Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
07.06.2022

MATTHIAS HANAUSKE

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

8. Vorlesung

Plan für die heutige Vorlesung

- Kurze Wiederholung der Vorlesung 7
- C++ Container und die vector Klasse der Standardbibliothek
- Differentialgleichungen: Numerische Lösung von Anfangswertproblemen
- Übungsaufgaben: Übungsblatt Nr.9

Wiederholung der Vorlesung 7

- Objekt-orientierte Programmierung und C++ Klassen
- Theorie: Numerische Integration
- Anwendungsbeispiel: Numerische Integration

Benutzerdefinierte Typen und Abstraktionsmechanismen in C++

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

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

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

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

Konstruktoren und Destruktoren

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

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

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

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

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

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

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

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

Ein einfaches Beispiel für eine konkrete Klasse

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

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

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

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

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

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

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

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

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

Die Klasse „Ding“

Das folgende Programm zeigt die Implementierung der Klasse im Programm:

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

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

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

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

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

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

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

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

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

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

Die Klasse „Lagrangepoly“

Lagrange_Polynom_Klasse.cpp

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

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

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

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

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

Anwendung der Klasse „Lagrangepoly“ im Hauptprogramm

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

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

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

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

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

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

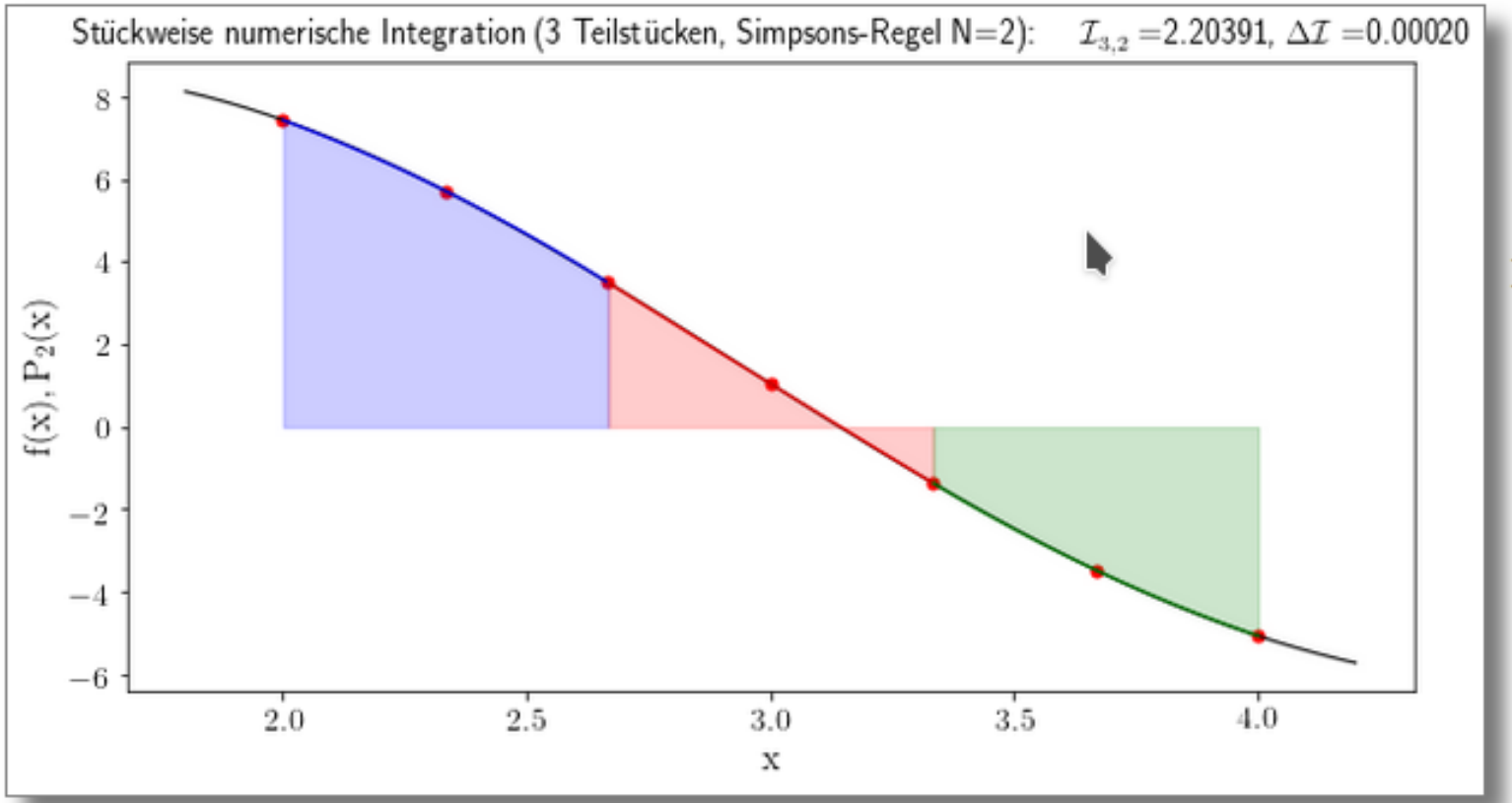
Lagrange_Polynom_Klasse_a.cpp

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

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

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

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

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

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

Die Klasse „Lagrangepoly“

Ausgelagerte „inline“-Funktion Der Klasse

Theorie: Numerische Integration



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

Zusammenfassung
Integrationsregeln der numerischen Mathematik

Anwendungsbeispiel: Numerische Integration

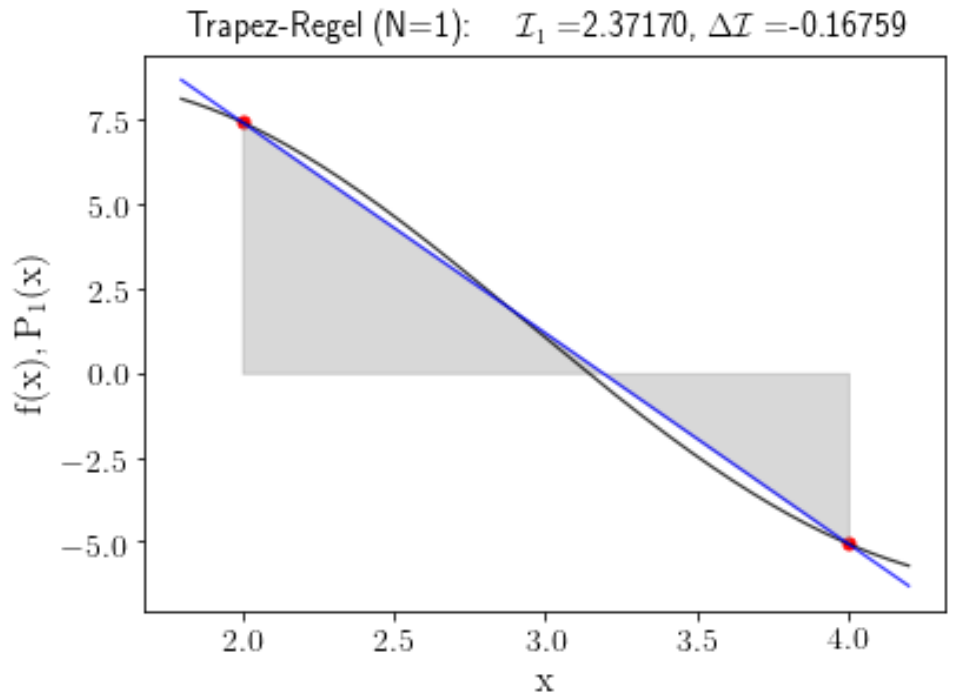
Die im vorigen Unterpunkt hergeleitete numerische Integrationsregeln werden nun in einem C++ Programm benutzt, um den Wert des bestimmten Integrals einer Funktion $\int_a^b f(x)\,dx$ approximativ zu bestimmen. Die Integrationsregeln lauteten:

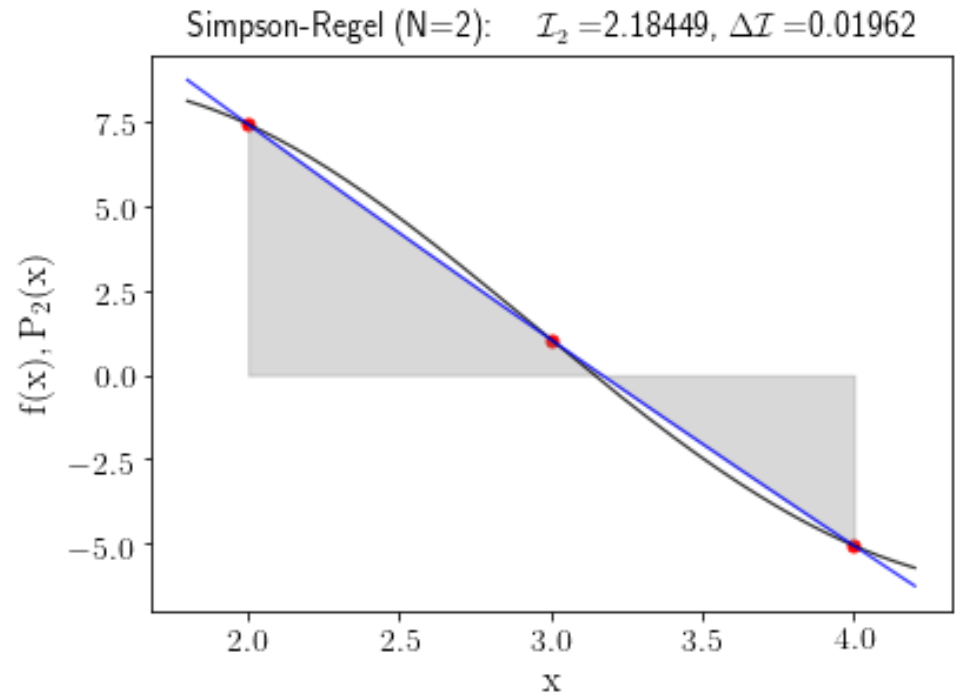
Die Trapez-Regel (N=1): $$\int_a^b f(x)\,dx \approx \frac{(x_1 - x_0)}{2} \cdot (f(x_0) + f(x_1)) = \frac{h}{2} \cdot (f(x_0) + f(x_1))$$, Stützstellen: $(x_0 = a, x_0 + h = b)$

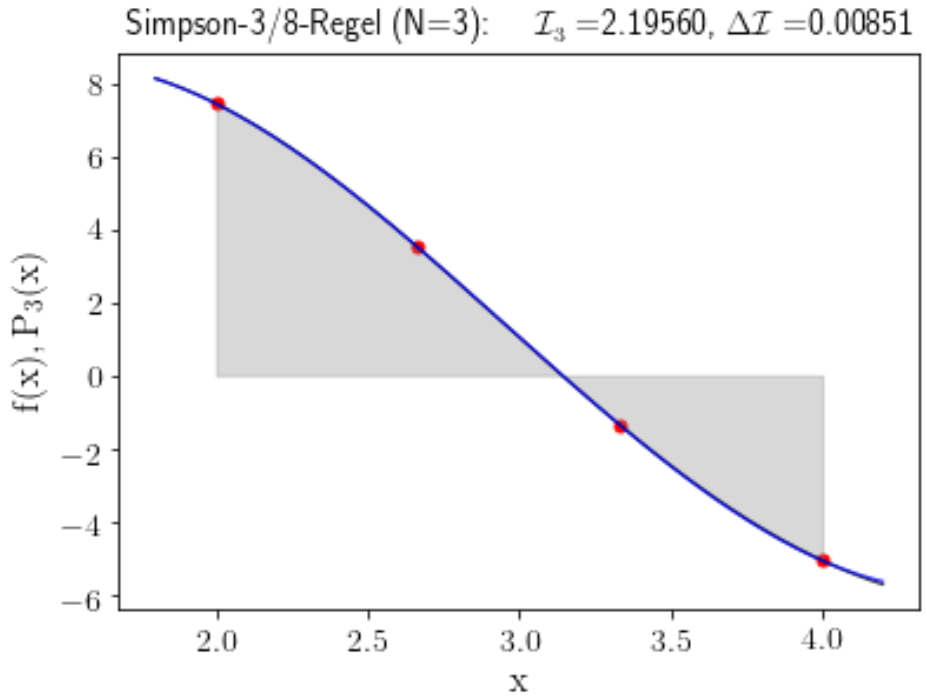
Die Simpson's-Regel (N=2): $$\int_a^b f(x)\,dx \approx \frac{h}{3} \cdot (f(x_0) + 4f(x_1) + f(x_2))$$, Stützstellen: $(x_0 = a, x_0 + h, x_0 + 2h = b)$

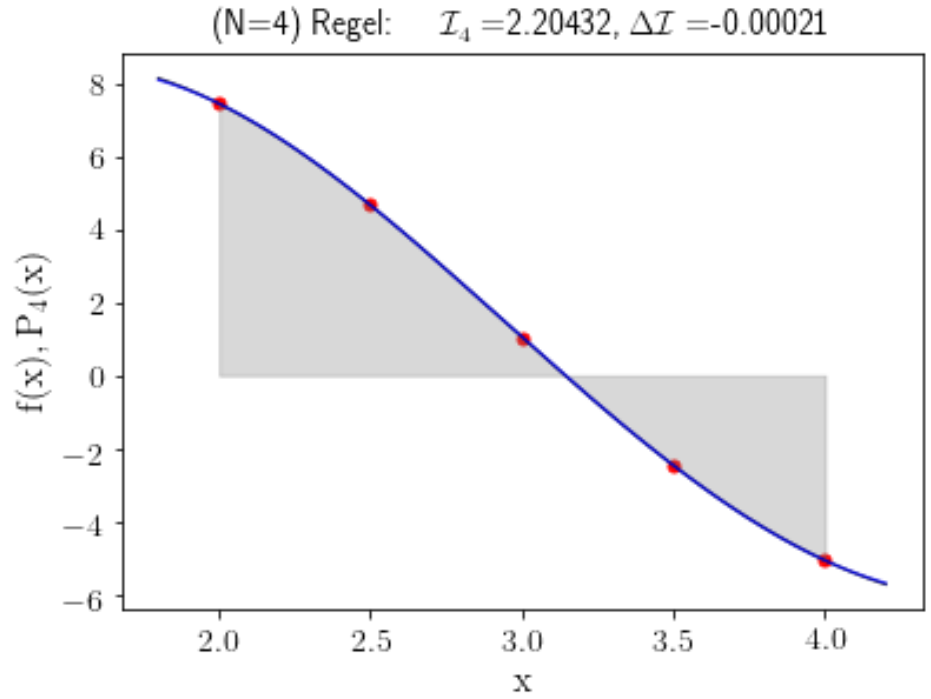
Die Simpson's-3/8-Regel (N=3): $$\int_a^b f(x)\,dx \approx \frac{3\,h}{8} \cdot (f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3))$$, Stützstellen: $(x_0 = a, x_0 + h, x_0 + 2h, x_0 + 3h = b)$

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


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


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


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


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

Stückweise numerische Integration

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

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

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

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

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

Übungsblatt Nr. 8

Aufgabe 1 *(10 Punkte)*

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

Übungsblatt Nr.8

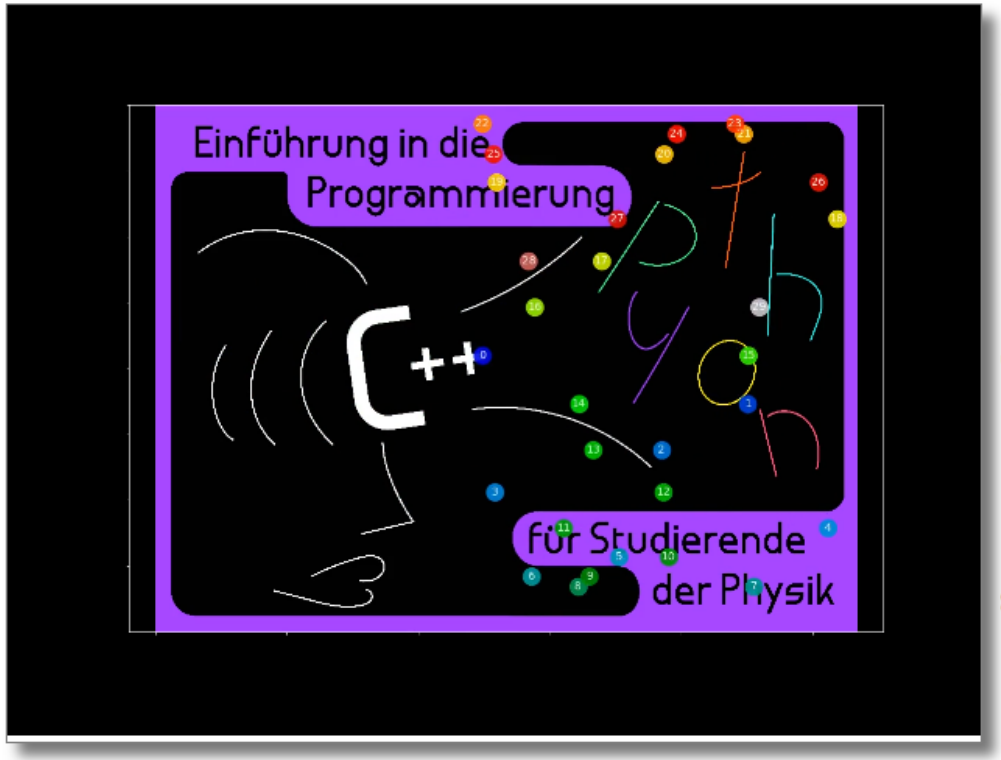
Aufgabe 2 *(10 Punkte)*

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

Vorlesung 8

Die Container der Standardbibliothek, insbesondere der sequentielle Container <vector>, sind ein wichtiges Abstraktionskonstrukt, das man einfach in seinen eigenen Programmen verwenden kann. In dieser Vorlesung werden wir im ersten Unterpunkt die Klasse <vector>, kennenlernen und auf unterschiedliche Beispiele anwenden. Der zweite Unterpunkt befasst sich dann mit dem numerischen Lösen von Differentialgleichungen erster Ordnung.

C++ Container und die vector Klasse der Standardbibliothek



Die *C++ Standardbibliothek* verfügt über eine Vielzahl nützlicher Programmierkonstrukte und ein oft verwendetes Klassenkonzept sind die sogenannten *C++ Container*. Ein *Container* ist ein Objekt, das eine Sammlung von Elementen aufnimmt. Die verfügbaren *STL-Container* gliedern sich in *sequentielle Container* (wie z.B. '<vector>' und '<list>') und ungeordnete/geordnete *assoziative Container* (wie z.B. '<map>' und '<unordered_map>'). In diesem Unterpunkt werden wir uns mit dem Container <vector> näher befassen. Der STL-Container <vector> stellt einen sequenziellen Typ von Objekten dar und ist somit eine Sequenz von Elementen eines bestimmten Typs. Man kann sich die Struktur eines vector-Objektes als ein eindimensionales Array vorstellen, bei welchem man zusätzlich noch die Anzahl der Elemente im Programmverlauf verändern kann. Außerdem stellt die vector-Klasse mehrere Memberfunktionen bereit, die einem bei der Konstruktion des Vektors helfen. Wir gehen zunächst auf die Klassenstruktur des standard Containers <vector> ein und verdeutlichen das Konzept des Vektors anhand von Integer Vektoren. Man kann die Klasse <vector> jedoch auch als ein Container von Objekten verwenden. Dies verdeutlichen wir anhand einer Simulation von nicht-wechselwirkenden Ding-Objekten (siehe nebenstehende Abbildung; näheres siehe C++ Container und die vector Klasse der Standardbibliothek).

Vorlesung 8

In der vorigen Vorlesung hatten wir das Klassenkonzept kennengelernt und eine Beispielklasse 'Ding' erstellt. Hierbei wurden die messbaren Eigenschaften des Dings (Daten-Member der Klasse) von den Verhaltensweisen des Dings (Member-Funktionen der Klasse) getrennt in der Klasse implementiert. Wie programmiert man die Verhaltensweisen eines Objektes? Das Verhalten eines Objektes unter einem einwirkenden Einfluss stellt eine Art von zeitlichem Verlauf dar. In Abhängigkeit von der jeweiligen Fragestellung und Natur des physikalischen Problems sind zwei generelle Programmzweige von zeitlich veränderlichen Systemen zu identifizieren: Die *Agenten-basierte Simulation* und *Simulationen von physikalischen Bewegungsgleichungen*. Im ersten Teil dieser Vorlesung (siehe C++ Container und die vector Klasse der Standardbibliothek) werden wir eine Art von Agenten-basierte Simulation kennenlernen. Die 'Agenten' stellen in diesem Fall die Teilchen in einer Kiste dar, die als Objekte der Klasse 'Ding' erzeugt wurden. Die zeitliche Entwicklung wird hierbei über eine Member-Funktion realisiert.
Im zweiten Teil werden wir dann sehen, wie man eine zeitliche Entwicklung eines Systems unter Verwendung seiner Bewegungsgleichungen simuliert (näheres siehe Differentialgleichungen: Numerische Lösung von Anfangswertproblemen). In dieser und der folgenden Vorlesung werden wir uns den Themenbereich des numerischen Lösens von Differentialgleichungen näher betrachten und mehrere Verfahren zum Lösen von Differentialgleichungen erster Ordnung kennenlernen. Die einzelnen Verfahren werden dann in einem C++ Programm implementiert und miteinander verglichen. Zusätzlich wird in einem Jupyter Notebook gezeigt, wie man mittels Python auch numerisch eine Differentialgleichung lösen kann.

Wie programmiert man die Verhaltensweisen eines Objektes?

Agenten-basierte Simulation vs. Simulationen von Bewegungsgleichungen

In der vorigen Vorlesung hatten wir das Klassenkonzept kennengelernt und eine Beispielklasse 'Ding' erstellt. Hierbei wurden die messbaren Eigenschaften des Dings (Daten-Member der Klasse) von den Verhaltensweisen des Dings (Member-Funktionen der Klasse) getrennt in der Klasse implementiert. Wie programmiert man die Verhaltensweisen eines Objektes? Das Verhalten eines Objektes unter einem einwirkenden Einfluss stellt eine Art von zeitlichem Verlauf dar. In Abhängigkeit von der jeweiligen Fragestellung und Natur des physikalischen Problems sind zwei generelle Programmzweige von zeitlich veränderlichen Systemen zu identifizieren: Die *Agenten-basierte Simulation* und *Simulationen von physikalischen Bewegungsgleichungen*.

Im ersten Teil dieser Vorlesung (siehe C++ Container und die vector Klasse der Standardbibliothek) werden wir eine Art von Agenten-basierte Simulation kennenlernen. Die 'Agenten' stellen in diesem Fall die Teilchen in einer Kiste dar, die als Objekte der Klasse 'Ding' erzeugt wurden. Die zeitliche Entwicklung wird hierbei über eine Member-Funktion realisiert.

Im zweiten Teil werden wir dann sehen, wie man eine zeitliche Entwicklung eines Systems unter Verwendung seiner Bewegungsgleichungen simuliert (näheres siehe Differentialgleichungen: Numerische Lösung von Anfangswertproblemen). In dieser und der folgenden Vorlesung werden wir uns den Themenbereich des numerischen Lösens von Differentialgleichungen näher betrachten und mehrere Verfahren zum Lösen von Differentialgleichungen kennenlernen.

C++ Container und die vector Klasse der Standardbibliothek

Die *C++ Standardbibliothek* verfügt über eine Vielzahl nützlicher Programmierkonstrukte und ein oft verwendetes Klassenkonzept sind die sogenannten *C++ Container*. Ein *Container* ist ein Objekt, das eine Sammlung von Elementen aufnimmt. Die verfügbaren *STL-Container* gliedern sich in *sequentielle Container* (wie z.B. '<vector>' und '<list>') und ungeordnete/geordnete *assoziative Container* (wie z.B. '<map>' und '<unordered_map>'). In diesem Unterpunkt werden wir uns mit dem Container <vector> näher befassen. Der STL-Container <vector> stellt einen sequenziellen Typ von Objekten dar und ist somit eine Sequenz von Elementen eines bestimmten Typs. Man kann sich die Struktur eines vector-Objektes als ein eindimensionales Array vorstellen, bei welchem man zusätzlich noch die Anzahl der Elemente im Programmverlauf verändern kann. Außerdem stellt die vector-Klasse mehrere Memberfunktionen bereit, die einem bei der Konstruktion des Vektors helfen. Wir gehen zunächst auf die Klassenstruktur des standard Containers <vector> ein und verdeutlichen das Konzept des Vektors anhand von Integer Vektoren. Man kann die Klasse <vector> jedoch auch als ein Container von Objekten verwenden. Dies verdeutlichen wir anhand einer Simulation von nicht-wechselwirkenden Ding-Objekten.

Der STL-Container <vector> der C++ Standardbibliothek stellt einen sequenziellen Typ von Objekten dar und ist somit eine Sequenz von Elementen eines bestimmten Typs. Bei der Definition eines vector-Objektes werden die einzelnen Elemente des Vektors, im Hauptspeicher aufeinanderfolgend abgelegt. Die Vektorklasse ist als eine *Template*-Klasse formuliert, was bedeutet, dass der Typ T der Objekte veränderbar ist, die einzelnen Objekte jedoch von gleichem Typ sein müssen. Man erzeugt ein vector-Objekt, indem man einen der vector-Konstruktoren im Hauptprogramm aufruft (z.B. ' vector<T> v;').

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

// Beispiel fuer die Struktur der vector-Klasse der Standardbibliothek
class Vektor {
    // Private Instanzvariablen (Daten-Member)
private:
    double* elem; // Zeiger auf das eindimensionale Array von 'anz' double Elementen
    int anz;      // Anzahl der Elemente des vector-Objektes

// Oeffentliche Bereiche der Klasse
public:
    // Ueberladene Konstruktoren der Klasse Vektor
    // Konstruktor mit einem Argument
    Vektor(int set_anz) : elem { new double[set_anz] }, anz{set_anz} {
        printf("Konstruktor(anz) erzeugt einen Vektor-Container mit %i Elementen \n", set_anz);
        for(int i = 0; i!=set_anz; ++i){
            elem[i] = 0;
        }
    }
    //...
    // Destruktor der Klasse Vektor
    ~Vektor() {
        printf("Destruktor loescht den Vektor-Container \n");
        delete[] elem;
    }

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

int main(){               // Hauptfunktion
    Vektor w = Vektor(3); // Deklaration eines Objektes der Vektor-Klasse mit drei Elementen
}
```

Die Klassenstruktur des standard Containers <vector>

```
(base) hanauske@hanauske-Aspire-A717-72G:~/www/Versuch1/VPROG/C++$ g++ Vector_0.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/www/Versuch1/VPROG/C++$ ./a.out
Konstruktor(anz) erzeugt einen Vektor-Container mit 3 Elementen
Destruktor loescht den Vektor-Container
```

Die Klasse <vector> am Beispiel eines Integer Vektors

Vector_1.cpp

```cpp
#include <iostream>                          // Ein- und Ausgabebibliothek
#include <vector>                            // Sequenzieller Container vector<Type> der Standardbibliothek
using namespace std;                         // Benutze den Namensraum std

int main(){                                  // Hauptfunktion
    vector<int> w = {1,4,6,8,9,5,3};         // Deklaration und Initialisierung eines Integer-vector-Containers mit sieben Einträgen
    vector <int>::iterator Iter;             // nur fuer Schleifen mit ::iterator noetig

    w.push_back(10);                         // Einfuegen eines neuen Elementes am Ende des Vektors
    printf("w = (");
    for (auto& n : w){                       // Bereichsbasierte for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", n);
    }
    printf(") \n \n");

    w.insert(w.begin()+3,99);                            // Einfuegen eines neuen Elementes an der vierten Position des Vektors

    printf("w = (");
    for ( Iter = w.begin() ; Iter != w.end() ; Iter++ ){ // for-Schleife zum Ausgeben der einzelnen Elemente des Vektors (mittels ::iterator)
        printf("%3i ", *Iter);
    }
    printf(") \n \n");

    w[5] = 77;                                           // Neue Wert-Zuweisung fuer das 6. Elementes des Vektors

    printf("w = (");
    for(int i=0; i<w.size(); ++i){                       // Normale for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", w[i]);
    }
    printf(") \n");
}
```

Vector_1.cpp

```cpp
#include <iostream>                         // Ein- und Ausgabebibliothek
#include <vector>                           // Sequenzieller Container vector<Type> der Standardbibliothek
using namespace std;                        // Benutze den Namensraum std

int main(){                                 // Hauptfunktion
    vector<int> w = {1,4,6,8,9,5,3};        // Deklaration und Initialisierung eines Integer-vector-Containers mit sieben Einträgen
    vector <int>::iterator Iter;            // nur fuer Schleifen mit ::iterator noetig

    w.push_back(10);                        // Einfuegen eines neuen Elementes am Ende des Vektors
    printf("w = (");
    for (auto& n : w){                      // Bereichsbasierte for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", n);
    }
    printf(") \n \n");

    w.insert(w.begin()+3,99);                           // Einfuegen eines neuen Elementes an der vierten Position des Vektors

    printf("w = (");
    for ( Iter = w.begin() ; Iter != w.end() ; Iter++ ){ // for-Schleife zum Ausgeben der einzelnen Elemente des Vektors (mittels ::iterator)
        printf("%3i ", *Iter);
    }
    printf(") \n \n");

    w[5] = 77;                                          // Neue Wert-Zuweisung fuer das 6. Elementes des Vektors

    printf("w = (");
    for(int i=0; i<w.size(); ++i){                      // Normale for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", w[i]);
    }
    printf(") \n");
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Container$ g++ Vector_1.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Container$ ./a.out
w = (   1   4   6   8   9   5   3  10 )

w = (   1   4   6  99   8   9   5   3  10 )

w = (   1   4   6  99   8  77   5   3  10 )
```

Öffentliche Methoden (Member Funktionen) der Klasse <vector>

Die untere Tabelle listet einige der verfügbaren öffentlichen Methoden der Klasse <vector> auf:

Anweisung	Bedeutung
v.push_back(val);	Fügt die Daten aus val an das Ende des Vektors v an.
v.pop_back();	Entfernt das letzte Element des Vektors v.
v.insert(pos,val);	Fügt die Daten aus val an die Position pos des Vektors v ein.
v.size();	Gibt die Anzahl aller Elemente im Vektors v zurück.
v.resize(n);	Setzt die Anzahl der Elemente im Vektors auf n.
v.clear();.	Entfernt alle Elemente des Vektors v.
v.front();	Gibt die Referenz auf das erste Element von v zurück.
v.back();	Gibt die Referenz auf das letzte Element von v zurück.
v.capacity();	Die Anzahl der Elemente die in v gespeichert werden können.
v.at(n);	Repräsentiert das n. Element des Vektors v (prüft zuvor, ob n im erlaubten Bereich liegt).
v[n];	Repräsentiert das n. Element des Vektors v (prüft nicht, ob n im erlaubten Bereich liegt).

Die Methode 'v.resize(n);' wird in dem unteren Quelltext des C++ Programms Vector_2.cpp benutzt. In dem Programm wurde zunächst ein Integer-Vektor w mit fünf Elementen deklariert (vector<int> w(5);). Die fünf Elemente des Vektors erhalten dann, innerhalb einer for-Schleife, ihre entsprechenden Werte und der Vektor wird im Terminal ausgegeben. Dann wird der Vektor mittels 'w.resize(7);' auf eine Kapazität von sieben erhöht und die Wertzuweisungen an die neuen Elemente gemacht.

Vector_2.cpp

```cpp
#include <iostream>                    // Ein- und Ausgabebibliothek
#include <vector>                      // Sequenzieller Container vector<Type> der Standardbibliothek
using namespace std;                   // Benutze den Namensraum std

int main(){                            // Hauptfunktion
    vector<int> w(5);                  // Deklaration eines Integer-vector-Containers mit fuenf Einträgen

    for(int i = 0; i < w.size(); ++i){ // Normale for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        w[i] = i*i;                    // Wert-Zuweisung an das t-te Elementes des Vektors
    }

    printf("w = (");
    for (auto& n : w){                 // Bereichsbasierte for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", n);
    }
    printf(") \n");

    w.resize(7);                       // Die Anzahl der Eintraege im Vektor w wird auf 7 erhoeht
    w[5] = 5*5;                        // Wert-Zuweisung an das 5-te Elementes des Vektors
    w[6] = 6*6;                        // Wert-Zuweisung an das 6-te Elementes des Vektors

    printf("w = (");
    for (auto& n : w){                 // Bereichsbasierte for-Schleife zum Ausgeben der einzelnen Elemente des Vektors
        printf("%3i ", n);
    }
    printf(") \n");
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Container$ g++ Vector_2.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/Container$ ./a.out
w = (   0   1   4   9  16 )
w = (   0   1   4   9  16  25  36 )
```

Die Klasse <vector> als ein Container von Objekten

Wie wir im vorigen Unterpunkt gesehen haben, kann man die Klasse <vector> benutzen, um eindimensionale Arrays in Form von Standardvektoren darzustellen. Die Verwendung von einem vector-Objekt anstelle von einem integrierten eindimensionalen Array hat den Vorteil, dass man die Kapazität des Vektors im Laufe des Programms einfach verändern kann. In diesem Unterpunkt wollen wir einen weiteren Anwendungsfall der Klasse <vector> betrachten, bei dem die Elemente des Containers selbst Objekte von Klassen sind. Betrachten wir z.B. die in der vorigen Vorlesung entworfene Klasse 'Ding' und stellen uns vor, dass wir eine bestimmte Anzahl von Dingen in eine Kiste einsperren möchten. Wir möchten somit einen Container, bestehend aus Objekten der Klasse 'Ding' erstellen.

Wir möchten im Folgenden eine gewisse Anzahl von Teilchen (z.B. unsigned int Anz_Teilchen = 10;) in eine zweidimensionale Kiste einsperren (Abmessung der Kiste z.B. double kiste_x = 40; und double kiste_y = 40;). Die 10 Teilchen sollen im Hauptprogramm als Instanzen der Klasse Ding erzeugt werden, wobei die Klasse Ding eine, ein wenig allgemeinere Struktur hat, als die in der vorigen Vorlesung vorgestellte Ding-Klasse. Die Klasse Ding sollte neben den Ortskoordinaten (double x = 0, y = 0, z = 0;) auch die Teilchengeschwindigkeiten als private Daten-Member enthalten (z.B. zunächst initialisiert auf Null: double v_x = 0, v_y = 0, v_z = 0;). Mittels des Konstruktors kann der Benutzer dann die Anfangsorte und Anfangsgeschwindigkeiten der einzelnen Teilchen festlegen.

Nun wird zusätzlich noch das zeitliche Verhalten der Dinge als eine inline Funktion definiert. Die Teilchen sollen hierbei zunächst nicht miteinander wechselwirken. Die Begrenzungen der Kiste bilden jedoch nicht überwindbare Barrieren für die klassischen Teilchen und es soll eine vollständige Reflexion an den Kistenbegrenzungen stattfinden. Wir implementieren dieses zeitliche Verhalten der Teilchen als eine inline-Methode der Klasse 'Ding' und benennen sie 'Gehe_Zeitschritt': **inline** void Ding::Gehe_Zeitschritt(double dt, double max_x, double max_y, double max_z){ ... } .
Die Funktion beschreibt die Veränderung der Ortskoordinaten (x, y, z) bei einem Zeitschritt dt, wobei die speziellen Randbedingungen an die Bewegung der Teilchen (Reflexion an den Rändern der Kiste) mittels der Argumente double max_x, double max_y, double max_z spezifiziert werden.

Im Hauptprogramm sollen dann ein vector-Container mittels 'vector<Ding> Kiste_Teilchen;' deklariert werden und dieser wird dann mittels 'Kiste_Teilchen.push_back(Ding {...});' aufgefüllt. Nach diesem Initialisierungsprozess wird die zeitliche Entwicklung der Teilchen für z.B. 100 Zeitschritte (double Anz_tSchritte = 100; mit double dt = 0.05;) im Terminal ausgegeben. Das folgende C++ Programm stellt eine Realisierung des vector-Containers bestehend aus 10 Dingen mit zeitlicher Entwicklung der Ortrskoordinaten dar.

Die Klasse <vector> als ein Container von Objekten

```
for (unsigned int n = 0; n < Anz_Teilchen; ++n){
    Kiste_Teilchen.push_back( Ding {n, 1, (n+1)*kiste_x/Anz_Teilchen, 0, (n+1.0), 0, 0} );
}
```

Wie wir im v … m von Standardvektoren darzuste … Vorteil, dass man die Kapazität de… …wendungsfall der Klasse <vector> … ler vorigen Vorlesung entworfene K… Wir möchten somit einen

Wir möchten im Folgenden eine gewisse Anzahl von Teilchen (z.B. unsigned int Anz_Teilchen = 10;) in eine zweidimensionale Kiste einsperren (Abmessung der Kiste z.B. double kiste_x = 40; und double kiste_y = 40;). Die 10 Teilchen sollen im Hauptprogramm als Instanzen der Klasse Ding erzeugt werden, wobei die Klasse Ding eine, ein wenig allgemeinere Struktur hat als die in der vorigen Vorlesung vorgestellte Ding-Klasse. Die Klasse Ding sollte neben den Ortskoordinaten (double x = 0, y = 0, z = 0;) auch die Teilchengeschwindigkeiten als private Daten-Member enthalten (z.B. zunächst initialisiert auf Null: double v_x = 0, v_y = 0, v_z = 0;). Mittels des Konstruktors kann der Benutzer dann die Anfangsorte und Anfangsgeschwindigkeiten der einzelnen Teilchen festlegen.

Nun wird zusätzlich noch das zeitliche Verhalten der Dinge als eine inline Funktion definiert. Die Teilchen sollen hierbei zunächst nicht miteinander wechselwirken. Die Begrenzungen der Kiste bilden jedoch nicht überwindbare Barrieren für die klassischen Teilchen und es soll eine vollständige Reflexion an den Kistenbegrenzungen stattfinden. Wir implementieren dieses zeitliche Verhalten der Teilchen als eine inline-Methode der Klasse 'Ding' und benennen sie 'Gehe_Zeitschritt': **inline** void Ding::Gehe_Zeitschritt(double dt, double max_x, double max_y, double max_z){ … } .
Die Funktion beschreibt die Veränderung der Ortskoordinaten (x, y, z) bei einem Zeitschritt dt, wobei die speziellen Randbedingungen an die Bewegung der Teilchen (Reflexion an den Rändern der Kiste) mittels der Argumente double max_x, double max_y, double max_z spezifiziert werden.

Im Hauptprogramm sollen dann ein vector-Container mittels 'vector<Ding> Kist… 'Kiste_Teilchen.push_back(Ding {…});' aufgefüllt. Nach diesem Initialisierungspr… Zeitschritte (double Anz_tSchritte = 100; mit double dt = 0.05;) im Terminal ausgeg… vector-Containers bestehend aus 10 Dingen mit zeitlicher E…

```
n.Gehe_Zeitschritt(dt,kiste_x, kiste_y, 0.0);
```

Die Klasse Ding
der nicht-wechselwirkenden Teichen

Vector_Dinge.cpp

```cpp
/* Beispiel fuer einen Vector-Container bestehend aus mehreren Elementen des Typs Ding
 * Ding ist eine Klasse bestehend aus
 * Sieben privaten Instanzvariablen (Ort und Geschwindigkeiten des Dings in 3D)
 * Fuenf ueberladenen Konstruktoren
 * Sieben oeffentlichen const Member-Funktionen
 * und einer oeffentlichen Member-Funktionen 'Gehe_Zeitschritt',
 * die eine zeitliche Entwicklung im Programm implementiert
 * Ausgabe zum Plotten (Python) mittels: "./a.out > Vector_Dinge.dat"
 * python3 PythonPlot_Vector_Dinge.py
*/
#include <iostream>                  // Ein- und Ausgabebibliothek
#include <vector>                    // Sequenzieller Container vector<Type> der Standardbibliothek
using namespace std;                 // Benutze den Namensraum std

//Definition der Klasse 'Ding'
class Ding{
    // Private Instanzvariablen (Daten-Member) der Klasse
    unsigned int n;                  // Nummer des Dinges
    double x = 0, y = 0, z = 0;      // Ort des Dinges
    double v_x = 0, v_y = 0, v_z = 0; // Geschwindigkeit des Dinges

    // Oeffentliche Konstruktoren und Member-Funktionen der Klasse
    public:
        // Fuenf ueberladene Konstruktoren der Klasse
        // Konstruktor mit sieben Argumenten
        Ding(unsigned int set_n, double set_x, double set_y, double set_z, double set_v_x, double set_v_y, double set_v_z) : n{set_n}, x{set_x}, y{set_y}, z{set_z}, v_x{set_v_x}, v_y{set_v_y}, v_z{set_v_z} {
            printf("# Konstruktor(n,x,y,z,,vx,vy,vz) erzeugt das Ding %i \n",n);
        }
        // Konstruktor mit vier Argumenten
        Ding(unsigned int set_n, double set_x, double set_y, double set_z) : n{set_n}, x{set_x}, y{set_y}, z{set_z} {
            printf("# Konstruktor(n,x,y,z) erzeugt ein das Ding %i \n",n);
        }
        // Konstruktor mit drei Argumenten
        Ding(unsigned int set_n, double set_x, double set_y) : n{set_n}, x{set_x}, y{set_y} {
            printf("# Konstruktor(n,x,y) erzeugt das Ding %i \n",n);
        }
        // Konstruktor mit zwei Argumenten
        Ding(unsigned int set_n, double set_x) : n{set_n}, x{set_x} {
            printf("# Konstruktor(n,x) erzeugt das Ding %i \n",n);
        }
        // Konstruktor ohne Argument (Standard-Konstruktor)
        Ding() : n{0} {
            printf("# Konstruktor() erzeugt das Ding %i \n", n);
        }

        // Member-Funktionen der Klasse
        void Gehe_Zeitschritt(double dt, double max_x, double max_y, double max_z); // Deklaration einer Member-Funktion (Definition findet ausserhalb der Klasse statt)

        // als const deklariert, da sie die privaten Instanzvariablen nicht veraendern:
        unsigned int get_Nummer() const {return n;}
        double get_Ort_x() const {return x;}
        double get_Ort_y() const {return y;}
        double get_Ort_z() const {return z;}

        double get_Geschw_x() const {return v_x;}
        double get_Geschw_y() const {return v_y;}
        double get_Geschw_z() const {return v_z;}
};
```

Die inline Funktion „Gehe_Zeitschritt(...)“ und das Hauptprogramm main()

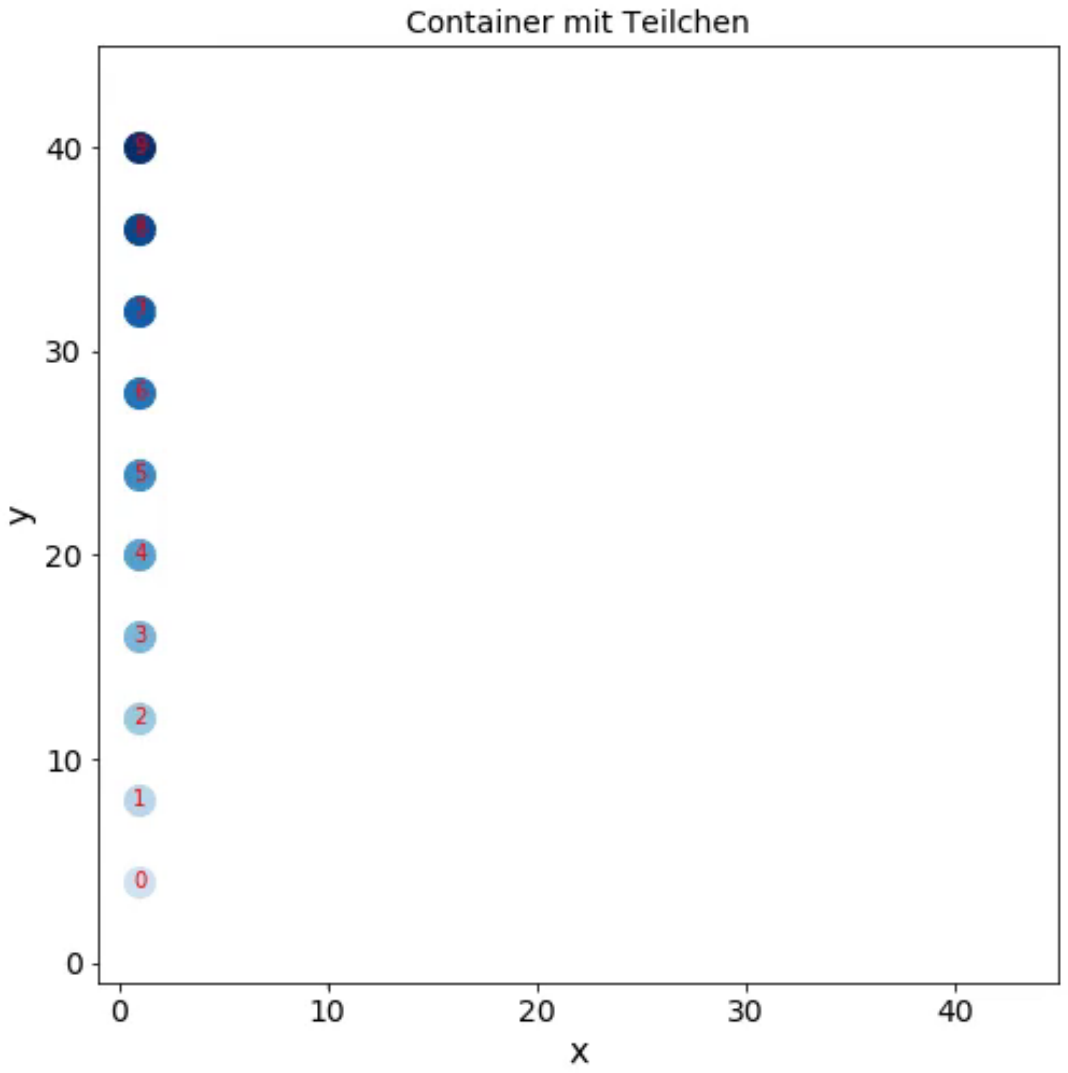
```
/* Definition der Funktion Gehe_Zeitschritt(...) als inline-Methode der Klasse Ding
 * Die Funktion beschreibt die Veränderung der Ortskoordinaten (x,y,z) bei einem Zeitschritt dt
 * Es sind zusaetzlich spezielle Randbedingungen an die Bewegung formuliert, so dass sich die
 * Dinge nur in einem Bereich von [0,max_x], [0,max_y] und [0,max_z] bewegen koennen */
inline void Ding::Gehe_Zeitschritt(double dt, double max_x, double max_y, double max_z){
    x = x + v_x * dt;
    y = y + v_y * dt;
    z = z + v_z * dt;
    if (x >= max_x || x <= 0){v_x = - v_x;}
    if (y >= max_y || y <= 0){v_y = - v_y;}
    if (z >= max_z || z <= 0){v_z = - v_z;}
}

int main(){                          // Hauptfunktion
    double kiste_x = 40;             // Laenge der Kiste
    double kiste_y = 40;             // Breite der Kiste
    unsigned int Anz_Teilchen = 10; // Definition der Anzahl der zu erzeugenden Dinge

    double t;                        // Deklaration des Zeitparameters
    double dt = 0.05;                // Definition der Laenge des Zeitschrittes
    double Anz_tSchritte = 100;      // Anzahl der Zeitschritte

    vector<Ding> Kiste_Teilchen;     // Deklaration eines vector-Containers

    for (unsigned int n = 0; n < Anz_Teilchen; ++n){ // for-Schleife zum Auffuellen des Containers mit Elementen vom Typ 'Ding'
        Kiste_Teilchen.push_back( Ding {n, 1, (n+1)*kiste_x/Anz_Teilchen, 0, (n+1.0), 0, 0} ); // Initialisierung: Nur in x-y-Ebene, Geschwindigkeit nur in x-Richtung
    }

    printf("# 0: Index i \n# 1: Zeit t \n# 2: Nummer des Teilchens 1 \n");          // Beschreibung der ausgegebenen Groessen
    printf("# 3: x-Position des Teilchens 1 \n# 4: y-Position des Teilchens 1 \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 5: Nummer des Teilchens 2 \n# 6: .... bis Anzahl der Teichen \n");    // Beschreibung der ausgegebenen Groessen

    for (int i = 0; i < Anz_tSchritte; ++i){             // for-Schleife fuer die zeitliche Entwicklung der Dinge in der Kiste
        t = i * dt;                                      // Zeit geht in dt-Schritten voran
        printf("%3i %10.6f ", i, t);                     // Ausgabe Index i und Zeit t
        for (auto& n : Kiste_Teilchen){                  // Bereichsbasierte for-Schleife zum Ausgeben der x-y-Orte der Dinge
            printf("%3i %10.6f %10.6f ", n.get_Nummer(), n.get_Ort_x(), n.get_Ort_y()); // Ausgabe Teilchenorte
            n.Gehe_Zeitschritt(dt,kiste_x, kiste_y, 0.0); // Aufruf der inline Funktion Gehe_Zeitschritt(...)
        }
        printf("\n");
    }
}
```

PythonPlot_Vector_Dinge.py

```python
# Python Programm zum Plotten der Daten des Vector-Containers mit 10 Dingen (Vector_Dinge.cpp)
# Es werden hier mehrere Bilder der zeitlichen Entwicklung des Systems in einem Ordner 'Bilder' gespeichert
# !!!! Sie muessen vor der Ausfuehrung des Programms den Ordner Bilder erstellen !!!!
# Die einzelnen Bilder kann mann dann mittels des folgenden Terminalbefehls zu einem Video binden:
# ffmpeg -framerate 5 -i './Vector_Dinge_%03d.png' -c:v libx264 Vector_Dinge.mp4

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/ )

data = np.genfromtxt("./Vector_Dinge.dat")  # Einlesen der berechneten Daten von Vector_Dinge.cpp

plt.title(r'Container mit Teilchen')        # Titel der Abbildung
plt.ylabel('y')                             # Beschriftung y-Achse
plt.xlabel('x')                             # Beschriftung x-Achse

r = 200                                     # Radius eines Dings
plot_min=0                                  # Festlegung der x-Untergrenze (Abmessung Kiste)
plot_max=40                                 # Festlegung der x-Obergrenze (Abmessung Kiste)
anz_teilchen = 10                           # Definition der Anzahl der Dinge

cmap = plt.cm.Blues                                 # Definition der Farbschattierung der Dinge
line_colors = cmap (np.linspace(0.2,1,anz_teilchen)) # Definition der Farbschattierung der Dinge

for it in range(len(data[:,0])):                    # for-Schleife fuer die zeitliche Entwicklung der Dinge in der Kiste
    print(it)                                       # Terminalausgabe der Erstellung des i-ten Bildes
    plt.cla()
    for i in range(anz_teilchen):                   # for-Schleife ueber die Teilchen in der Kiste
        plt.scatter(data[it,3*i+3],data[it,3*i+4], marker='o', color=line_colors[i], s=r) # Kennzeichnung der Position des Dinges durch einen blauen Kreis
        plt.text(data[it,3*i+3],data[it,3*i+4], str(int(data[it,3*i+2])), fontsize=10, verticalalignment='center', horizontalalignment='center', color="red") # Ding Nr.
    plt.xlim(-1,45)                                 # Plot-Limit x-Achse
    plt.ylim(-1,45)                                 # Plot-Limit y-Achse

    # Bild-Ausgabe mit Speicherung eines individuellen Iteration-Namens
    pic_name = "./Bilder/" + "Vector_Dinge_" + "{:0>3d}".format(it) + ".png"
    plt.savefig(pic_name, dpi=200,bbox_inches="tight",pad_inches=0.05,format="png")
    plt.close()
```

Python Skript zur Visualisierung der Bewegung der Teilchen in der Kiste

Python Visualisierung der Daten aus Vector_Dinge.cpp



Die zum Anfang initialisierten Orte und Geschwindigkeiten der Teilchen werden mittels des Konstruktors festgelegt: 'Ding { Argumentenliste des aufgerufenen Konstruktors }'.

```
for (unsigned int n = 0; n < Anz_Teilchen; ++n){
    Kiste_Teilchen.push_back( Ding {n, 1, (n+1)*kiste_x/Anz_Teilchen, 0, (n+1.0), 0, 0} );
}
```

Mittels der vom Benutzer festgelegten Argumentenliste wird einer der überladenen Konstruktoren der Klasse Ding aufgerufen. Hier wurde der Konstruktor mit sieben Argumenten gewählt (Ding(Nummer,x,y,z,vx,vy,vz)), bei dem die Teilchennummer und der Orte und die Geschwindigkeiten des Teilchens individuell initialisiert wird. Die gewählte Anfangskonfiguration der Teilchen entspricht einer Teilchenbewegung in x-Richtung, wobei alle Teilchen bei x=1 und z=0 starten und ihre y-Postion äquidistant variiert. Die Teilchen mit einer hohen Teilchennummer n bewegen sich schneller als die Teilchen mit niedriger Nummer.

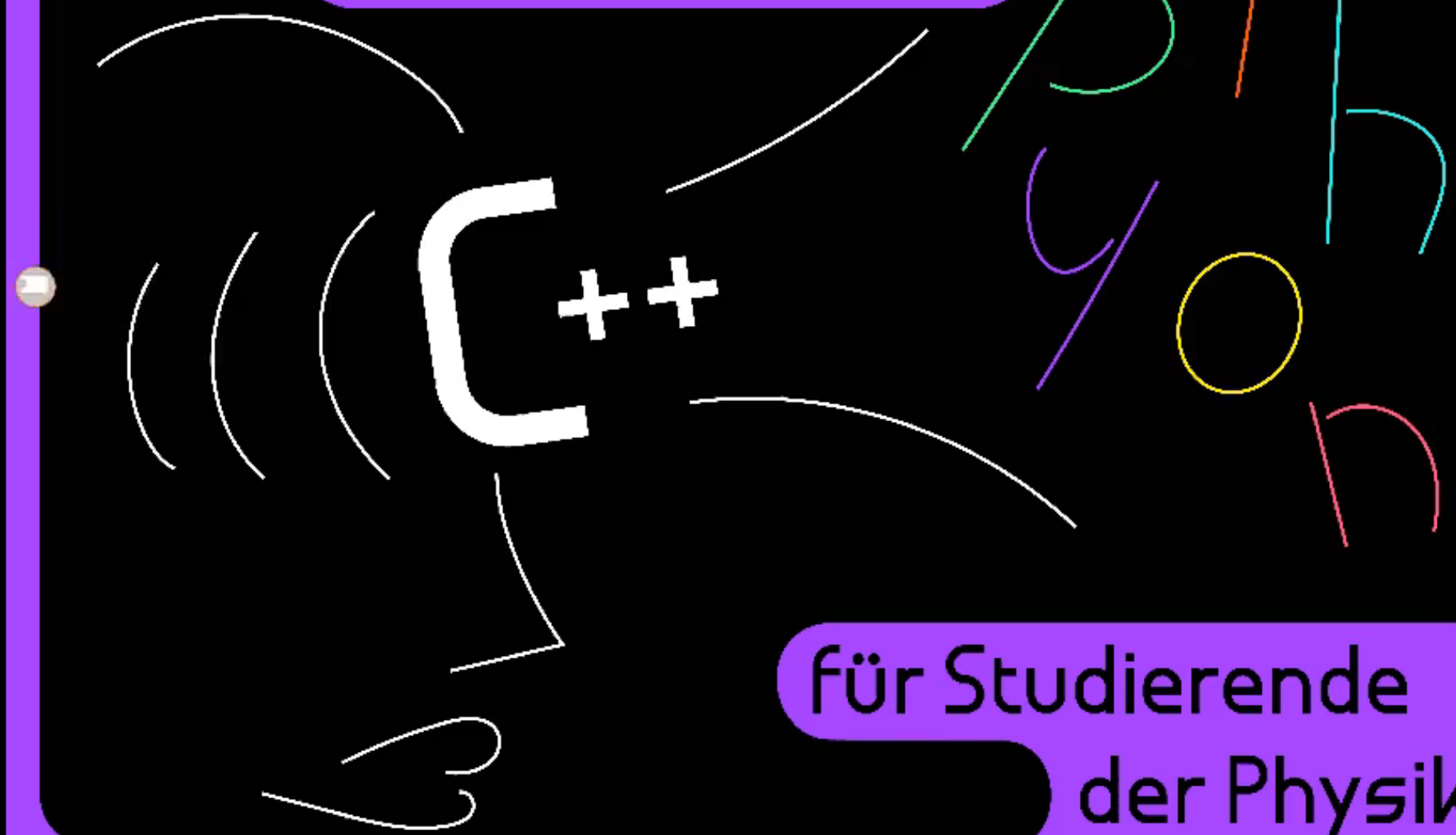
```cpp
/* Definition der Funktion Gehe_Zeitschritt(...) als inline-Methode der Klasse Ding
 * Die Funktion beschreibt die Veränderung der Ortskoordinaten (x,y,z) bei einem Zeitschritt dt
 * Es sind zusaetzlich spezielle Randbedingungen an die Bewegung formuliert, so dass sich die
 * Dinge nur in einem Bereich von [50,max_x], [50,max_y] und [50,max_z] bewegen koennen */
inline void Ding::Gehe_Zeitschritt(double dt, double max_x, double max_y, double max_z){
    x = x + v_x * dt;
    y = y + v_y * dt;
    z = z + v_z * dt;
    if (x >= max_x || x <= 90){v_x = - v_x;}
    if (y >= max_y || y <= 90){v_y = - v_y;}
    if (z >= max_z || z <= 90){v_z = - v_z;}
}

int main(){                          // Hauptfunktion
    double kiste_x = 5334 - 200;     // Laenge der Kiste (ein wenig kuerzer als
    double kiste_y = 4000 - 200;     // Breite der Kiste (ein wenig kuerzer als
    unsigned int Anz_Teilchen = 30; // Definition der Anzahl der zu erzeugende

    double t;                        // Deklaration des Zeitparameters
    double dt = 0.01;                // Definition der Laenge des Zeitschrittes
    double Anz_tSchritte = 200;      // Anzahl der Zeitschritte

    vector<Ding> Kiste_Teilchen;     // Deklaration eines vector-Containers

    for (unsigned int n = 0; n < Anz_Teilchen; ++n){ // for-Schleife zum Auffuellen des Containers mit Elementen vom Typ 'Ding'
        Kiste_Teilchen.push_back( Ding {n, 100, kiste_y/2, 0, kiste_x/2 + kiste_x*pow(sin(n),2), kiste_y*sin(2*M_PI*n/Anz_Teilchen)/2 , 0} );
    }

    printf("# 0: Index i \n# 1: Zeit t \n# 2: Nummer des Teilchens 1 \n");          // Beschreibung der ausgegebenen Groessen
    printf("# 3: x-Position des Teilchens 1 \n# 4: y-Position des Teilchens 1 \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 5: Nummer des Teilchens 2 \n# 6: .... bis Anzahl der Teichen \n");    // Beschreibung der ausgegebenen Groessen

    for (int i = 0; i < Anz_tSchritte; ++i){             // for-Schleife fuer die zeitliche Entwicklung der Dinge in der Kiste
        t = i * dt;                                      // Zeit geht in dt-Schritten voran
        printf("%3i %10.6f ", i, t);                     // Ausgabe Index i und Zeit t
        for (auto& n : Kiste_Teilchen){                  // Bereichsbasierte for-Schleife zum Ausgeben der x-y-Orte der Dinge
            printf("%3i %10.6f %10.6f ", n.get_Nummer(), n.get_Ort_x(), n.get_Ort_y()); // Ausgabe Teilchenorte
            n.Gehe_Zeitschritt(dt,kiste_x, kiste_y, 0.0); // Aufruf der inline Funktion Gehe_Zeitschritt(...)
        }
        printf("\n");
    }
}
```

Vector_Dingea.cpp

Wir möchten nun kleine Abänderungen in den Anfangsbedingungen der Teilchen machen. Die Größe der Kiste soll jetzt den Abmessungen des Leitbildes der Vorlesung illustration_DeborahMoldawski.jpg entsprechen und es sollen 30 Teilchen in diese Kiste eingesperrt sein. Die Teilchen sollen am Anfang auf der linken Seite auf mittlerer Höhe starten und sich dann in unterschiedlicher Weise nach rechts bewegen. Das folgende C++ Programm stellt eine solche Teilchenentwicklung dar.

PythonPlot_Vector_Dingea.py

Einführung in die Programmierung



für Studierende der Physik

Differentialgleichungen: Numerische Lösung von Anfangswertproblemen

Im vorigen Unterpunkt hatten wir die Bewegung einzelner Teilchen in einer Kiste simuliert. In der Physik ist die zeitliche Entwicklung eines Systems of in Form von Differentialgleichungen (DGLs) gegeben. In diesem Unterpunkt betrachten wir das numerische Lösen einer Differentialgleichung erster Ordnung der Form

$$\dot{y}(t) = \frac{dy(t)}{dt} = f(t, y(t)) \quad \text{, mit: } a \leq t \leq b\,,\ y(a) = \alpha \quad .$$

Die Funktion $f(t, y(t))$ bestimmt die DGL und somit das Verhalten der gesuchten Funktion $y(t)$. Es wird hierbei vorausgesetzt, dass $f(t, y(t))$ auf einer Teilmenge $\mathcal{D} = \{(t, y) | a \leq t \leq b\,,\ -\infty \leq y \leq \infty\}$ kontinuierlich definiert ist. Weiter wird angenommen, dass das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $y(t)$ existiert ("well-posed" bedeutet hier, dass die Differentialgleichung eine Struktur hat, bei der kleine Störungen im Anfangszustand nicht exponentiell anwachsen). Wir hatten bereits gesehen, wie man Differentialgleichungen mittels Jupyter Notebooks und SymPy DGLs analytisch löst (siehe Jupyter Notebooks und das Rechnen mit symbolischen Ausdrücken). Nicht jede DGL lässt sich analytisch lösen und falls der Befehl "dsolve()" keine sinnvollen Resultate liefert, muss man die zeitliche Entwicklung der Funktion $y(t)$ numerisch berechnen. Die numerische Lösung der DGL kann man sich auch direkt in Python mittels der Methode "integrate.odeint()" berechnen (Python-Modul "scipy"). Möchte man die Lösung jedoch in einem C++ Programm berechnen, so ist man auf die Anwendung eines numerischen Verfahrens angewiesen.

Das einfache Euler Verfahren zum Lösen einer DGL

Das wohl einfachste Verfahren zum Lösen einer DGL erster Ordnung ist die Euler Methode. Hierzu schreibt man die DGL als eine Differenzengleichung um

$$\frac{dy(t)}{dt} = f(t, y(t)) \;\rightarrow\; \Delta y = f(t, y) \cdot \Delta t \;\rightarrow\; \Delta y = h \cdot f(t, y)$$

und unterteilt das Zeitintervall $[a, b]$ in $N + 1$ äquidistante Zeit-Gitterpunkte $(t_0, t_1, t_2, \ldots, t_N)$, wobei $t_i = a + i\,h \quad \forall i = 0, 1, 2, \ldots, N$. Im Algorithmus der Euler Methode startet man bei $t = t_0$ und $y = y_0 = \alpha$ (Anfangsbedingungen des Systems) und erhöht dann iterativ die Zeit t um den Wert von h. Den neuen y-Wert erhält man mittels $y_1 = y_0 + h \cdot f(t_0, y_0)$ und man führt das Verfahren so lange aus, bis man an den letzten zeitlichen Gitterpunkt gelangt.

Betrachten wir z.B. die einfache Differentialgleichung

$$\frac{dy(t)}{dt} = f(t, y(t)) = -y(t) \quad ,$$

die den exponentiellen Abfall einer Funktion $y(t)$ beschreibt. Obwohl sich die allgemeine Lösung der DGL einfach bestimmen lässt ($y(t) = \alpha \cdot e^{-t}$, mit $\alpha = y(0)$), möchten wir die DGL auf numerischem Wege lösen. Das folgende C++ Programm benutzt die Eulermethode und entwickelt die obere Differentialgleichung im Zeitintervall $[a, b] = [0, 2]$ mittels 101 Gitterpunkten. Die simulierten Daten werden dann, zusammen mit der analytischen Lösung im Terminal ausgegeben.

DGL_0.cpp

```cpp
/* Berechnung der Lösung einer Differentialgleichung der Form y'=f(t,y)
 * mittels der einfachen Euler Methode und f(t,y) = y
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 */
#include <stdio.h>                                          // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>                                            // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f(double t, double y){                               // Definition der Funktion f(t,x)
    double wert;
    wert = - y;                                             // Eigentliche Definition der Funktion
    return wert;                                            // Rueckgabewert der Funktion f(t,x)
}                                                           // Ende der Funktion f(t,x)

double y_analytisch(double t, double alpha){                // Analytische Loesung der DGL
    double wert;                                            // bei gegebenem Anfangswert y(a)=alpha
    wert = alpha*exp(-t);                                   // Eigentliche Definition der analytische Loesung
    return wert;                                            // Rueckgabewert
}                                                           // Ende der Definitiom

int main(){                                                 // Hauptfunktion
    double a = 0;                                           // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
    double b = 2;                                           // Obergrenze des Intervalls [a,b]
    int N = 100;                                            // Anzahl der Punkte in die das t-Intervall aufgeteilt wird
    double h = (b - a)/N;                                   // Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
    double alpha = 0.5;                                     // Anfangswert bei t=a: y(a)=alpha
    double t;                                               // Aktueller Zeitwert
    double y = alpha;                                       // Deklaration und Initialisierung der numerischen Loesung der Euler Methode

    printf("# 0: Index i \n# 1: t-Wert \n# 2: Euler Methode \n"); // Beschreibung der ausgegebenen Groessen
    printf("# 3: Analytische Loesung  \n");                 // Beschreibung der ausgegebenen Groessen

    for(int i = 0; i <= N; ++i){                            // for-Schleife ueber die einzelnen Punkte des t-Intervalls
        t = a + i*h;                                        // Zeit-Parameter wird um h erhoeht
        printf("%3d %19.15f %19.15f %19.15f\n",i, t, y, y_analytisch(t,0.5)); // Ausgaben der Loesung

        y = y + h*f(t,y);                                   // Euler Methode
    }                                                       // Ende for-Schleife
}                                                           // Ende der Hauptfunktion
```

Lösen einer einfachen DGL erster Ordnung mittels des Euler Verfahrens

Numerische Verfahren zum Lösen von Differentialgleichung erster Ordnung

Das Euler Verfahren:

$$y_{i+1} = y_i + h \cdot f(t_i, y_i)$$

Mittelpunktmethode:

$$y_{i+1} = y_i + h \cdot \left[f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2} f(t_i, y_i)\right)\right]$$

Modifizierte Euler Methode:

$$y_{i+1} = y_i + \frac{h}{2} \cdot [f(t_i, y_i) + f(t_{i+1}, y_i + h\, f(t_i, y_i))]$$

Runge-Kutta Ordnung vier:

$$y_{i+1} = y_i + \frac{1}{6} \cdot (k_1 + 2\,k_2 + 2\,k_3 + k_4) \quad \text{, wobei:}$$

$$k_1 = h\, f(t_i, y_i)$$

$$k_2 = h\, f\left(t_i + \frac{h}{2}, y_i + \frac{1}{2} k_1\right)$$

$$k_3 = h\, f\left(t_i + \frac{h}{2}, y_i + \frac{1}{2} k_2\right)$$

$$k_4 = h\, f\left(t_{i+1}, y_i + k_3\right)$$

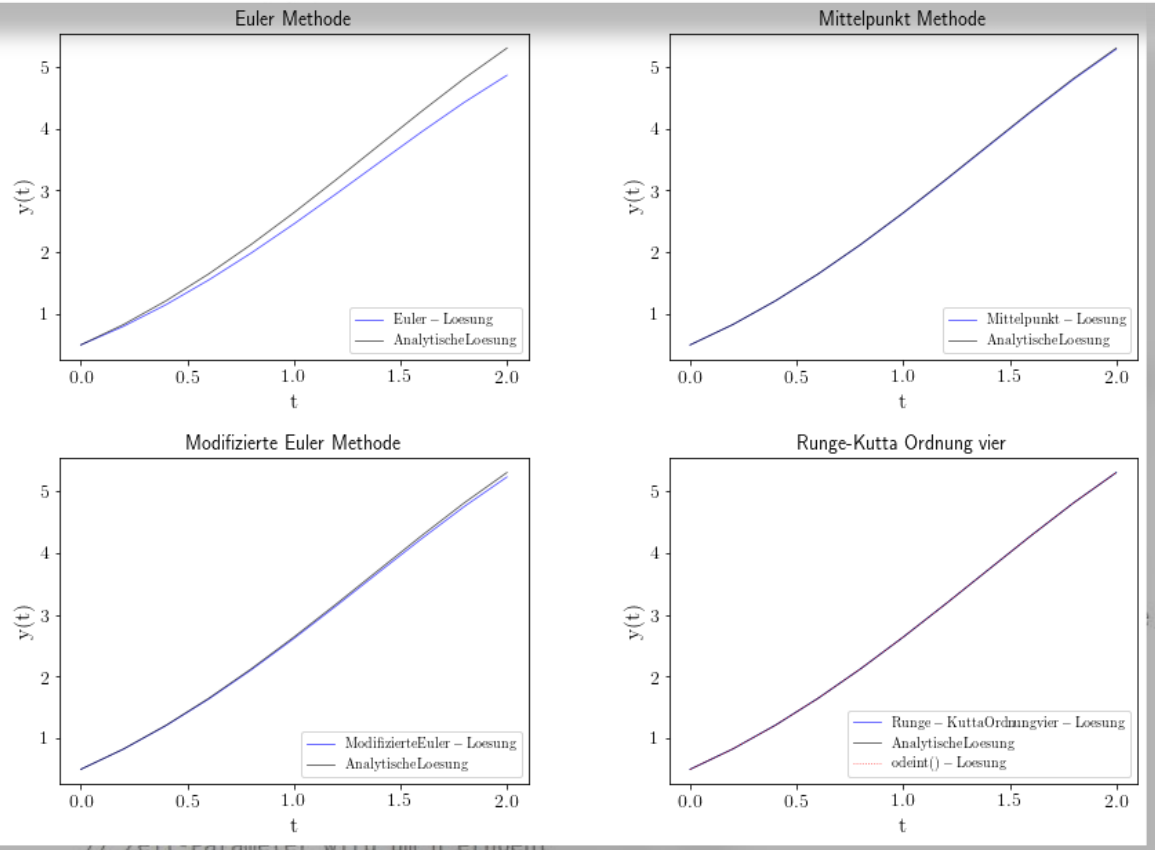
Anwendung auf die DGL: $\frac{dy}{dt} = y - t^2 + 1$

```
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 *  Ausgabe zum Plotten mittels Python Jupyter Notebook DGL_1.ipynb: "./a.ou
 */
#include <stdio.h>
#include <cmath>

double f(double t, double y){
    double wert;
    wert = y - pow(t,2) +1;
    return wert;
}

double y_analytisch(double t, double alpha){
    double wert;
    wert = (alpha + (pow(t,2) + 2*t + 1)*exp(-t) -1)*exp(t);
    return wert;
}

int main(){
    double a = 0;
    double b = 2;
    int N = 10;
    double h = (b - a)/N;
    double alpha = 0.5;
    double t;
    double y_Euler = alpha;
    double y_Midpoint = alpha;
    double y_Euler_M = alpha;
    double y_RungeK_4 = alpha;
    double k1, k2, k3, k4;

    printf("# 0: Index i \n# 1: t-Wert \n# 2: Euler Methode \n");
    printf("# 3: Mittelpunkt Methode \n# 4: Modifizierte Euler Methode \n");
    printf("# 5: Runge-Kutta Ordnung vier  \n# 6: Analytische Loesung  \n");

    for(int i = 0; i <= N; ++i){
        t = a + i*h;
        printf("%3d %19.15f %19.15f %19.15f %19.15f %19.15f %19.15f\n",i, t, y_Euler, y_Midpoint, y_Euler_M, y_RungeK_4, y_analytisch(t,0.5));

        y_Euler = y_Euler + h*f(t,y_Euler);
        y_Midpoint = y_Midpoint + h*f(t+h/2,y_Midpoint+h/2*f(t,y_Midpoint));
        y_Euler_M = y_Euler_M + h/2*( f(t,y_Euler_M) + f(t+h,y_Euler_M+h*f(t,y_Euler_M)) );

        k1 = h*f(t,y_RungeK_4);
        k2 = h*f(t+h/2,y_RungeK_4+k1/2);
        k3 = h*f(t+h/2,y_RungeK_4+k2/2);
        k4 = h*f(t+h,y_RungeK_4+k3);
        y_RungeK_4 = y_RungeK_4 + (k1 + 2*k2 + 2*k3 + k4)/6;
    }
}
```

```
// Eigentliche Definition der Funktion
// Rueckgabewert der Funktion f(t,x)
// Ende der Funktion f(t,x)

// Analytische Loesung der DGL
// bei gegebenem Anfangswert y(a)=alpha
// Eigentliche Definition der analytische Loesung
// Rueckgabewert
// Ende der Definitiom

// Hauptfunktion
// Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
// Obergrenze des Intervalls [a,b]
// Anzahl der Punkte in die das t-Intervall aufgeteilt wird
// Abstand dt zwischen den aequidistanten Punkten des t-Intervalls (h=dt)
// Anfangswert bei t=a: y(a)=alpha
// Aktueller Zeitwert
// Deklaration und Initialisierung der numerischen Loesung der Euler Methode
// Deklaration und Initialisierung der numerischen Loesung der Mittelpunkt Methode
// Deklaration und Initialisierung der numerischen Loesung der modifizierte Euler Methode
// Deklaration und Initialisierung der numerischen Loesung der Runge-Kutta Ordnung vier Methode
// Deklaration der vier Runge-Kutta Parameter

// Beschreibung der ausgegebenen Groessen
// Beschreibung der ausgegebenen Groessen
// Beschreibung der ausgegebenen Groessen

// for-Schleife ueber die einzelnen Punkte des t-Intervalls
// Zeit-Parameter wird um h erhoeht
// Ausgaben der Loesung

// Euler Methode
// Mittelpunkt Methode
// Modifizierte Euler Methode

// Runge-Kutta Parameter 1
// Runge-Kutta Parameter 2
// Runge-Kutta Parameter 3
// Runge-Kutta Parameter 4
// Runge-Kutta Ordnung vier Methode
// Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
// Ende der Hauptfunktion
```

Anwendung auf die DGL: $\frac{dy}{dt} = y - t^2 + 1$

```
 * Zeitentwicklung der fuer unterschiedliche t-Werte in [a,b]
 *  Ausgabe zum Plotten mittels Python Jupyter Notebook DGL_1.ipynb: "./a.ou
 */
#include <stdio.h>
#include <cmath>

double f(double t, double y){
    double wert;
    wert = y - pow(t,2) +1;
    return wert;
}

double y_analytisch(double t, double alpha){
    double wert;
    wert = (alpha + (pow(t,2) + 2*t + 1)*exp(-t) -1)*exp(t);
    return wert;
}

int main(){
    double a = 0;
    double b = 2;
    int N = 10;
    double h = (b - a)/N;
    double alpha = 0.5;
    double t;
    double y_Euler = alpha;
    double y_Midpoint = alpha;
    double y_Euler_M = alpha;
    double y_RungeK_4 = alpha;
    double k1, k2, k3, k4;

    printf("# 0: Index i \n# 1: t-Wert \n# 2: Euler Methode \n");
    printf("# 3: Mittelpunkt Methode \n# 4: Modifizierte Euler Methode \n");
    printf("# 5: Runge-Kutta Ordnung vier  \n# 6: Analytische Loesung  \n");

    for(int i = 0; i <= N; ++i){
        t = a + i*h;
        printf("%3d %19.15f %19.15f %19.15f %19.15f %19.15f %19.15f\n",i, t, y_Euler, y_Midpoint, y_Euler_M, y_RungeK_4, y_analytisch(t,0.5)); // Ausgaben der Loesung

        y_Euler = y_Euler + h*f(t,y_Euler);                                          // Euler Methode
        y_Midpoint = y_Midpoint + h*f(t+h/2,y_Midpoint+h/2*f(t,y_Midpoint));         // Mittelpunkt Methode
        y_Euler_M = y_Euler_M + h/2*( f(t,y_Euler_M) + f(t+h,y_Euler_M+h*f(t,y_Euler_M)) ); // Modifizierte Euler Methode

        k1 = h*f(t,y_RungeK_4);                                                      // Runge-Kutta Parameter 1
        k2 = h*f(t+h/2,y_RungeK_4+k1/2);                                             // Runge-Kutta Parameter 2
        k3 = h*f(t+h/2,y_RungeK_4+k2/2);                                             // Runge-Kutta Parameter 3
        k4 = h*f(t+h,y_RungeK_4+k3);                                                 // Runge-Kutta Parameter 4
        y_RungeK_4 = y_RungeK_4 + (k1 + 2*k2 + 2*k3 + k4)/6;                         // Runge-Kutta Ordnung vier Methode
    }                                                                    // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
}                                                                        // Ende der Hauptfunktion
```



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

Numerisches Lösen einer DGL erster Ordnung mit Python

Numerisches Lösen von Differentialgleichungen (das Anfangswertproblem)

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 betrachten in diesem Jupyter Notebook das numerische Lösen einer Differentialgleichung (DGL) erster Ordnung der Form

$$\dot{y}(t) = \frac{dy(t)}{dt} = f(t, y(t)) \quad \text{, mit: } a \leq t \leq b\,,\ y(a) = \alpha \quad .$$

Die Funktion $f(t, y(t))$ bestimmt die DGL und somit das Verhalten der gesuchten Funktion $y(t)$. Es wird hierbei vorausgesetzt, dass $f(t, y(t))$ auf einer Teilmenge $D = \{(t, y) | a \leq t \leq b\,,\ -\infty \leq y \leq \infty\}$ kontinuierlich definiert ist. Weiter wird angenommen, dass das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $y(t)$ existiert ("well-posed" bedeutet hier, dass die Differentialgleichung eine Struktur hat, bei der kleine Störungen im Anfangszustand nicht exponentiell anwachsen).

Übungsblatt Nr. 9

Aufgabe 1 *(10 Punkte)*

Im Unterpunkt Differentialgleichungen: Numerische Lösung von Anfangswertproblemen hatten wir mittels des C++ Programmes DGL_1.cpp die numerische Lösung einer Differentialgleichung berechnet. Lagern Sie den Kern-Algorithmus der Berechnung der numerischen Lösung, eines Verfahrens Ihrer Wahl (z.B. Euler Verfahren oder Runge-Kutta Ordnung vier), als eine C++ Klasse (oder als eine C++ Funktion) aus. Berechnen Sie dann die numerische Lösung der folgenden Differentialgleichung erster Ordnung

$$\dot{y}(t) = \frac{dy(t)}{dt} = y - t^2 + 1 \quad \text{, mit: } a = 0 \leq t \leq b = 4\,,\ y(a) = y(0) = \alpha = 0.3$$

mittels eines Verfahrens Ihrer Wahl (z.B. Euler Verfahren oder Runge-Kutta Ordnung vier). Führen Sie drei numerische Simulationen durch, wobei Sie für die Anzahl N der Zeit-Gitterpunkte $t_i, i = 0, 1, 2, \ldots, N-1$ die folgenden Werte benutzen: $N = 50$, $N = 500$ und $N = 5000$. Geben Sie die berechneten Werte $(t_i, y_i, y_{analytisch}(t_i)$ und den Fehler $\mathcal{F} = y_i - y_{analytisch}(t_i))$ mit 15 Nachkommastellen für $t = 1$, $t = 2$ und $t = 4$ an.

Aufgabe 2 *(10 Punkte)*

Im Unterpunkt C++ Container und die vector Klasse der Standardbibliothek wurde mit dem C++ Programm Vector_Dinge.cpp eine Kiste (ein C++ Container) mit 10 Objekten der Klasse 'Ding' erzeugt. Die zeitliche Entwicklung der Ortskoordinaten der nicht miteinander wechselwirkenden Dinge wurde mittels der inline Funktion 'Gehe_Zeitschritt(...) modelliert. In dieser Funktion wurden ebenfalls die Randbedingungen der Kiste implementiert (Reflexion an den Wänden der Kiste).

Ändern Sie das Programm so ab, dass anstatt der Reflexion an den Rändern der Kiste, *periodische Randbedingungen* existieren. Bauen Sie zusätzlich ein weiteres Teilchenverhalten, bzw. eine weitere Umgebungseigenschaft, in das Programm ein . Starten Sie dann eine Simulation mit 70 Teilchen (unterschiedliche Anfangsorte und Geschwindigkeiten) und stellen Sie die Bewegung der Teilchen mittels eines Python-Skriptes (Jupyter Notebooks) als animierten Film dar.