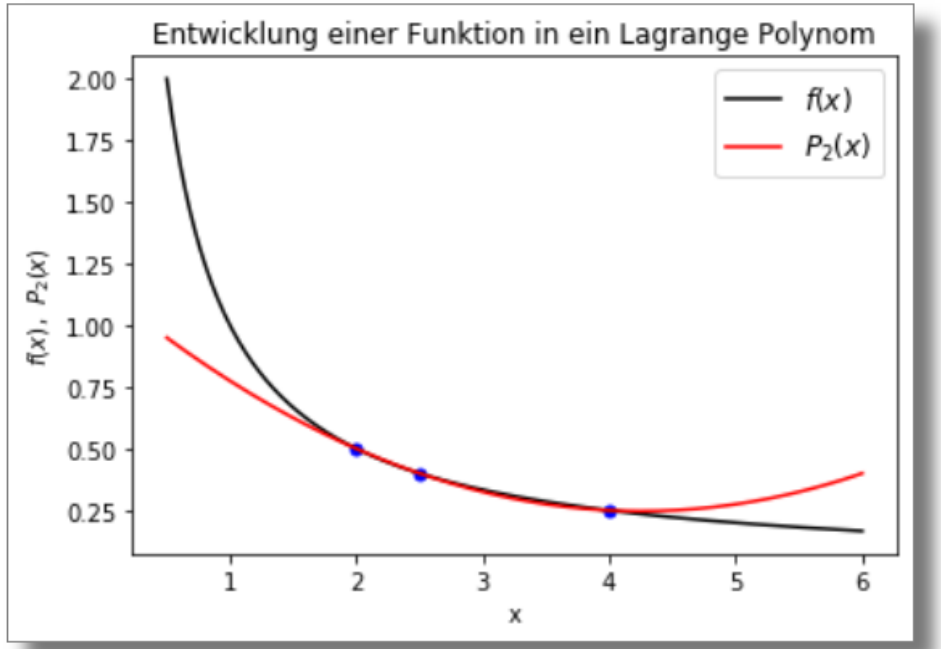
Bei der Entwicklung einer Funktion in ein Lagrange Polynom steht man vor der Aufgabe eine Funktion $f(x)$ mittels $(n + 1)$ vorgegebener Punkte $((x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)),)$ in ein Lagrange Polynom vom Grade n zu entwickeln. Das n-te Lagrange Polynom $P_n(x)$ einer Funktion $f(x)$ ist wie folgt definiert:

$$P_n(x) = \sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x) \quad \text{, wobei} \quad L_{n,k}(x) = \prod_{i=0,\, i \neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

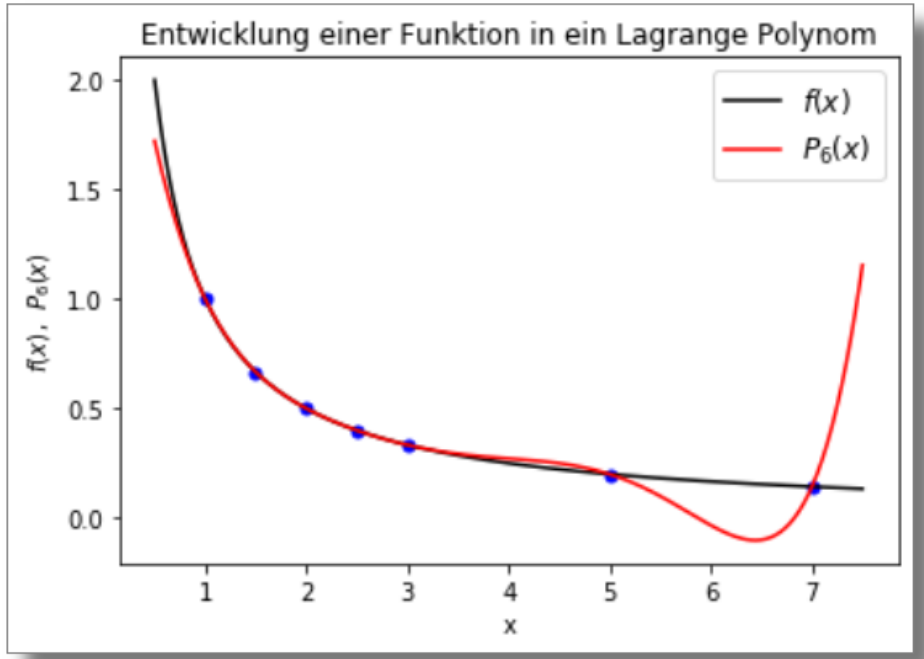
Lagrange Polynome

In diesem Unterpunkt werden wir die Interpolation einer Funktion f(x) mittels der *Methode der Lagrange Polynome* vorstellen, bei der man mehrere Stützstellenpunkte für die Konstruktion des approximierenden Polynoms verwendet.

Im Gegensatz zu Taylorschen Polynomen, benutzt die Methode der Lagrange Polynome mehrere unterschiedliche Punkte der Funktion bei der Entwicklung der Funktion in ein Polynom. Taylorschen Polynome haben somit den Nachteil, dass sie von ihrer Konstruktion her nur eine Stützstelle $x_0 \in [a,b]$ verwenden und somit die Funktion f(x) zwar gut in dem Bereich um die Stützstelle beschreiben, aber nicht im gesamten Teilintervall [a,b].



Die nebenstehende linke Abbildung verdeutlicht die oben beschriebene Lagrange Polynom Entwicklung, wobei die blauen Punkte die drei vorgegebenen Stützstellen des Polynoms darstellen, bei welchen die Übereinstimmung $f(x) = P_2(x)$ exakt gilt. Die rechte Abbildung zeigt das entsprechende Polynom $P_6(x)$, wobei hierbei die sieben Stützstellen an den folgenden x-Werten genommen wurden: $\vec{x} = (1, 1.5, 2, 2.5, 3, 5, 7)$.



$$P_n(x) = \sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x) \quad \text{, wobei} \quad L_{n,k}(x) = \prod_{i=0,\, i\neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)} \quad .$$

Approximiert man z.B. die Funktion $f(x) = \frac{1}{x}$ im Teilintervall $[a, b] = [0.5, 6]$ mittels dreier vorgegebener Punkte ($n = 2$, mit $n + 1$ Punkten $(2, f(2)), (2.5, f(2.5))$ und $(4, f(4))$) in ein Lagrange Polynom vom Grade 2, so erhält man:

$$P_2(x) = \sum_{k=0}^{2} f(x_k) \cdot L_{2,k}(x) = f(x_0) \cdot L_{2,0}(x) + f(x_1) \cdot L_{2,1}(x) + f(x_2) \cdot L_{2,2}(x)$$

$$= \frac{1}{2} \cdot L_{2,0}(x) + \frac{1}{2.5} \cdot L_{2,1}(x) + \frac{1}{4} \cdot L_{2,2}(x)$$

mit:

$$L_{2,0}(x) = \prod_{i=0,\, i\neq 0}^{2} \frac{(x - x_i)}{(x_0 - x_i)} = \frac{(x - x_1)}{(x_0 - x_1)} \cdot \frac{(x - x_2)}{(x_0 - x_2)} = \frac{(x - 2.5)}{(2 - 2.5)} \cdot \frac{(x - 4)}{(2 - 4)}$$

$$L_{2,1}(x) = \prod_{i=0,\, i\neq 1}^{2} \frac{(x - x_i)}{(x_1 - x_i)} = \frac{(x - x_0)}{(x_1 - x_0)} \cdot \frac{(x - x_2)}{(x_1 - x_2)} = \dots$$

$$L_{2,2}(x) = \prod_{i=0,\, i\neq 2}^{2} \frac{(x - x_i)}{(x_2 - x_i)} = \frac{(x - x_0)}{(x_2 - x_0)} \cdot \frac{(x - x_1)}{(x_2 - x_1)} = \dots$$

Beispiel:
Lagrange Polynom vom Grade n=2
3 Stützpunkte

$$\Longrightarrow P_2(x) = 0.05\, x^2 - 0.425\, x + 1.15$$

Fehlerformel der Lagrange Polynom Methode

Die Abbildungen zeigen, dass die Übereinstimmung des berechneten Polynoms $P_n(x)$ mit der Funktion $f(x)$ nur an den Stützstellen exakt ist. Betrachtet man sich den Unterschied beider Funktionen an einem beliebigen x-Wert im Bereich der Stützstellen, so kann man die folgende Fehlerformel des Lagrangeschen Polynoms herleiten

$$f(x) = P_n(x) + \underbrace{\frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i)}_{\text{Fehler}} \ , \ \forall\ \xi(x) \in [x_0, x_n] \quad ,$$

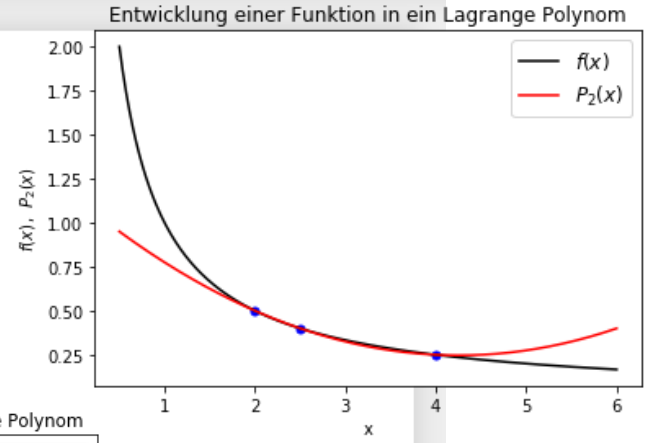
wobei $f^{(n+1)(x)}$ die $(n+1)$-te Ableitung der Funktion darstellt und $\xi(x)$ ein x-Wert im Bereich der Stützstellen ist.

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom
 * 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 3 Punkte ( (2,f(2)), (2.5,f(2.5)), (4,f(4)) )
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_1.dat" */
#include <iostream>                                                 // Ein- und Ausgabebibliothek

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)

int main(){                                                         // Hauptfunktion
    double points[] = { 2, 2.5, 4 };                                // Deklaration und Initialisierung der Stützstellen als double-Array
    const 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
    double Lk;                                                      // Deklaration einer Variable, die fuer Zwischenrechnungen benoetigt wird

    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]=f(x);                                                 // Eintrag des aktuellen f(x)-Wertes in das f(x)-Array

        Pfp[j]=0;                                                   // Initialisierung des j-ten Polynom-Arrays mit 0 (Wert beim Anfang der Summenbildung)
        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[j] = Pfp[j] + f(points[k])*Lk;                      // Kern-Gleichung in der Lagrange Polynom Methode
        }                                                           // Ende for-Schleife der Summenbildung
    }                                                               // 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
```

```cpp
/* Entwicklung einer Funktion in ein Lagrange Polynom
 * 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 3 Punkte ( (2,f(2)), (2.5,f(2.5)), (4,f(4)) )
 * Ausgabe zum Plotten (Gnuplot oder Python) mittels: "./a.out > Lagrange_Polynom_1.dat" */
#include <iostream>                                                 // Ein- und Ausgabebibliothek

double f(double x){                                                 // Deklaration und Definition der
    double wert;
    wert = 1.0/x;                                                   // Eigentliche Definition der Funk
    return wert;                                                    // Rueckgabewert der Funktion f(x)
}                                                                   // Ende der Funktion f(x)

int main(){                                                         // Hauptfunktion
    double points[] = { 2, 2.5, 4 };                                // Deklaration und Initialisierung
    const unsigned int N_points = sizeof(points)/sizeof(points[0]); // Anzahl der Punkte die zur Appro
    double plot_a=0.5;                                              // Untergrenze des x-Intervalls in
    double plot_b=6;                                                // Obergrenze des x-Intervalls in
    const unsigned int N_xp=300;                                    // Anzahl der Punkte in die das x-
    double dx = (plot_b - plot_a)/N_xp;                             // Abstand dx zwischen den aequidi
    double x = plot_a-dx;                                           // Aktueller x-Wert
    double xp[N_xp+1];                                              // Deklaration der x-Ausgabe-Punkt
    double fp[N_xp+1];                                              // Deklaration der f(x)-Ausgabe-Pu
    double Pfp[N_xp+1];                                             // Deklaration der Ausgabe-Punkte
    double Lk;                                                      // Deklaration einer Variable, die

    printf("# x-Werte der %3d Stuetzstellen-Punkte: \n", N_points); // Beschreibung der ausgegebenen G
    for(int k = 0; k < N_points; ++k){                              // For-Schleife der Ausgabe der St
        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 G
    printf("# 3: Approximierter Wert des Lagrange Polynoms P(x) \n"); // Beschreibung der ausgegebenen G
    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 ei
        x=x+dx;                                                     // Aktueller x-Wert
        xp[j]=x;                                                    // Eintrag des aktuellen x-Wertes
        fp[j]=f(x);                                                 // Eintrag des aktuellen f(x)-We

        Pfp[j]=0;                                                   // Initialisierung des j-ten Poly
        for(int k = 0; k < N_points; ++k){                          // For-Schleife der Summation in
            Lk=1;                                                   // Initialisierung der Produktva
            for(int i = 0; i < N_points; ++i){                      // For-Schleife der Produktbildun
                if(i != k){                                         // Die Produktbildung soll nur f
                    Lk = Lk * (x - points[i])/(points[k] - points[i]); // Berechnung der Lk-Werte in de
                }                                                   // Ende if-Bedingung
            }                                                       // Ende for-Schleife der Produkt
            Pfp[j] = Pfp[j] + f(points[k])*Lk;                      // Kern-Gleichung in der Lagrange
        }                                                           // Ende for-Schleife der Summenbi
    }                                                               // Ende der for-Schleife ueber di

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

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V4$ ./a.out
# x-Werte der   3 Stuetzstellen-Punkte:
   2.00000   2.50000   4.00000
# 0: Index j
# 1: x-Wert
# 2: f(x)-Wert
# 3: Approximierter Wert des Lagrange Polynoms P(x)
# 4: Fehler zum wirklichen Wert f(x)-P(x)
  0   0.5000000000   2.0000000000   0.9500000000   1.0500000000
  1   0.5183333333   1.9292604502   0.9431418056   0.9861186446
  2   0.5366666667   1.8633540373   0.9363172222   0.9270368150
  3   0.5550000000   1.8018018018   0.9295262500   0.8722755518
  4   0.5733333333   1.7441860465   0.9227688889   0.8214171576
  5   0.5916666667   1.6901408451   0.9160451389   0.7740957062
  6   0.6100000000   1.6393442623   0.9093550000   0.7299892623
  7   0.6283333333   1.5915119363   0.9026984722   0.6888134641
  8   0.6466666667   1.5463917526   0.8960755556   0.6503161970
  9   0.6650000000   1.5037593985   0.8894862500   0.6142731485
 10   0.6833333333   1.4634146341   0.8829305556   0.5804840786
 11   0.7016666667   1.4251781473   0.8764084722   0.5487696750
 12   0.7200000000   1.3888888889   0.8699200000   0.5189688889
 13   0.7383333333   1.3544018059   0.8634651389   0.4909366670
```

```
290   5.8166666667   0.1719197708   0.3695972222  -0.1976774514
291   5.8350000000   0.1713796058   0.3724862500  -0.2011066442
292   5.8533333333   0.1708428246   0.3754088889  -0.2045660643
293   5.8716666667   0.1703093954   0.3783651389  -0.2080557435
294   5.8900000000   0.1697792869   0.3813550000  -0.2115757131
295   5.9083333333   0.1692524683   0.3843784722  -0.2151260040
296   5.9266666667   0.1687289089   0.3874355556  -0.2187066467
297   5.9450000000   0.1682085786   0.3905262500  -0.2223176714
298   5.9633333333   0.1676914477   0.3936505556  -0.2259591078
299   5.9816666667   0.1671774868   0.3968084722  -0.2296309855
300   6.0000000000   0.1666666667   0.4000000000  -0.2333333333
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V4$
```



Einführung in die Programmierung für Studierende der Physik

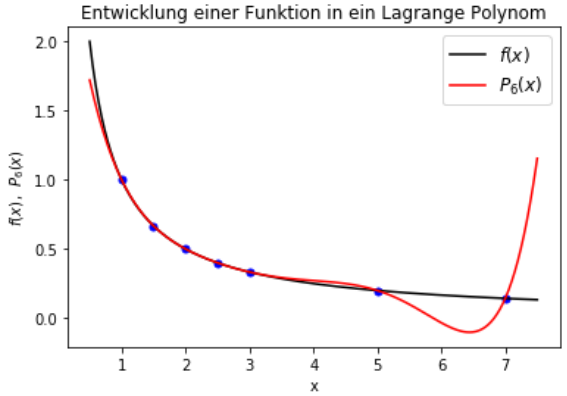
(Introduction to Programming for Physicists)

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 01.04.2022



Entwicklung einer Funktion in ein Lagrange Polynom

Zunächst wird das Python Modul "sympy" eingebunden, das ein Computer-Algebra-System für Python bereitstellt und eine Vielzahl an symbolischen Berechnungen im Bereich der Mathematik und Physik relativ einfach möglich macht. 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()
```

Wir möchten nun die Funktion $f(x) = \frac{1}{x}$ mittels $(n+1)$ vorgegebener Punkte $((x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)),)$ in ein Lagrange Polynom vom Grade N entwickeln. Hierzu definieren wir zunächst die Funktion:

Übungsblatt Nr. 5

Aufgabe 1 *(7.5 Punkte)*

Der im C++ Programm Lagrange_Polynom_1.cpp implementierte Algorithmus stellt eine nützliche Implementierung der Lagrange Polynom Methode dar, die wir vielleicht auch in zukünftigen Programmen verwenden können. Definieren Sie eine C++ Funktion, die den Kern-Algorithmus des Programms Lagrange_Polynom_1.cpp beinhaltet. Die Struktur der Funktion sollte dem allgemeinen C++ Funktionenschema "'Rückgabe Typ' **Funktionsname** ('Argumentenliste') { 'Block von Anweisungen' }" folgen (siehe Funktionen in C++) und in der 'Argumentenliste' sollte einerseits der Zeiger auf das im Hauptprogramm deklarierte Stützstellen-Array und seine Dimension stehen und andererseits sollten die im Hauptprogramm deklarierten Arrays double xp[N_xp+1], double fp[N_xp+1] und double Pfp[N_xp+1] mittels eines Aufrufs der Funktion mit den entsprechenden Datenwerten der Berechnung gefüllt werden. Testen Sie das Programm zunächst mittels des vorgegebenen Beispieles ($f(x) = 1/x$ und $\vec{x} = (2, 2.5, 4)$) und berechnen dann das Lagrange Polynom vom Grade $n = 8$ im Teilintervall $[a, b] = [0.91, 9.07]$ mit $f(x) = 10\,e^{-x/5} \cdot \sin(3\,x)$ und $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8, 9)$.

Aufgabe 2 *(7.5 Punkte)*

Die im C++ Programm Lagrange_Polynom_1.cpp implementierte Lagrange Polynom Methode 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: $\vec{x} = (2, 10.1, 15, 20, 40, 60, 90)$ und $\vec{y} = (2.1, 41, 43, 40.2, 12, 5, 17.5)$. 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.

Aufgabe 3 *(5 Punkte)*

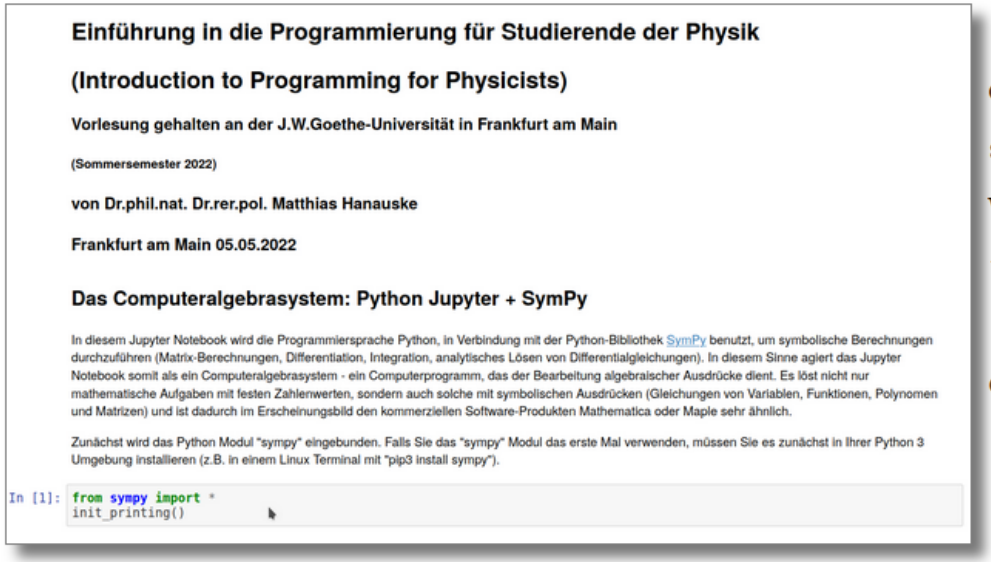
Im Unterpunkt C++ Arrays, Zeiger und Referenzen hatten wir gesehen, wie man ein Array an eine Funktion als Argument übergeben sollte: "Möchte man ein Array an eine Funktion als Argument übergeben, sollte man dies nicht über die einzelnen Werte des Arrays machen, da man dann die Array-Einträge nicht verändern kann. Stattdessen übergibt man ein Array (dies gilt auch für mehrdimensionale Arrays) als Zeiger auf sein erstes Element mit einem zusätzlichen Vermerk zu seiner Dimension." In gleicher Weise gilt dies auch für die gewöhnlichen skalaren Datentypen (int, double, ...). Konstruieren Sie eine Tausch-Funktion für zwei skalare double-Datentypen ("double a" und "double b"). Die Funktion soll dabei die im Hauptprogramm initialisierten Werte von "a" und "b" vertauschen, sodass nach dem Funktionsaufruf die Variable "a" den Wert von "b" und die Variable "b" den ursprünglich initialisierten Wert von "a" besitzt. Verwenden Sie hierbei als 'Rückgabe Typ' void und als 'Argumentenliste' den Zeiger auf a und auf b. Was wäre geschehen, hätten Sie anstatt der Zeiger die Werte der Variablen an die Funktion übergeben?

Vorlesung 6

Mehrdimensionale C++ Arrays

In diesem Unterpunkt werden wir uns mit der Deklaration von integrierten *mehrdimensionale C++ Arrays* befassen. C++ Zahlen-Arrays haben bei der Programmierung von physikalischen Problemen eine bedeutende Rolle, da sie der numerischen Implementierung von Matrizen entsprechen. Obwohl die eigentliche Definition dieser zusammenhängenden Objekte einfachen Darstellungen folgt und z.B. der Zugriff auf die Elemente eines C++ Arrays mittels einer, in der Physik gebräuchlichen Index-Schreibweise, erfolgt, ist die genaue interne Speicherung solcher Matrix-ähnlicher Strukturen nicht so einfach zu verstehen und bedarf eines Zeigerkonzeptes (siehe C++ Arrays, Zeiger und Referenzen). In gleicher Weise wie auch bei eindimensionalen Arrays sind mehrdimensionale C++ Arrays mittels des Konstruktes des Zeigers implementiert. Eine $(m \times n)$-Matrix $\mathcal{A}$, bestehend aus Gleitkommazahlen vom Typ double kann auf tiefster Ebene als ein integriertes Array deklariert werden: double A[m][n]; Der Zugriff auf den Wert eines Array-Elementes kann nun 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)). Danach wird am Beispiel der *Multiplikation zweier Matrizen* der Zugriff auf die einzelnen Elemente des Arrays verdeutlicht. Zuletzt wird das Thema *Mehrdimensionale C++ Arrays und Funktionen* behandelt. C++ Arrays lassen sich nicht direkt als Wert übergeben; stattdessen übergibt man ein Array als Zeiger auf sein erstes Element, mit einer zusätzlichen Angabe seiner Dimensionen (näheres siehe Mehrdimensionale C++ Arrays).

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

Vorlesung 6

Im ersten Teil dieser Vorlesung werden wir uns zunächst mit der Deklaration von *mehrdimensionale C++ Arrays* befassen. Zwei dimensionale C++ Zahlen-Arrays entsprechen der numerischen Implementierung von zwei dimensionale Matrizen, welche in der Physik eine bedeutende Rolle spielen. Möchten wir z.B. eine $(m \times n)$ -Matrix $\mathcal{A} = (a_{ij})_{i=1,2,\ldots,m\,;\,j=1,2,\ldots,n}$, 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:

$$\mathcal{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \Longrightarrow \text{Deklaration: double A[m][n];}$$

In gleicher Weise wie auch bei eindimensionalen Arrays sind mehrdimensionale C++ Arrays mittels des Konstruktes des Zeigers implementiert und die einzelnen Elemente eines Arrays (z.B. "double A[m][n]") sind im Hauptspeicher in Form von Zeigern/Referenzen geordnet.

Der zweite Teil der Vorlesung befasst sich mit dem Thema der *numerischen Differentiation*. Die Ableitung (Differentiation) einer Funktion stellt ein fundamentales Konzept der Mathematik dar, welches bei der Formulierung von physikalischen Bewegungsgleichungen essenziell ist. Die mathematische Definition der Ableitung einer Funktion, der sogenannte Differentialquotient, benutzt dabei eine Grenzwert-Formulierung ($h \to 0$), wobei in der numerischen Mathematik hingegen mehrere *Differentiationsregeln* formuliert werden, die den wirklichen Wert der Ableitung approximieren. Die analytische Herleitung dieser Regeln benutzt die *Methode der Lagrange Polynome* (siehe Anwendungsbeispiel: Interpolation und Polynomapproximation). Im zweiten Unterpunkt des linken Frames dieser Vorlesung wird die Herleitung der *Differentiationsregeln* mittels eines Python Jupyter Notebooks behandelt und im dritten Unterpunkt werden die Differentiationsregeln in einem C++ Programm verwendet, um die Ableitung einer Funktion $f(x)$ approximativ zu bestimmen.

Numerische Differentiation

Die im vorigen Unterpunkt hergeleiteten *Differentiationsregeln der numerischen Mathematik* sind im Folgenden nochmals zusammenfassend dargestellt:

$$\text{Zweipunkteformel (n=1):} \quad f'(x_0) \approx \frac{1}{h}\left[f(x_0+h) - f(x_0)\right]$$

$$\text{Dreipunkte-Endpunkt-Formel (n=2):} \quad f'(x_0) \approx \frac{1}{2h}\left[-3\,f(x_0) + 4\,f(x_0+h) - f(x_0+2\,h)\right]$$

$$\text{Dreipunkte-Mittelpunkt-Formel (n=2):} \quad f'(x_0) \approx \frac{1}{2h}\left[f(x_0+h) - f(x_0-h)\right]$$

$$\text{Fünfpunkte-Mittelpunkt-Formel (n=4):} \quad f'(x_0) \approx \frac{1}{12\,h}\left[f(x_0-2\,h) - 8\,f(x_0-h) + 8\,f(x_0+h) - f(x_0+2\,h)\right]$$

In diesem Unterpunkt werden die *Ableitungsregeln* in einem C++ Programm verwendet, um die numerische Ableitung der Funktion $f(x) = 10\,e^{-x/10} \cdot \sin(x)$ zu approximieren und die Ergebnisse dann mit der wirklichen, analytisch bestimmbaren Ableitung $f'(x)$ zu vergleichen (näheres siehe Numerische Differentiation).

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.

```
 * 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

//Ausgelagerte Funktion, Ausgabe der (m x n)-Matrix im Terminal
void print_Matrix(int* M, int dim_1, int dim_2){ // Anfang des Anweisungsblockes
    for(int i=0; i < dim_1; ++i){                // Anfang Schleife 1 (Zeilen der Matrix)
        printf("| ");
        for(int j=0; j < dim_2; ++j){            // Anfang Schleife 2 (Spalten der Matrix)
            printf("%3i ", M[i*dim_2 +j]);       // Ausgabe des Matrixwertes M_{ij}
        }                                        // Ende der 2.Schleife
        printf("| \n");
    }                                            // Ende der 1.Schleife
}                                                // Ende der Funktion

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

    printf("\nMatrix A: \n");
    print_Matrix(&A[0][0], m, n);                      // Aufruf der Funktion, Matrix A, (m x n)-Matrix

    printf("\nMatrix B: \n");
    print_Matrix(&B[0][0], n, m);                      // Aufruf der Funktion, Matrix B, (n x m)-Matrix

    printf("\nDie Matrix Multiplikation ergibt C = A * B : \n");
    print_Matrix(&C[0][0], m, m);                      // Aufruf der Funktion, Matrix C, (m x m)-Matrix
}
```

Mehrdimensionale C++ Arrays und Funktionen

Arrays lassen sich nicht direkt als Wert übergeben. Stattdessen kann man ein C++ Array als Zeiger auf sein erstes Element übergeben (mit einer zusätzlichen Angabe seiner Dimensionen). Im folgenden C++ Programm wurde eine void-Funktion dem Hauptprogramm ausgelagert, die zur Aufgabe hat eine (m×n)-Matrix im Terminal auszugeben.

Die Funktion "void print_Matrix(int* M, int dim_1, int dim_2){ ... }" wird dann im Hauptprogramm dreimal aufgerufen, um die Matrizen A, B und C darzustellen. Man beachte hierbei, dass beim Aufruf der Funktion im Hauptprogramm (z.B. für die Matrix A: print_Matrix(&A[0][0], m, n);) die Adresse des ersten Elementes der Matrix übergeben wird (&A[0][0]) - man hätte hier auch '&A[0]' schreiben können, jedoch nicht '&A'. Das folgende Bild veranschaulicht die Terminalausgabe.

```
 * 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

//Ausgelagerte Funktion, Ausgabe der (m x n)-Matrix im Terminal
void print_Matrix(int* M, int dim_1, int dim_2){ // Anfang des Anweisungsblockes
    for(int i=0; i < dim_1; ++i){                // Anfang Schleife 1 (Zeilen der Matrix)
        printf("| ");
        for(int j=0; j < dim_2; ++j){            // Anfang Schleife 2 (Spalten der Matrix)
            printf("%3i ", M[i*dim_2 +j]);       // Ausgabe des Matrixwertes M_{ij}
        }                                        // Ende der 2.Schleife
        printf("| \n");
    }                                            // Ende der 1.Schleife
}                                                // Ende der Funktion

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

    printf("\nMatrix A: \n");
    print_Matrix(&A[0][0], m, n);                    // Aufruf der Funktion, Matrix A, (m x n)-Matrix

    printf("\nMatrix B: \n");
    print_Matrix(&B[0][0], n, m);                    // Aufruf der Funktion, Matrix B, (n x m)-Matrix

    printf("\nDie Matrix Multiplikation ergibt C = A * B : \n");
    print_Matrix(&C[0][0], m, m);                    // Aufruf der Funktion, Matrix C, (m x m)-Matrix
}
```

Mehrdimensionale C++ Arrays und Funktionen

Die definierte Funktion hat jedoch mehrere Nachteile und wir werden später, im Themenbereich der *Objekt-orientierten Programmierung* sehen, dass man durch eine *Kapselung* des Programms eine Funktion entwerfen kann, die es dem Anwender erleichtert, die print_Matrix(..)-Funktion aufzurufen. Ein wichtiger Anwendungsbereich der *Objekt-orientierten Programmierung* besteht darin, C++-Klassen und Funktionen zu erstellen, die von anderen Programmierern verwendet werden können. Die Anwendung der geschriebenen Programme sollte dabei möglichst einfach und fehlerfrei sein. Die Anwendung unserer Funktion "void print_Matrix(int* M, int dim_1, int dim_2){ ... }" ist jedoch nicht so einfach und gerade bei der Angabe der Dimensionen des Arrays (int dim_1, int dim_2) kann der Anwender einen Fehler machen, der dann das Programm zum Absturz bringen könnte. Auch ist die print_Matrix Funktion auf ganzzahlige Matrizen beschränkt, was dem Anwender vielleicht nicht klar ist.

Template Funktionen

In dem folgenden C++ Programm sind die genannten Nachteile der ‚print_Matrix' Funktion behoben. Das Programm wurde von **Herrn Johannes Misch** erstellt und es verwendet, in einer eleganten Template-Funktion, zwei ineinander geschachtelte bereichsbasierte for-Schleifen. Beim Aufrufen der Funktion aus dem Hauptprogramm 'print_Matrix(A);' muss der Anwender lediglich das Array in die Argumentenliste der Funktion schreiben und abhängig vom Datentyp der Matrixelemente und der Dimension des Arrays verwendet diese Template-Funktion den geeigneten Matrixtyp.

Array_mehrdim_funktion_template.cpp

```cpp
/** Matrix Multiplikation mit ausgelagerter print-Funktion
 * (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
 * Diese Version benutzt eine Template-Funktion und bereichbasierte for-Schleifen zur Terminalausgabe der Matrix
 * Diese Version des C++ Programm wurde von Johannes Misch erstellt ( misch@itp.uni-frankfurt.de )
 **/
#include <iostream>
#include <iomanip>

// Dies ist ein funktions-template. Eine Blaupause fuer funktionen die der Compiler verwendet um funktionen nach bedarf zu erstellen
// Die funktion akzeptiert ein array der form T arr[a][b];
// sie bestimmt sowohl den type 'T', als auch die groesse des arrays automatisch anhand ihres arguments
template <typename T, std::size_t a, std::size_t b>
void print_Matrix( const T (&M)[a][b] )
{
        // range-based for loops iterieren ueber alle elemente einer collection, in diesem fall ueber alle alle zeilen der matrix
        // const heisst dass wir die zeile nicht modifizieren
        // auto sagt dem compiler, dass er den typ automatisch anhand des initializers bestimmen soll
        // es ist eine reference, weil wir keine Kopie der zeile haben wollen, sondern nur lesen
    for ( const auto& row : M )
    {
        std::cout << "| ";
        for ( const auto value : row ) // diesmal machen wir eine kopie, da dies fuer die meisten fundamentalen typen (wie z.b. int) besser ist
        {
            std::cout << std::setw(8); //wir setzen die breite der naechsten ausgabe auch mindestens 8 zeichen. Dadurch wird die matrix "schoen" dargestellt
            std::cout << value << ' ';
        }
        std::cout << "|\n";
    }
}
```

Template Funktionen

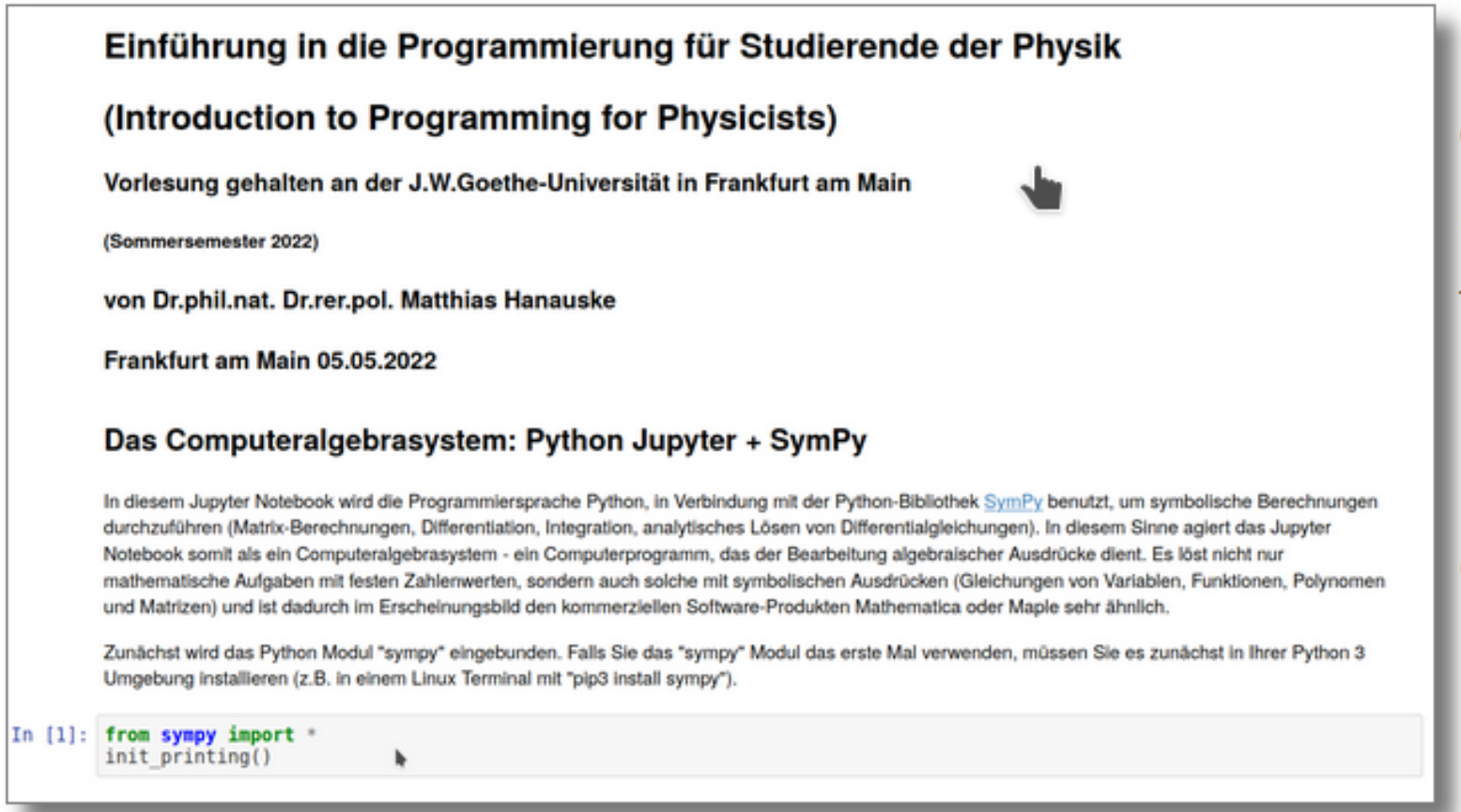
Beim Aufrufen der Funktion aus dem Hauptprogramm 'print_Matrix(A);' muss der Anwender lediglich das Array in die Argumentenliste der Funktion schreiben und abhängig vom Datentyp der Matrixelemente und der Dimension des Arrays verwendet diese Template-Funktion den geeigneten Matrixtyp.

Array_mehrdim_funktion_template.cpp

```cpp
/** Matrix Multiplikation mit ausgelagerter print-Funktion
 * (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
 * Diese Version benutzt eine Template-Funktion und bereichbasierte for
 * Diese Version des C++ Programm wurde von Johannes Misch erstellt ( m
 **/
#include <iostream>
#include <iomanip>

// Dies ist ein funktions-template. Eine Blaupause fuer funktionen die
// Die funktion akzeptiert ein array der form T arr[a][b];
// sie bestimmt sowohl den type 'T', als auch die groesse des arrays au
template <typename T, std::size_t a, std::size_t b>
void print_Matrix( const T (&M)[a][b] )
{
        // range-based for loops iterieren ueber alle elemente einer co
        // const heisst dass wir die zeile nicht modifizieren
        // auto sagt dem compiler, dass er den typ automatisch anhand d
        // es ist eine reference, weil wir keine Kopie der zeile haben
    for ( const auto& row : M )
    {
        std::cout << "| ";
        for ( const auto value : row ) // diesmal machen wir eine kopie
        {
            std::cout << std::setw(8); //wir setzen die breite der naec
            std::cout << value << ' ';
        }
        std::cout << "|\n";
    }
}
```

```cpp
int main()
{

    constexpr int m = 3; // Dimension der Zeilen der Matrix A
    constexpr int n = 5; // Dimension der Spalten der Matrix A
        //constexpr bezeichnet eine compile-time konstante. Built-in arrays muessen eine compile-time konstante groesse haben.

    // Definition eines (3x5)-Integer-Arrays, (3x5)-Matrix
        // 'const' heisst das wir dieses array waehrend der restlichen laufzeit nicht mehr aendern werden
        // man sollte i.A. immer 'const' verwenden wenn man nicht vor hat die variable zu veraendern
    const int A[m][n] =
    {
        {  1,  2,  3,  4,  5 },
        { 21, 22, 23, 24, 25 },
        { 51, 52, 53, 54, 55 }
    };

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

    // Definition eines (3x3)-Integer-Arrays, (3x3)-Matrix. Automatische initializierung mit 0
        // Dieses array is nicht 'const' da es das ergebnis enthalten wird
    int C[m][m] = {}; // eine leere initializer-list fuert dazu das alle werte im array mit 0 initialized werden

    // Matrix Multiplikation C = A * B
    for ( int i=0; i < m; ++i )
    {
        for ( int j=0; j < m; ++j )
        {
            for ( int k=0; k < n; ++k )
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    std::cout << "\nMatrix A: \n";
    print_Matrix( A ); // Dies ruft print_Matrix<int,3,5> auf. Der compiler bestimmt diese parameter automatisch anhand des types von 'A'

    std::cout << "\nMatrix A: \n";
    print_Matrix( B );

    std::cout << "\nDie Matrix Multiplikation ergibt C = A * B : \n";
    print_Matrix( C );

}
```

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

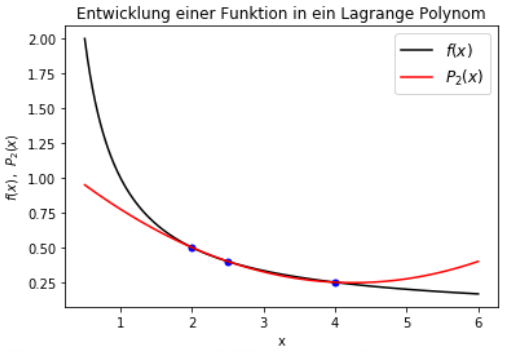
(Introduction to Programming for Physicists)

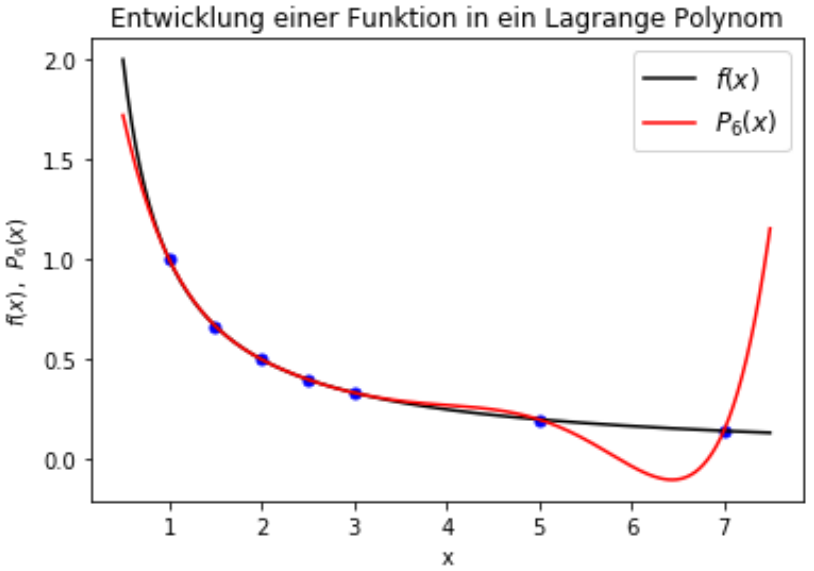
Vorlesung gehalten an der J.W.Goethe-Universität in Frankfurt am Main

(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

Zur Approximation dieser Zahl werden wir nun die Funktion $f(x)$ in ein Lagrange Polynom entwickeln und diesen Ausdruck dann Differenzieren. Wir möchten jetzt die Funktion $f(x)$ mittels $(n+1)$ vorgegebener Punkte $((x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n)))$ in ein Lagrange Polynom vom Grade N entwickeln, wobei wir annehmen, dass $x_0 \in [a, b]$ und $x_n = x_0 + n \cdot h$. Der Wert von h sollte hierbei hinreichend klein sein um sicherzustellen, dass $x_n \in [a, b]$ gilt.

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

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

Wir approximieren nun die Funktion $f(x)$ durch das n-te Lagrange Polynom $P_n(x)$

$$f(x) = P_n(x) + \underbrace{\frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i)}_{\text{Fehler}} \ , \ \forall \ \xi(x) \in [x_0, x_n] \quad ,$$

wobei $f^{(n+1)(x)}$ die $(n+1)$-te Ableitung der Funktion darstellt und $\xi(x)$ ein x-Wert im Bereich der Stützstellen ist.

Vernachlässigt man den Fehlerterm und leitet den Ausdruck ab, erhält man:

$$f'(x) \approx \frac{d}{dx} P_n(x) = \frac{d}{dx} \left(\sum_{k=0}^{n} f(x_k) \cdot L_{n,k}(x) \right) = \sum_{k=0}^{n} f(x_k) \cdot \frac{d}{dx} L_{n,k}(x) = \sum_{k=0}^{n} f(x_k) \cdot L'_{n,k}(x)$$

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.cpp

```cpp
/* Berechnung der Ableitung f'(x) einer Funktion f(x)
 * mittels unterschiedlich genauer Approximationen
 * (Zweipunkteformel, Dreipunkte-Endpunkt-Formel, Dreipunkte-Mittelpunkt-Formel und Fünfpunkte-Mittelpunkt-Formel)  */
#include <stdio.h>                                          // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>                                            // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f(double x){                                         // Deklaration und 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 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
```

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
```

```
 */
#include <stdio.h>                                  // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>                                    // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f(double x){                                 // Deklaration und 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 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 a = 0.5;                                 // Untergrenze des Intervalls [a,b] in dem f(x)=0 gelten soll
    double b = 10;                                  // Obergrenze des Intervalls [a,b] in dem f(x)=0 gelten soll
    int N_xp = 300;                                 // Anzahl der Punkte in die das x-Intervall aufgeteilt wird
    double dx = (b - a)/N_xp;                       // Abstand dx zwischen den aequidistanten Punkten des x-Intervalls
    double x = a - dx;                              // Aktueller x-Wert
    double h_list[3]={0.5, 0.1, 0.01};              // Aequidistanter Abstand zwischen den x-Werten die zur numerischen Differentation benutzt werden
    double h;                                       // Aktueller h-Wert
    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 (3Dreipunkte-Mittelpunkt-Formel)
    double f5_strich_p;                             // Deklaration der f'(x)-Ausgabe-Punkte (Fünfpunkte-Mittelpunkt-Formel)

    printf("# Benutzte h-Werte der Simulation: \n"); // Beschreibung der ausgegebenen Groessen
    for(int k = 0; k < 3; ++k){                     // For-Schleife der Ausgabe der Stuetzstellen x-Werte
        printf("%10.5f",h_list[k]);                 // Ausgabe der unterschiedlichen h-Werte
    }                                               // Ende for-Schleife der Ausgabe
    printf("\n");

    printf("# 0: Index i \n# 1: x-Wert \n# 2: Wirklicher Wert von f'(x) \n");                // Beschreibung der ausgegebenen Groessen
    printf("# h = %8.7f : \n", h_list[0]);                                                  // 0-ter h-Wert
    printf("# 3: Zweipunkteformel f'(x) \n# 4: Dreipunkte-Endpunkt-Formel \n");             // Beschreibung der ausgegebenen Groessen
    printf("# 5: Dreipunkte-Mittelpunkt-Formel  \n# 6: Fünfpunkte-Mittelpunkt-Formel \n"); // Beschreibung der ausgegebenen Groessen
    printf("# h = %8.7f : \n", h_list[1]);                                                  // 1-ter h-Wert
    printf("# 7: Zweipunkteformel f'(x) \n# 8: Dreipunkte-Endpunkt-Formel \n");             // Beschreibung der ausgegebenen Groessen
    printf("# 9: Dreipunkte-Mittelpunkt-Formel  \n# 10: Fünfpunkte-Mittelpunkt-Formel \n"); // Beschreibung der ausgegebenen Groessen
    printf("# h = %8.7f : \n", h_list[2]);                                                  // 2-ter h-Wert
    printf("# 11: Zweipunkteformel f'(x) \n# 12: Dreipunkte-Endpunkt-Formel \n");           // Beschreibung der ausgegebenen Groessen
    printf("# 13: Dreipunkte-Mittelpunkt-Formel  \n# 14: Fünfpunkte-Mittelpunkt-Formel \n"); // Beschreibung der ausgegebenen Groessen

    for(int i =0; i < N_xp; ++i){                                   // for-Schleife die ueber die einzelnen Punkte des x-Intervalls geht
        x=x+dx;                                                     // Aktueller x-Wert
        printf("%3d %14.10f %14.10f",i, x, f_strich(x));            // Ausgaben

        for(int k = 0; k < 3; ++k){                                 // For-Schleife die ueber die unterschiedlichen h-Werte
            h = h_list[k];                                          // Uebergabe des aktuellen h-Wertes
            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("%14.10f %14.10f %14.10f %14.10f", f1_strich_p, f3a_strich_p, f3b_strich_p, f5_strich_p); // Ausgaben der berechneten Groessen
        }                                                           // Ende For-Schleife ueber die unterschiedlichen h-Werte

        printf("\n");                                               // Zeilenumbruch
    }                                                               // Ende for-Schleife ueber die einzelnen Punkte des x-Intervalls
}                                                                   // Ende der Hauptfunktion
```

Nun wollen wir uns die Ableitung der Funktion in einem Teilintervall [a,b] ausgeben lassen und die numerischen Lösungen des C++ Programms für unterschiedliche Werte von h mittels eines Python Skriptes visualisieren. Das folgende C++ Programm berechnet 300 Werte von f'(x) im Teilintervall [0.5,10] und benutzt dabei drei unterschiedliche h-Werte (h=0.5,0.1,0.01).

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V6$ g++ Numerical_Differentiation_2.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V6$ ./a.out
# Benutzte h-Werte der Simulation:
   0.50000   0.10000   0.01000
# 0: Index i
# 1: x-Wert
# 2: Wirklicher Wert von f'(x)
# h = 0.5000000 :
# 3: Zweipunkteformel f'(x)
# 4: Dreipunkte-Endpunkt-Formel
# 5: Dreipunkte-Mittelpunkt-Formel
# 6: Fünfpunkte-Mittelpunkt-Formel
# h = 0.1000000 :
# 7: Zweipunkteformel f'(x)
# 8: Dreipunkte-Endpunkt-Formel
# 9: Dreipunkte-Mittelpunkt-Formel
# 10: Fünfpunkte-Mittelpunkt-Formel
# h = 0.0100000 :
# 11: Zweipunkteformel f'(x)
# 12: Dreipunkte-Endpunkt-Formel
# 13: Dreipunkte-Mittelpunkt-Formel
# 14: Fünfpunkte-Mittelpunkt-Formel
  0   0.5000000000   7.8917798738  6.1070150814   8.1889480404   7.6139443325   7.8809956049  7.5716576421   7.9122697304  7.8805612448   7.8917622795  7.8607459746   7.8920024300  7.8916676439   7.8917798721
  1   0.5316666667   7.6926047741  5.8663159930   7.9665395217   7.4247523236   7.6823813066  7.3624240284   7.7122470363  7.6817909925   7.6925881055  7.6605279178   7.6928191883  7.6924965950   7.6926047724
  2   0.5633333333   7.4869212930  5.6212143523   7.7374927591   7.2292593093   7.4772654071  7.1470788599   7.5057014827  7.4765207196   7.4869055610  7.4538404463   7.4871274002  7.4868172483   7.4869212914
  3   0.5950000000   7.2749781616  5.3719853890   7.5020808892   7.0277024239   7.2658960218  6.9258781161   7.2928827624  7.2649986873   7.2749633762  7.2409330588   7.2751758062  7.2748783302   7.2749781601
  4   0.6266666667   7.0570288599  5.1189067619   7.2605817500   6.8203234262   7.0485260125  6.6990820717   7.0740453245  7.0474779006   7.0570150300  7.0220599578   7.0572178957  7.0569333160   7.0570288585
  5   0.6583333333   6.8333313308  4.8622582629   7.0132775703   6.6073684264   6.8254127025  6.4669550067   6.8494480880  6.8242158231   6.8333184645  6.7974797636   6.8335116214  6.8332401439   6.8333313295
  6   0.6900000000   6.6041476920  4.6023215216   6.7604546579   6.3890876098   6.5968175883  6.2297649145   6.6193541516  6.5954740890   6.6041357960  6.5674552250   6.6043191105  6.6040609265   6.6041476908
```

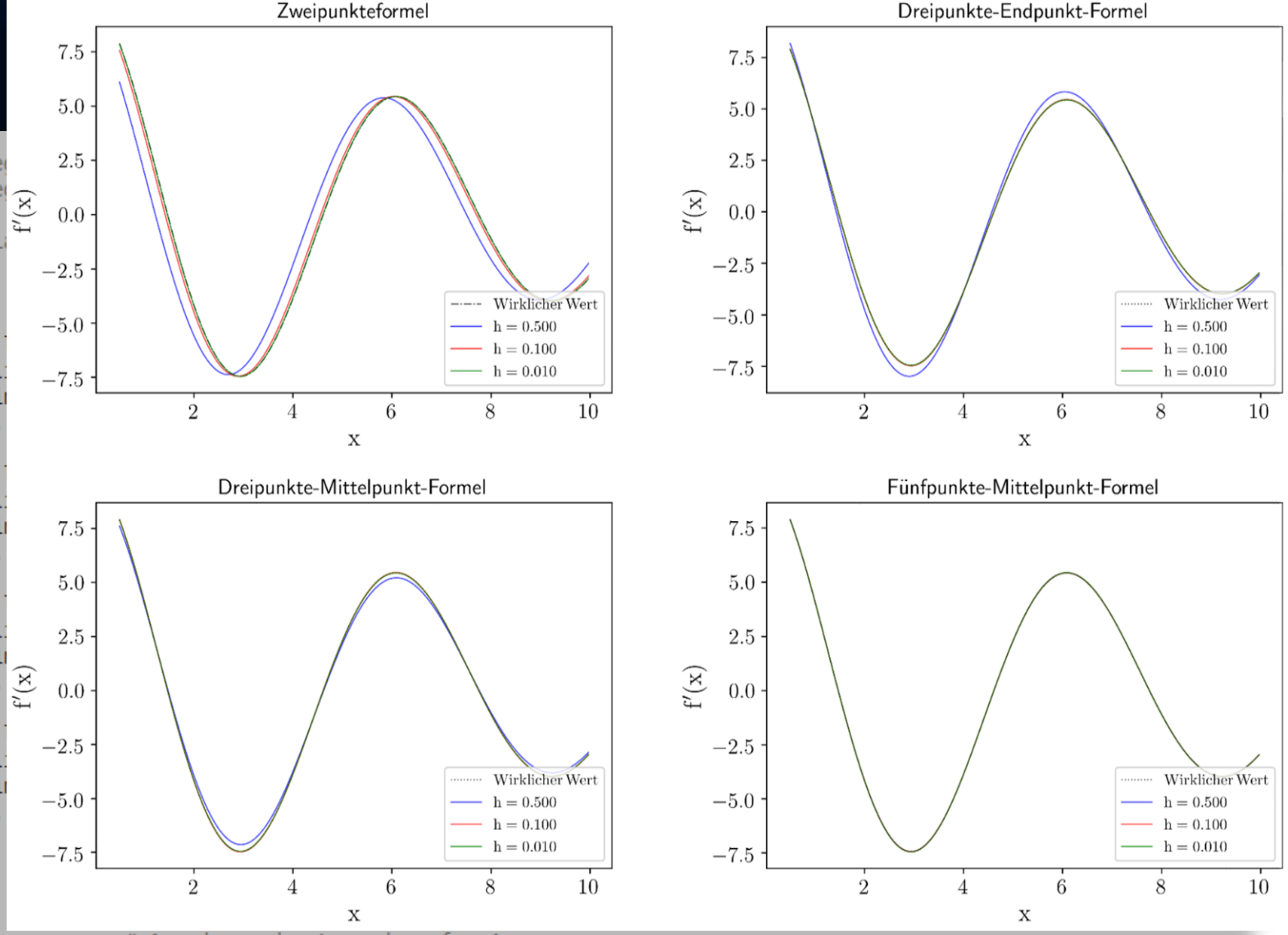
```
    printf("# h = %8.7f : \n", h_list[2]);                                    // 2-ter h-Wert
    printf("# 11: Zweipunkteformel f'(x) \n# 12: Dreipunkte-Endpunkt-Formel \n");          // Beschreibung der ausgegebenen Groessen
    printf("# 13: Dreipunkte-Mittelpunkt-Formel  \n# 14: Fünfpunkte-Mittelpunkt-Formel \n"); // Beschreibung der ausgegebenen Groessen

    for(int i =0; i < N_xp; ++i){                                            // for-Schleife die ueber die einzelnen Punkte des x-Intervalls geht
        x=x+dx;                                                              // Aktueller x-Wert
        printf("%3d %14.10f %14.10f",i, x, f_strich(x));                     // Ausgaben

        for(int k = 0; k < 3; ++k){                                          // For-Schleife die ueber die unterschiedlichen h-Werte
            h = h_list[k];                                                   // Uebergabe des aktuellen h-Wertes
            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("%14.10f %14.10f %14.10f %14.10f", f1_strich_p, f3a_strich_p, f3b_strich_p, f5_strich_p); // Ausgaben der berechneten Groessen
        }                                                                    // Ende For-Schleife ueber die unterschiedlichen h-Werte

        printf("\n");                                                        // Zeilenumbruch
    }                                                                        // Ende for-Schleife ueber die einzelnen Punkte des x-Intervalls
}                                                                            // Ende der Hauptfunktion
```

PythonPlot_Numerical_Differentiation_2.py

```
# Python Programm zum Plotten der berechneten Daten von Numerical_Differentiation_2.cpp
import matplotlib                                                   # Python Bibliothek zum Plotten (siehe https://matplotlib.org/ )
import matplotlib.pyplot as plt                                     # Python Bibliothek zum Plotten (siehe https://matplotlib.org/ )
import numpy as np                                                  # Python Bibliothek fuer Mathematisches (siehe https://numpy.org/ )
import matplotlib.gridspec as gridspec                              # Mehrere Bilder in einem Plot anordnen

# Bildabmessungen usw.
params = {
    'figure.figsize'    : [14,10],
    'text.usetex'       : True,
    'axes.titlesize' : 14,
    'axes.labelsize' : 16,
    'xtick.labelsize' : 14 ,
    'ytick.labelsize' : 14
}
matplotlib.rcParams.update(params)

h_list = np.genfromtxt("./Numerical_Differentiation_2.dat", max_rows=1)   # Einlesen der benutzten x-Werte der Stuetzstellen
data = np.genfromtxt("./Numerical_Differentiation_2.dat", skip_header=20) # Einlesen der berechneten Daten von Numerical_Differentiation_2.cpp

fig = plt.figure()                                                  # Hauptbild
gs = gridspec.GridSpec(2, 2, width_ratios=[1,1], wspace=0.3, hspace=0.3)  # Anordnung der vier Unterbilder
ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])

ax1.set_title(r'Zweipunkteformel')                      # Titel der Abbildung in ax1
ax1.set_xlabel(r"$\rm x$")
ax1.set_ylabel(r"$\rm f'(x)$")
ax2.set_title(r'Dreipunkte-Endpunkt-Formel')            # Titel der Abbildung in ax2
ax2.set_xlabel(r"$\rm x$")
ax2.set_ylabel(r"$\rm f'(x)$")
ax3.set_title(r'Dreipunkte-Mittelpunkt-Formel')         # Titel der Abbildung in ax3
ax3.set_xlabel(r"$\rm x$")
ax3.set_ylabel(r"$\rm f'(x)$")
ax4.set_title(r'Fünfpunkte-Mittelpunkt-Formel')         # Titel der Abbildung in ax4
ax4.set_xlabel(r"$\rm x$")
ax4.set_ylabel(r"$\rm f'(x)$")
```

```
l_width=0.8                                         # Festlegung der Plot-Liniendicke
alp=0.7                                             # Festlegung der Transparenz der Kurven

label_0=r'$\rm h='+'{0:.3f}'.format(h_list[0])+'$'      # Plot-Labels fuer die unterschiedlichen h-Werte
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, linestyle='-.', alpha=alp, label=r'$\rm Wirklicher \, Wert$') # Plotten auf ax1
ax1.plot(data[:,1],data[:,3], color="blue", linewidth=l_width, linestyle='-', alpha=alp, label=label_0)
ax1.plot(data[:,1],data[:,7], color="red", linewidth=l_width, linestyle='-', alpha=alp, label=label_1)
ax1.plot(data[:,1],data[:,11], color="green", linewidth=l_width, linestyle='-', alpha=alp, label=label_2)

ax2.plot(data[:,1],data[:,2], color="black", linewidth=l_width, linestyle=':', alpha=alp, label=r'$\rm Wirklicher \, Wert$')  # Plotten auf ax2
ax2.plot(data[:,1],data[:,4], color="blue", linewidth=l_width, linestyle='-', alpha=alp, label=label_0)
ax2.plot(data[:,1],data[:,8], color="red", linewidth=l_width, linestyle='-', alpha=alp, label=label_1)
ax2.plot(data[:,1],data[:,12], color="green", linewidth=l_width, linestyle='-', alpha=alp, label=label_2)

ax3.plot(data[:,1],data[:,2], color="black", linewidth=l_width, linestyle=':', alpha=alp, label=r'$\rm Wirklicher \, Wert$')  # Plotten auf ax3
ax3.plot(data[:,1],data[:,5], color="blue", linewidth=l_width, linestyle='-', alpha=alp, label=label_0)
ax3.plot(data[:,1],data[:,8], color="red", linewidth=l_width, linestyle='-', alpha=alp, label=label_1)
ax3.plot(data[:,1],data[:,12], color="green", linewidth=l_width, linestyle='-', alpha=alp, label=label_2)

ax4.plot(data[:,1],data[:,2], color="black", linewidth=l_width, linestyle=':', alpha=alp, label=r'$\rm Wirklicher \, Wert$')  # Plotten auf ax4
ax4.plot(data[:,1],data[:,6], color="blue", linewidth=l_width, linestyle='-', alpha=alp, label=label_0)
ax4.plot(data[:,1],data[:,9], color="red", linewidth=l_width, linestyle='-', alpha=alp, label=label_1)
ax4.plot(data[:,1],data[:,13], color="green", linewidth=l_width, linestyle='-', alpha=alp, label=label_2)

ax1.legend(frameon=True, loc="lower right",fontsize=10)                 # Anordnung der Legende auf ax1
ax2.legend(frameon=True, loc="lower right",fontsize=10)                 # Anordnung der Legende auf ax2
ax3.legend(frameon=True, loc="lower right",fontsize=10)                 # Anordnung der Legende auf ax3
ax4.legend(frameon=True, loc="lower right",fontsize=10)                 # Anordnung der Legende auf ax4

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
```

```
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.