Einführung in die Programmierung für Studierende der Physik

JOHANN WOLFGANG GOETHE UNIVERSITÄT
14.06.2022

MATTHIAS HANAUSKE

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

9. Vorlesung

Plan für die heutige Vorlesung

- Kurze Wiederholung der Vorlesung 8
- Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung
- Vorstellung möglicher Projekte

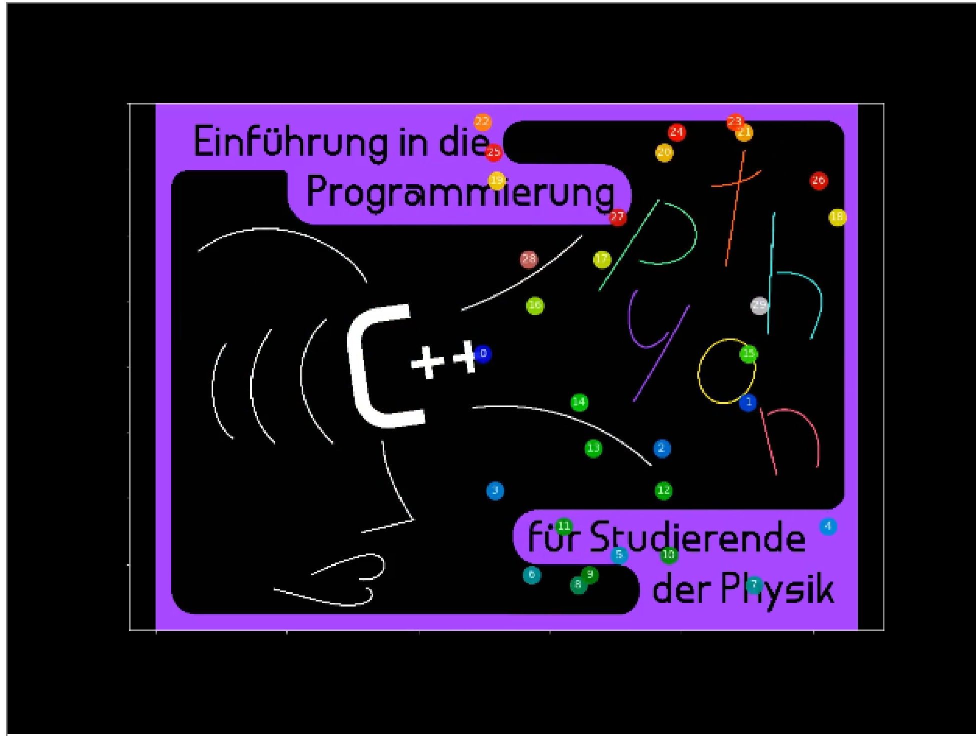
Wiederholung der Vorlesung 8

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

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.

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;').

Vector_1.cpp

Die Klasse <vector> am Beispiel eines Integer Vektors

```
#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 Klasse <vector> als ein Container von Objekten

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

```
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} );
}
```

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()

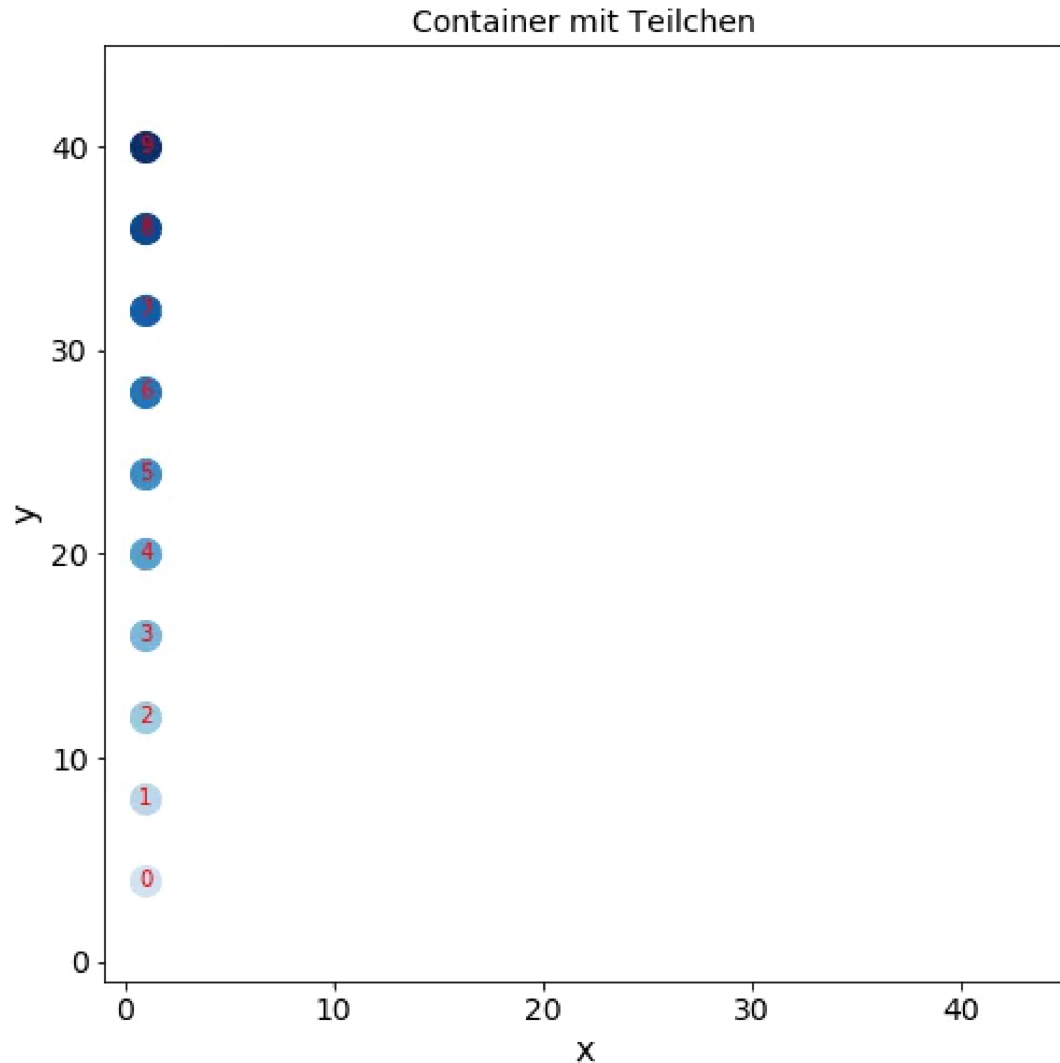
```
/* Definition der Funktion Gehe_Zeitschritt(...) als inline-Methode der Klasse Ding
 * Die Funktion beschreibt die Veränderung der Ortskoordinaten (x,y,z) bei einem Zeitschritt dt
 * Es sind zusaetzlich spezielle Randbedingungen an die Bewegung formuliert, so dass sich die
 * Dinge nur in einem Bereich 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

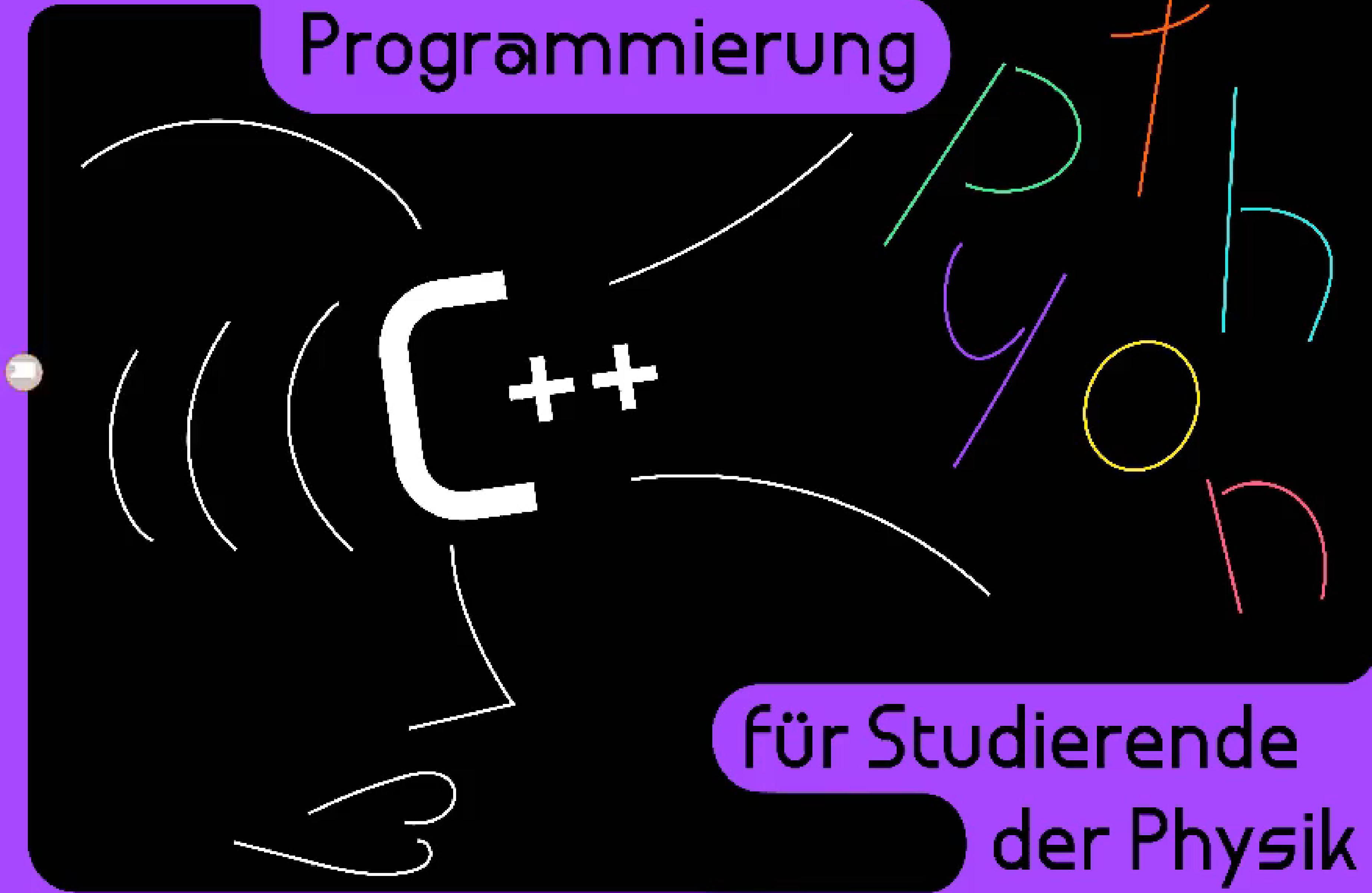


Die zum Anfang initialisierten Orte und Geschwindigkeiten der Teilchen werden mittels des Konstruktors festgelegt: 'Ding { Argumentenliste des aufgerufenen Konstruktors }'.

```
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.

PythonPlot_Vector_Dingea.py



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

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

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

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

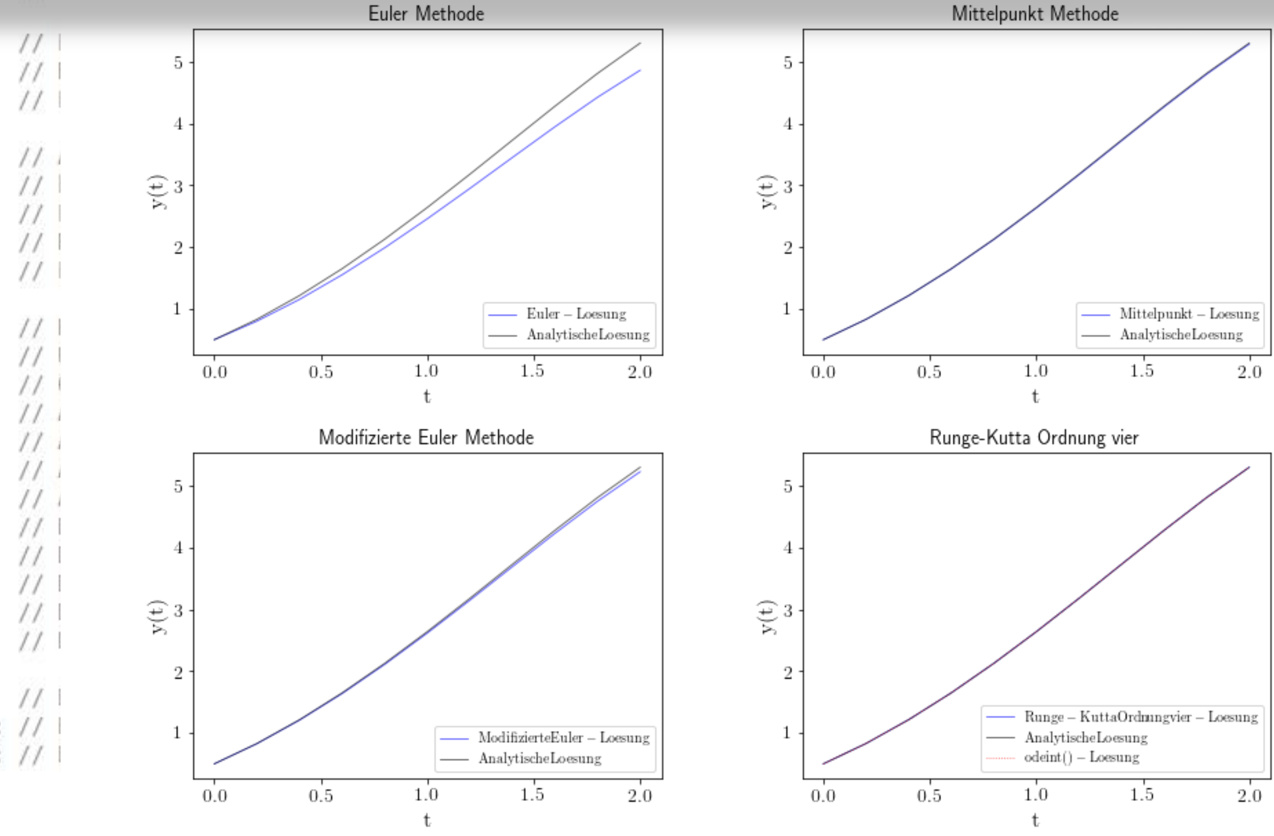
$$k_2 = h\, f\left(t_i + \frac{h}{2}, y_i + \frac{1}{2} k_1\right)$$

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

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

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
```

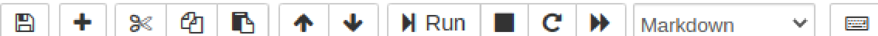


jupyter DGL_1 Last Checkpoint: vor 15 Stunden (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3



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 $\mathcal{D} = \{(t, y) | a \leq t \leq b \, , \; -\infty \leq y \leq \infty\}$ 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).

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.

Vorlesung 9

In dieser Vorlesung befassen wir uns zunächst mit dem numerischen Lösen von Systemen gekoppelter Differentialgleichungen und Differentialgleichungen zweiter Ordnung und stellen im darauf folgenden Teil mögliche Programmierprojekte vor, die von den Studierenden bearbeitet werden können. Beim Klicken auf die Überschriften der Projekte gelangen Sie zu einer detaillierteren Beschreibung der einzelnen Projektthemen.

Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung

In der vorigen Vorlesung hatten wir die unterschiedlichen Verfahren zum Lösen von Differentialgleichungen erster Ordnung kennengelernt. Die Bewegungsgleichungen vieler physikalischer Systeme sind jedoch von zweiter Ordnung in der Zeit und in diesem Teilkapitel beschreiben wir die Vorgehensweise wie man solche Differentialgleichungen höherer Ordnung numerisch mittels eines C++ Programmes löst. Um ein Differentialgleichung zweiter Ordnung mittels des Computers lösen zu können, schreibt man zunächst die DGL in ein System von zwei gekoppelten Differentialgleichungen ersten Ordnung um und diese löst man dann mit den Verfahren, die in der vorigen Vorlesung behandelt wurden. In diesem Unterpunkt werden wir uns zunächst mit Systemen von gekoppelten Differentialgleichungen erster Ordnung befassen und dann das numerische Lösen von Differentialgleichungen zweiter Ordnung vorstellen (näheres siehe Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung).

Studentische Projekte

Das Foucaultsche Pendel

Im Jahre 1851 gelang Jean Bernard Léon Foucault ein anschaulicher Beweis der Erdrotation. Aufgrund der, in rotierenden Bezugssystemen auftretenden Coriolisbeschleunigung, dreht sich die Schwingungsebene des Pendels langsam. Dieses Projekt ist ein Anwendungsfall der Newtonschen Mechanik in bewegten Bezugssystemen (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I3. Seite 18]) und das zugrundeliegende System von drei gekoppelten Differentialgleichungen zweiter Ordnung gilt es numerisch mittels eines C++ Programmes zu lösen und die berechneten Daten mittels Python zu visualisiern.

Das periodisch angetriebene Pendel

Das Projekt *periodisch angetriebenes Pendel* ist ein Anwendungsfall aus der klassischen Mechanik (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel VII27. Seite 496]). Das System besteht aus einem Pendel, auf welches zusätzlich eine äußere Kraft mit periodischer Zeitabhängigkeit wirkt. Außerdem soll das Pendel durch

Vorlesung 9

Das numerische Lösen von Differentialgleichungen ist ein mathematisch anspruchsvolles Thema und kann in dieser Vorlesung nicht im Detail erläutert werden. Im ersten Teil dieser Vorlesung sollen die im vorigen Unterpunkt (Differentialgleichungen: Numerische Lösung von Anfangswertproblemen) besprochenen Verfahren auf Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung angewendet werden. Die Bewegungsgleichungen vieler physikalischer Systeme sind von zweiter Ordnung in der Zeit und es soll die Vorgehensweise besprochen werden, wie man solche Differentialgleichungen höherer Ordnung numerisch mittels eines C++ Programmes löst.
Ab der nächsten Vorlesung (Vorlesung 10) werden die Studierenden dann an eigenen Projekten arbeiten, wobei viele dieser Projekte Probleme behandeln, die man nur mittels einer numerischen Lösung adäquat beschreiben kann (siehe linkes Panel dieser Vorlesung). Beim Klicken auf die Überschriften der Projekte gelangen Sie zu einer detaillierteren Beschreibung der einzelnen Projektthemen. Neben den hier vorgestellten Projekten, können auch eigene Projektthemen behandelt werden. Bei Interesse können auch Projekte aus dem Bereich der Physik der sozio-ökonomischen Systeme (z.B. Replikatordynamik der Evolutionären Spieltheorie, Simulationen von komplexen Netzwerken), oder Projekte im Themengebiet der allgemeinen Relativitätstheorie (z.B. Bewegung um ein schwarzes Loch mittels der Geodätengleichung, näheres siehe Allgemeine Relativitätstheorie mit dem Computer)) behandelt werden. Die studentischen Projekte können alleine oder in Gruppen (bis zu drei Personen) durchgeführt werden.

Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung

Im vorigen Unterpunkt hatten wir die unterschiedlichen Verfahren zum Lösen von Differentialgleichungen erster Ordnung kennengelernt. Die Bewegungsgleichungen vieler physikalischer Systeme sind jedoch von zweiter Ordnung in der Zeit und dieses Unterkapitel der Vorlesung 9 befasst sich damit, wie man solche Differentialgleichungen höherer Ordnung numerisch mittels eines C++ Programmes löst. Um ein Differentialgleichung zweiter Ordnung mittels des Computers lösen zu können, schreibt man zunächst die DGL in ein System von zwei gekoppelten Differentialgleichungen ersten Ordnung um und diese löst man dann mit den Verfahren, die in der vorigen Vorlesung behandelt wurden. In diesem Unterpunkt werden wir uns zunächst mit Systemen von gekoppelten Differentialgleichungen erster Ordnung befassen und dann das numerische Lösen von Differentialgleichungen zweiter Ordnung vorstellen.

Systeme von gekoppelten Differentialgleichungen

Wir betrachten zunächst das numerische Lösen eines Systems von m-gekoppelten Differentialgleichungen (DGLs) erster Ordnung der Form

$$\begin{aligned}
\dot{y}_1(t) &= \frac{dy_1}{dt} = f_1(t, y_1, y_2, \dots, y_m) \\
\dot{y}_2(t) &= \frac{dy_2}{dt} = f_2(t, y_1, y_2, \dots, y_m) \\
\dot{y}_3(t) &= \dots = \\
\dots &= \dots \\
\dot{y}_m(t) &= \frac{dy_m}{dt} = f_m(t, y_1, y_2, \dots, y_m) \quad ,
\end{aligned}$$

wobei die zeitliche Entwicklung der Vektorfunktion $\vec{y}(t) = (y_1(t), y_2(t), \dots, y_m(t))$ in den Grenzen $a \leq t \leq b$ gesucht wird. Die m-Funktionen $f_i(t, y_1, y_2, \dots, y_m)\,,\; i \in [1, 2, \dots, m]$ bestimmen das System der DGLs und somit das Verhalten der gesuchten Funktion $\vec{y}(t)$. Es wird hierbei vorausgesetzt, dass die Funktionen $f_i(t, y_1, y_2, \dots, y_m)$ auf einer Teilmenge $\mathcal{D}$ ($\mathbb{R}^{m+1} \supseteq \mathcal{D}$) kontinuierlich definiert sind und das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $\vec{y}(t)$ existiert. Bei gegebener Anfangskonfiguration

$$y_1(a) = \alpha_1\,,\; y_2(a) = \alpha_2\,,\; \dots\;,\; y_m(a) = \alpha_m$$

ist es dann numerisch möglich das System von gekoppelten DGLs zu lösen.

Systeme von gekoppelten Differentialgleichungen

Wir betrachten zunächst das numerische Lösen eines Systems von m-gekoppelten Differentialgleichungen (DGLs) erster Ordnung der Form

$$\begin{aligned}
\dot{y}_1(t) &= \frac{dy_1}{dt} = f_1(t, y_1, y_2, \dots, y_m) \\
\dot{y}_2(t) &= \frac{dy_2}{dt} = f_2(t, y_1, y_2, \dots, y_m) \\
\dot{y}_3(t) &= \dots = \\
\dots &= \dots \\
\dot{y}_m(t) &= \frac{dy_m}{dt} = f_m(t, y_1, y_2, \dots, y_m) \quad ,
\end{aligned}$$

wobei die zeitliche Entwicklung der Vektorfunktion $\vec{y}(t) = (y_1(t), y_2(t), \dots, y_m(t))$ in den Grenzen $a \leq t \leq b$ gesucht wird. Die m-Funktionen $f_i(t, y_1, y_2, \dots, y_m)\,,\; i \in [1, 2, \dots, m]$ bestimmen das System der DGLs und somit das Verhalten der gesuchten Funktion $\vec{y}(t)$. Es wird hierbei vorausgesetzt, dass die Funktionen $f_i(t, y_1, y_2, \dots, y_m)$ auf einer Teilmenge $\mathcal{D}$ ($\mathbb{R}^{m+1} \supseteq \mathcal{D}$) kontinuierlich definiert sind und das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $\vec{y}(t)$ existiert. Bei gegebener Anfangskonfiguration

$$y_1(a) = \alpha_1\,,\;\; y_2(a) = \alpha_2\,,\;\; \dots\;\;,\, y_m(a) = \alpha_m$$

ist es dann numerisch möglich das System von gekoppelten DGLs zu lösen.

Beispiel: Numerische Lösung eines Systems von zwei gekoppelten Differentialgleichungen erster Ordnung

Wir betrachten speziell das folgende System bestehend aus zwei gekoppelten DGLs ($m = 2$):

$$\dot{y}_1(t) = \frac{dy_1}{dt} = 3y_1 + 2y_2 - \left(2t^2 + 1\right) \cdot e^{2t} =: f_1(t, y_1, y_2)$$
$$\dot{y}_2(t) = \frac{dy_2}{dt} = 4y_1 + y_2 + \left(t^2 + 2t - 4\right) \cdot e^{2t} =: f_2(t, y_1, y_2) \quad ,$$

und sind an der numerischen Lösung $\vec{y}(t) = (y_1(t), y_2(t))$ im Zeitintervall $t \in [0, 1]$ interessiert. Die Anfangsbedingungen lauten

$$y_1(0) = \alpha_1 = 1\,,\; y_2(0) = \alpha_2 = 1 \quad .$$

Das Lösen dieses Systems von DGLs ist auf gleichem Wege möglich, wie man einzelne Differentialgleichungen numerisch approximiert.

C++ Programm: Implementierung der DGL bestimmenden Funktionen

und analytische Lösungen zum Vergleich

DGL_2.cpp

```
/* Berechnung der Lösung eines Systems von Differentialgleichung
 * der Form y1'=f1(t,y1,y2) , y2'=f2(t,y1,y2)
 * mittels der Euler Methode und Runge-Kutta Ordnung vier Methode
 * Zeitentwicklung von y1(t), y2(t) der fuer unterschiedliche t-Werte in [a,b]
 *  Ausgabe zum Plotten mittels Python Jupyter Notebook DGL_2.ipynb: "./a.out > DGL_2.dat
 */
#include <stdio.h>             // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>               // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f_1(double t, double y_1, double y_2){                          // Deklaration und Definition der Funktion f_1(t,y1,y2)
    double wert;
    wert = 3*y_1 + 2*y_2 - (2*pow(t,2) + 1)*exp(2*t);                  // Eigentliche Definition der Funktion
    return wert;                                                       // Rueckgabewert der Funktion f_1
}                                                                      // Ende der Funktion f_1

double f_2(double t, double y_1, double y_2){                          // Deklaration und Definition der Funktion f_2(t,y1,y2)
    double wert;
    wert = 4*y_1 + y_2 + (pow(t,2) +2*t - 4)*exp(2*t);                 // Eigentliche Definition der Funktion
    return wert;                                                       // Rueckgabewert der Funktion f_2
}                                                                      // Ende der Funktion f_2

double y_1_analytisch(double t){                                       // Analytische Loesung y_1(t)
    double wert;                                                       // bei gegebenem Anfangswert y_1(a)=1, y_2(a)=1
    wert = exp(5*t)/3 - exp(-t)/3 + exp(2*t);                          // Eigentliche Definition der analytische Loesung
    return wert;                                                       // Rueckgabewert
}                                                                      // Ende der Definition

double y_2_analytisch(double t){                                       // Analytische Loesung y_2(t)
    double wert;                                                       // bei gegebenem Anfangswert y_1(a)=1, y_2(a)=1
    wert = exp(5*t)/3 + 2*exp(-t)/3 + pow(t,2)*exp(2*t);               // Eigentliche Definition der analytische Loesung
    return wert;                                                       // Rueckgabewert
}                                                                      // Ende der Definitiom
```

```c
int main(){                                                                         // Hauptfunktion
    double a = 0;                                                                   // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
    double b = 1;                                                                   // 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_1 = 1;                                                             // 1.Anfangswert bei t=a: y_1(a)=alpha_1
    double alpha_2 = 1;                                                             // 2.Anfangswert bei t=a: y_2(a)=alpha_2
    double t;                                                                       // Aktueller Zeitwert
    double y_Euler_1 = alpha_1;                                                     // Deklaration und Initialisierung der numerischen Loesung der Euler Methode fuer y_1
    double y_RungeK_4_1 = alpha_1;                                                  // Deklaration und Initialisierung der numerischen Loesung der Runge-Kutta Ordnung vier Method
    double k1_1,k2_1,k3_1,k4_1;                                                     // Deklaration der vier Runge-Kutta Parameter fuer y_1
    double y_Euler_2 = alpha_2;                                                     // Deklaration und Initialisierung der numerischen Loesung der Euler Methode fuer y_2
    double y_RungeK_4_2 = alpha_2;                                                  // Deklaration und Initialisierung der numerischen Loesung der Runge-Kutta Ordnung vier Method
    double k1_2,k2_2,k3_2,k4_2;                                                     // Deklaration der vier Runge-Kutta Parameter fuer y_2
    double tmp;                                                                     // Variable zum Zwischenspeichern von Ergebnissen

    printf("# 0: Index i \n# 1: t-Wert \n# 2: Euler Methode y1 \n# 3: Euler Methode y2 \n");   // Beschreibung der ausgegebenen Groessen
    printf("# 4: Runge-Kutta Ordnung vier y1 \n# 5: Runge-Kutta Ordnung vier y2 \n");          // Beschreibung der ausgegebenen Groessen
    printf("# 6: Analytische Loesung y1 \n# 7: Analytische Loesung y2 \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 %19.15f ",i, t, y_Euler_1, y_Euler_2, y_RungeK_4_1);   // Ausgaben der Loesungen
        printf(" %19.15f %19.15f %19.15f \n", y_RungeK_4_2, y_1_analytisch(t), y_2_analytisch(t)); // Ausgaben der Loesungen

        tmp = y_Euler_1 + h*f_1(t,y_Euler_1,y_Euler_2);                             // Euler Methode
        y_Euler_2 = y_Euler_2 + h*f_2(t,y_Euler_1,y_Euler_2);                       // y_2 Euler Methode
        y_Euler_1 = tmp;                                                            // y_1 Euler Methode

        k1_1 = h*f_1(t,y_RungeK_4_1,y_RungeK_4_2);                                  // Runge-Kutta Parameter k1 fuer y_1
        k1_2 = h*f_2(t,y_RungeK_4_1,y_RungeK_4_2);                                  // Runge-Kutta Parameter k1 fuer y_2
        k2_1 = h*f_1(t+h/2,y_RungeK_4_1+k1_1/2,y_RungeK_4_2+k1_2/2);                // Runge-Kutta Parameter k2 fuer y_1
        k2_2 = h*f_2(t+h/2,y_RungeK_4_1+k1_1/2,y_RungeK_4_2+k1_2/2);                // Runge-Kutta Parameter k2 fuer y_2
        k3_1 = h*f_1(t+h/2,y_RungeK_4_1+k2_1/2,y_RungeK_4_2+k2_2/2);                // Runge-Kutta Parameter k3 fuer y_1
        k3_2 = h*f_2(t+h/2,y_RungeK_4_1+k2_1/2,y_RungeK_4_2+k2_2/2);                // Runge-Kutta Parameter k3 fuer y_2
        k4_1 = h*f_1(t+h,y_RungeK_4_1+k3_1,y_RungeK_4_2+k3_2);                      // Runge-Kutta Parameter k4 fuer y_1
        k4_2 = h*f_2(t+h,y_RungeK_4_1+k3_1,y_RungeK_4_2+k3_2);                      // Runge-Kutta Parameter k4 fuer y_2
        y_RungeK_4_1 = y_RungeK_4_1 + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6;            // Runge-Kutta Ordnung vier Methode fuer y_1
        y_RungeK_4_2 = y_RungeK_4_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6;            // Runge-Kutta Ordnung vier Methode fuer y_2
    }                                                                               // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
}                                                                                   // Ende der Hauptfunktion
```

main()-Programm Zur Lösung des Systems bestehend aus zwei DGLs erster Ordnung

```
int main(){
    double a = 0;
    double b = 1;
    int N = 100;
    double h = (b - a)/N;
    double alpha_1 = 1;
    double alpha_2 = 1;
    double t;
    double y_Euler_1 = alpha_
    double y_RungeK_4_1 = al
    double k1_1,k2_1,k3_1,k4_
    double y_Euler_2 = alpha_
    double y_RungeK_4_2 = al
    double k1_2,k2_2,k3_2,k4_
    double tmp;

    printf("# 0: Index i \n#
    printf("# 4: Runge-Kutta
    printf("# 6: Analytische

    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 %19.15f ",i, t, y_Euler_1, y_Euler_2, y_RungeK_4_1);   // Ausgaben der Loesungen
        printf(" %19.15f %19.15f %19.15f \n", y_RungeK_4_2, y_1_analytisch(t), y_2_analytisch(t)); // Ausgaben der Loesungen

        tmp = y_Euler_1 + h*f_1(t,y_Euler_1,y_Euler_2);         // Euler Methode
        y_Euler_2 = y_Euler_2
        y_Euler_1 = tmp;

        k1_1 = h*f_1(t,y_Rung
        k1_2 = h*f_2(t,y_Rung
        k2_1 = h*f_1(t+h/2,y_
        k2_2 = h*f_2(t+h/2,y_
        k3_1 = h*f_1(t+h/2,y_
        k3_2 = h*f_2(t+h/2,y_
        k4_1 = h*f_1(t+h,y_Ru
        k4_2 = h*f_2(t+h,y_Ru
        y_RungeK_4_1 = y_Rung
        y_RungeK_4_2 = y_RungeK_4_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6;   // Runge-Kutta Ordnung vier Methode fuer y_2
    }                                                           // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
}                                                               // Ende der Hauptfunktion
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$ g++ DGL_2.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$ ./a.out
# 0: Index i
# 1: t-Wert
# 2: Euler Methode y1
# 3: Euler Methode y2
# 4: Runge-Kutta Ordnung vier y1
# 5: Runge-Kutta Ordnung vier y2
# 6: Analytische Loesung y1
# 7: Analytische Loesung y2
  0   0.000000000000000   1.000000000000000   1.000000000000000   1.000000000000000   1.000000000000000   1.000000000000000   1.000000000000000
  1   0.010000000000000   1.040000000000000   1.010000000000000   1.040608426315013   1.010558940181763   1.040608427569041   1.010558941425456
  2   0.020000000000000   1.081195946197052   1.021097006868275   1.082468186487541   1.022272409932680   1.082468189115352   1.022272412539396
  3   0.030000000000000   1.123637330492212   1.033343871369918   1.125632778808522   1.035197418735856   1.125632782938618   1.035197422833657
  4   0.040000000000000   1.167375849311085   1.046795999898338   1.170158168240702   1.049393799070299   1.170158174010907   1.049393804796552
  5   0.050000000000000   1.212465508925469   1.061511473409989   1.216102907913251   1.064924341689591   1.216102915471323   1.064924349191579
  6   0.060000000000000   1.258962735934773   1.077551171969109   1.263528266739016   1.081854937478282   1.263528276243294   1.081854946913853
  7   0.070000000000000   1.306926493163091   1.094978905171568   1.312498363466237   1.100254726211197   1.312498375086330   1.100254737749452
  8   0.080000000000000   1.356418401240555   1.113861548730063   1.363080307492500   1.120196252557172   1.363080321410022   1.120196266379195
  9   0.090000000000000   1.407502866150967   1.134269187516644   1.415344346785449   1.141755629685859   1.415344363194790   1.141755645985495
 10   0.100000000000000   1.460247213041785   1.156275265373008   1.469364023272402   1.165012710854272   1.469364042381559   1.165012729838951
 11   0.110000000000000   1.514721826607265   1.179956742014139   1.525216336079555   1.190051269368648   1.525216358111003   1.190051291260258
 89   0.890000000000000  30.930880042824768  30.063568069606163  34.335278279597091  33.513116889352368  34.335285835073890  33.513124438401135
 90   0.900000000000000  32.306838455932400  31.516766532214152  35.919826655719859  35.176963303871311  35.919834678006687  35.176971319508823
 91   0.910000000000000  33.747878176687038  33.040117636018252  37.581811296515667  36.923393059891680  37.581819813513697  36.923401570012345
 92   0.920000000000000  35.257179970562206  34.636996585061937  39.325128086735063  38.756498858038242  39.325137127976433  38.756507892167264
 93   0.930000000000000  36.838082119087382  36.310942423138201  41.153871130268442  40.680578010174486  41.153880727010005  40.680587599561534
 94   0.940000000000000  38.494088264736298  38.065666064560588  43.072342882456958  42.700142759460462  43.072353067779481  42.700152937178956
 95   0.950000000000000  40.228875647423045  39.905058721679517  45.085064801277419  44.819931124470799  45.085075610190323  44.819941925522599
 96   0.960000000000000  42.046303752173394  41.833200748846942  47.196788543991907  47.044918294100583  47.196800013543999  47.044929755526724
 97   0.970000000000000  43.950423388516661  43.854370923514082  49.412507737215932  49.380328601353476  49.412519906612403  49.380340762351139
 98   0.980000000000000  45.945486223170619  45.973056186177487  51.737470349791899  51.831648105542975  51.737483260518005  51.831661007589204
 99   0.990000000000000  48.035954788670189  48.193961861970656  54.177191699361046  54.404637813947794  54.177205395312626  54.404651500929958
100   1.000000000000000  50.226512991722970  50.522022387834483  56.737468125110773  57.105347575549835  56.737482652732375  57.105362093903814
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$
```

Differentialgleichungen zweiter Ordnung

Wir betrachten nun das numerische Lösen einer Differentialgleichung höherer Ordnung (zunächst allgemein der Ordnung m) mit folgender Form

$$y^{(m)}(t) = \frac{d^m y(t)}{dt^m} = f(t, y, \dot{y}, \ddot{y}, \dots, y^{(m-1)}) \ , \quad a \leq t \leq b \quad ,$$

bei gegebener Anfangskonfiguration

$$y(a) = \alpha_1 \, , \ \dot{y}(a) = \alpha_2 \, , \ \dots \ , y^{(m-1)}(a) = \alpha_m \quad .$$

Man kann nun diese DGL von Ordnung m in ein System von m Differentialgleichungen umschreiben und dieses dann numerisch lösen. Wir machen zunächst die folgende Variablenumbenennung ($u_1(t) = y(t)$, $u_2(t) = \dot{y}(t)$, $u_3(t) = \ddot{y}(t)$, ... $u_m(t) = y^{(m-1)}(t)$) und definieren das System von DGLs wie folgt:

$$\begin{aligned}
\dot{u_1}(t) &= \frac{du_1}{dt} = \frac{dy}{dt} = u_2(t) \\
\dot{u_2}(t) &= \frac{du_2}{dt} = \frac{d\dot{y}}{dt} = \ddot{y}(t) = u_3(t) \\
\dot{u_3}(t) &= \frac{du_3}{dt} = \frac{d\ddot{y}}{dt} = \dots \\
\dots &= \dots \\
\dot{u}_{m-1}(t) &= \frac{dy^{(m-2)}}{dt} = y^{(m-1)}(t) = u_m(t) \\
\dot{u}_m(t) &= \frac{dy_{(m-1)}}{dt} = y^{(m)}(t) = f(t, y, \dot{y}, \ddot{y}, \dots, y^{(m-1)}) = f(t, u_1, u_2, u_3, \dots, u_m) \quad ,
\end{aligned}$$

wobei die zeitliche Entwicklung der Vektorfunktion $\vec{u}(t) = (u_1(t), u_2(t), \dots, u_m(t))$ in den Grenzen $a \leq t \leq b$ gesucht wird. Es wird hierbei wieder vorausgesetzt, dass die Funktion $f(t, y, \dot{y}, \ddot{y}, \dots, y^{(m-1)})$ auf einer Teilmenge $\mathcal{D}$ ($\mathbb{R}^{m+1} \supseteq \mathcal{D}$) kontinuierlich definiert ist und das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $\vec{u}(t)$ existiert. Bei gegebener Anfangskonfiguration

$$u_1(a) = y(a) = \alpha_1 \, , \ u_2(a) = \dot{y}(a) = \alpha_2 \, , \ \dots \ , u_m(a) = y^{(m-1)}(a) = \alpha_m$$

ist es dann numerisch möglich, das System von gekoppelten DGLs zu lösen und somit das zeitliche Verhalten der Funktion $y(t) = u_1(t)$ zu simulieren.

Differentialgleichungen m-ter Ordnung

Wir betrachten nun das numerische Lösen einer Differentialgleichung höherer Ordnung (zunächst allgemein der Ordnung m) mit folgender Form

$$y^{(m)}(t) = \frac{d^m y(t)}{dt^m} = f(t, y, \dot{y}, \ddot{y}, \ldots, y^{(m-1)}) \;, \quad a \leq t \leq b \quad ,$$

bei gegebener Anfangskonfiguration

$$y(a) = \alpha_1 \,, \; \dot{y}(a) = \alpha_2 \,, \; \ldots \; , y^{(m-1)}(a) = \alpha_m \quad .$$

Man kann nun diese DGL von Ordnung m in ein System von m Differentialgleichungen umschreiben und dieses dann numerisch lösen. Wir machen zunächst die folgende Variablenumbenennung ($u_1(t) = y(t)$, $u_2(t) = \dot{y}(t)$, $u_3(t) = \ddot{y}(t)$, ... $u_m(t) = y^{(m-1)}(t)$) und definieren das System von DGLs wie folgt:

$$
\begin{aligned}
\dot{u_1}(t) &= \frac{du_1}{dt} = \frac{dy}{dt} = u_2(t) \\
\dot{u_2}(t) &= \frac{du_2}{dt} = \frac{d\dot{y}}{dt} = \ddot{y}(t) = u_3(t) \\
\dot{u_3}(t) &= \frac{du_3}{dt} = \frac{d\ddot{y}}{dt} = \ldots \\
\ldots &= \ldots \\
\dot{u}_{m-1}(t) &= \frac{dy^{(m-2)}}{dt} = y^{(m-1)}(t) = u_m(t) \\
\dot{u}_m(t) &= \frac{dy_{(m-1)}}{dt} = y^{(m)}(t) = f(t, y, \dot{y}, \ddot{y}, \ldots, y^{(m-1)}) = f(t, u_1, u_2, u_3, \ldots, u_m) \quad ,
\end{aligned}
$$

Umschreiben der Differentialgleichung m-ter Ordnung in ein System von m DGLs erster Ordnung

wobei die zeitliche Entwicklung der Vektorfunktion $\vec{u}(t) = (u_1(t), u_2(t), \ldots, u_m(t))$ in den Grenzen $a \leq t \leq b$ gesucht wird. Es wird hierbei wieder vorausgesetzt, dass die Funktion $f(t, y, \dot{y}, \ddot{y}, \ldots, y^{(m-1)})$ auf einer Teilmenge $\mathcal{D}$ ($\mathbb{R}^{m+1} \supseteq \mathcal{D}$) kontinuierlich definiert ist und das so definierte Anfangswertproblem "well-posed" ist und eine eindeutige Lösung $\vec{u}(t)$ existiert. Bei gegebener Anfangskonfiguration

$$
u_1(a) = y(a) = \alpha_1 \, , \; u_2(a) = \dot{y}(a) = \alpha_2 \, , \; \ldots \; , u_m(a) = y^{(m-1)}(a) = \alpha_m
$$

ist es dann numerisch möglich, das System von gekoppelten DGLs zu lösen und somit das zeitliche Verhalten der Funktion $y(t) = u_1(t)$ zu simulieren.

Beispiel: Numerische Lösung einer Differentialgleichung zweiter Ordnung ($m = 2$)

Wir betrachten speziell die folgende Differentialgleichung zweiter Ordnung ($m = 2$):

$$\ddot{y} - 2\dot{y} + 2y = e^{2t}\sin(t) \quad \text{bzw.} \quad \ddot{y} = 2\dot{y} - 2y + e^{2t}\sin(t) =: f(t, y, \dot{y})$$

Wir machen die vorgegebene Variablenumbenennung ($u_1(t) = y(t)$, $u_2(t) = \dot{y}(t)$) und definieren das System von DGLs wie folgt:

$$\dot{u}_1(t) = \frac{du_1}{dt} = \frac{dy}{dt} = u_2(t)$$

$$\dot{u}_2(t) = \frac{du_2}{dt} = \frac{d\dot{y}}{dt} = \ddot{y}(t) = 2u_2(t) - 2u_1(t) + e^{2t}\sin(t) =: f(t, u_1, u_2)$$

Wir berechnen nun die numerische Lösung $\vec{u}(t) = (u_1(t), u_2(t))$ im Zeitintervall $t \in [0, 1]$ bei vorgegebener Anfangsbedingung

$$u_1(0) = y(0) = \alpha_1 = -0.4 \;, \quad u_2(0) = \dot{y}(0) = \alpha_2 = -0.6 \quad .$$

C++ Programm: Implementierung der DGL bestimmenden Funktionen

und analytische Lösungen zum Vergleich

DGL_2_a.cpp

```cpp
/* Berechnung der Lösung einer Differentialgleichung zweiter Ordnung ( y''= 2*y' - 2*y + exp(2*t)*sin(t) )
 * welches in ein System von zwei Differentialgleichung umgeschrieben wurde
 * Form des DGL Systems: u_1'=y'=f_1(t,u_1,u_2)=u_2 , u_2'=y''=f_2(t,u_1,u_2)
 * Loesung mittels der Euler Methode und Runge-Kutta Ordnung vier Methode
 * Zeitentwicklung von u_1(t), u_2(t) fuer unterschiedliche t-Werte in [a,b]
 *  Ausgabe zum Plotten mittels Python Jupyter Notebook DGL_2.ipynb: "./a.out >> DGL_2_a.dat
 */
#include <stdio.h>             // Standard Input- und Output Bibliothek in C, z.B. printf(...)
#include <cmath>               // Bibliothek für mathematisches (e-Funktion, Betrag, ...)

double f_1(double t, double u_1, double u_2){                        // Deklaration und Definition der Funktion f_1(t,u_1,u_2)
    double wert;
    wert = u_2;                                                       // Eigentliche Definition der Funktion
    return wert;                                                      // Rueckgabewert der Funktion f_1
}                                                                     // Ende der Funktion f_1

double f_2(double t, double u_1, double u_2){                        // Deklaration und Definition der Funktion f_2(t,u_1,u_2)
    double wert;
    wert = 2*u_2 - 2*u_1 + exp(2*t)*sin(t);                           // Eigentliche Definition der Funktion
    return wert;                                                      // Rueckgabewert der Funktion f_2
}                                                                     // Ende der Funktion f_2

double u_1_analytisch(double t, double alpha_1, double alpha_2){      // Analytische Loesung y(t)=u_1(t)
    double wert;                                                      // bei gegebenem Anfangswert y(a)=1, y'(a)=1
    wert = exp(t)*sin(t)*(alpha_2 + 1.0/5.0 - alpha_1) + exp(t)*cos(t)*(alpha_1 + 2.0/5.0) + ((sin(t) - 2*cos(t))*exp(2*t))/5.0;
    return wert;                                                      // Rueckgabewert
}                                                                     // Ende der Definition

double u_2_analytisch(double t, double alpha_1, double alpha_2){      // Analytische Loesung y'(t)=u_2(t)
    double wert;                                                      // bei gegebenem Anfangswert y(a)=1, y'(a)=1
    wert = ((-3*cos(t) + 4*sin(t))*exp(2*t))/5.0 - 2*((alpha_1 - alpha_2/2.0 + 1.0/10.0)*sin(t) - cos(t)*(alpha_2 + 3.0/5.0)/2.0)*exp(t);
    return wert;                                                      // Rueckgabewert
}                                                                     // Ende der Definition
```

main()-Programm Zur Lösung des Systems bestehend aus zwei DGLs erster Ordnung

```
int main(){                                                              // Hauptfunktion
    double a = 0;                                                        // Untergrenze des Zeit-Intervalls [a,b] in dem die Loesung berechnet werden soll
    double b = 1;                                                        // 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_1 = -0.4;                                               // 1.Anfangswert bei t=a: u_1(a)=alpha_1
    double alpha_2 = -0.6;                                               // 2.Anfangswert bei t=a: u_2(a)=alpha_2
    double t;                                                            // Aktueller Zeitwert
    double u_Euler_1 = alpha_1;                                          // Deklaration und Initialisierung der numerischen Loesung der Euler Methode fuer u_1
    double u_RungeK_4_1  =alpha_1;                                       // Deklaration und Initialisierung der numerischen Loesung der Runge-Kutta Ordnung vier Method
    double k1_1,k2_1,k3_1,k4_1;                                          // Deklaration der vier Runge-Kutta Parameter fuer u_1
    double u_Euler_2 = alpha_2;                                          // Deklaration und Initialisierung der numerischen Loesung der Euler Methode fuer u_2
    double u_RungeK_4_2 = alpha_2;                                       // Deklaration und Initialisierung der numerischen Loesung der Runge-Kutta Ordnung vier Method
    double k1_2,k2_2,k3_2,k4_2;                                          // Deklaration der vier Runge-Kutta Parameter fuer u_2
    double tmp;                                                          // Variable zum Zwischenspeichern von Ergebnissen

    printf("# 0: Index i \n# 1: t-Wert \n# 2: Euler Methode y=u1 \n# 3: Euler Methode y'=u2 \n# 4: Runge-Kutta Ordnung vier y=u1 \n");
    printf("# 5: Runge-Kutta Ordnung vier y'=u2 \n# 6: Analytische Loesung y=u1 \n# 7: Analytische Loesung y'=u2 \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 %19.15f %19.15f ",i, t, u_Euler_1, u_Euler_2,u_RungeK_4_1, u_RungeK_4_2); // Ausgaben der Loesungen
        printf(" %19.15f %19.15f \n",u_1_analytisch(t,alpha_1,alpha_2),u_2_analytisch(t,alpha_1,alpha_2));          // Ausgaben der Loesungen

        tmp = u_Euler_1 + h*f_1(t,u_Euler_1,u_Euler_2);                  // Euler Methode
        u_Euler_2 = u_Euler_2 + h*f_2(t,u_Euler_1,u_Euler_2);            // u_2
        u_Euler_1 = tmp;                                                 // u_1

        k1_1 = h*f_1(t,u_RungeK_4_1,u_RungeK_4_2);                      // Runge-Kutta Parameter k1 fuer u_1
        k1_2 = h*f_2(t,u_RungeK_4_1,u_RungeK_4_2);                      // Runge-Kutta Parameter k1 fuer u_2
        k2_1 = h*f_1(t+h/2,u_RungeK_4_1+k1_1/2,u_RungeK_4_2+k1_2/2);    // Runge-Kutta Parameter k2 fuer u_1
        k2_2 = h*f_2(t+h/2,u_RungeK_4_1+k1_1/2,u_RungeK_4_2+k1_2/2);    // Runge-Kutta Parameter k2 fuer u_2
        k3_1 = h*f_1(t+h/2,u_RungeK_4_1+k2_1/2,u_RungeK_4_2+k2_2/2);    // Runge-Kutta Parameter k3 fuer u_1
        k3_2 = h*f_2(t+h/2,u_RungeK_4_1+k2_1/2,u_RungeK_4_2+k2_2/2);    // Runge-Kutta Parameter k3 fuer u_2
        k4_1 = h*f_1(t+h,u_RungeK_4_1+k3_1,u_RungeK_4_2+k3_2);          // Runge-Kutta Parameter k4 fuer u_1
        k4_2 = h*f_2(t+h,u_RungeK_4_1+k3_1,u_RungeK_4_2+k3_2);          // Runge-Kutta Parameter k4 fuer u_2
        u_RungeK_4_1 = u_RungeK_4_1 + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6; // Runge-Kutta Ordnung vier Methode fuer u_1
        u_RungeK_4_2 = u_RungeK_4_2 + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6; // Runge-Kutta Ordnung vier Methode fuer u_2
    }                                                                    // Ende for-Schleife ueber die einzelnen Punkte des t-Intervalls
}                                                                        // Ende der Hauptfunktion
```

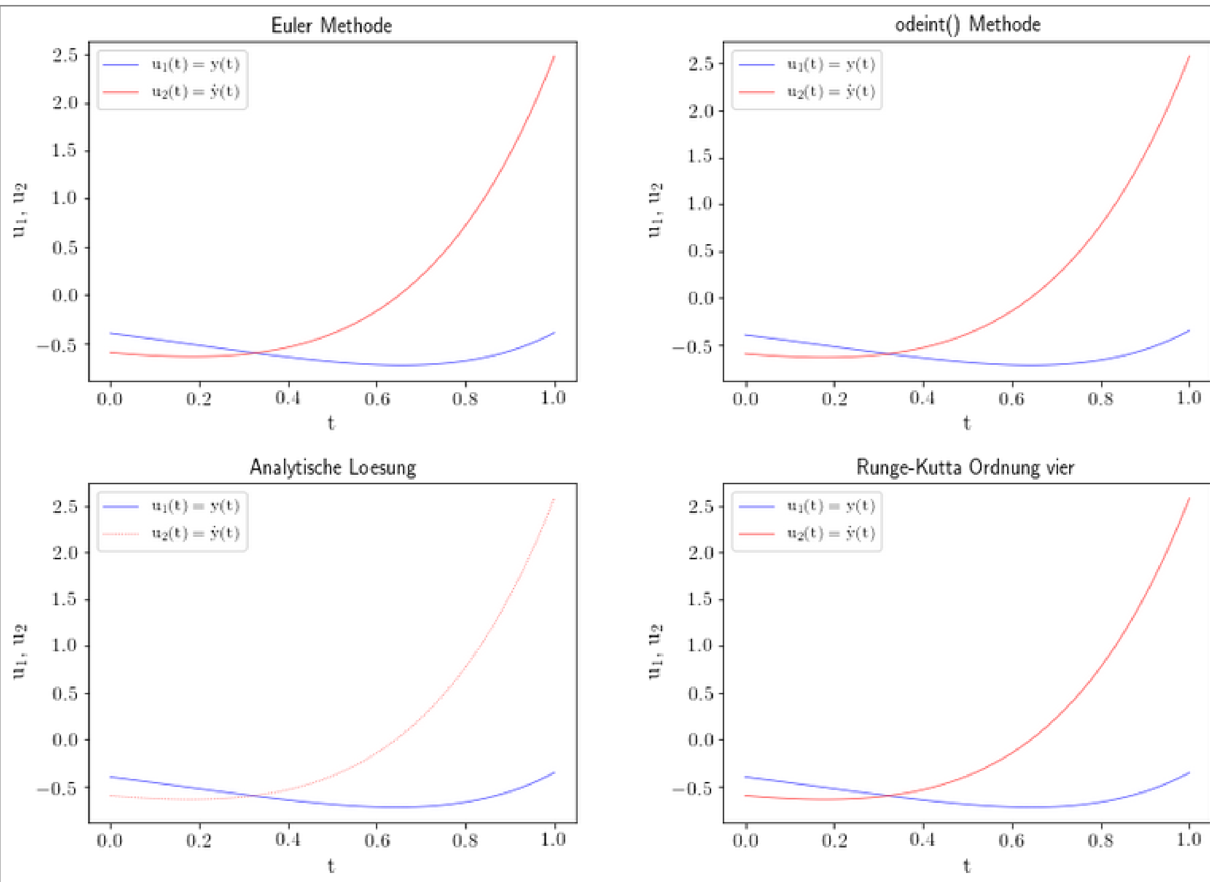
```
int main(){
    double a = 0;
    double b = 1;
    int N = 100;
    double h = (b - a)/N;
    double alpha_1 = -0.4;
    double alpha_2 = -0.6;
    double t;
    double u_Euler_1 = alpha
    double u_RungeK_4_1  =al
    double k1_1,k2_1,k3_1,k4
    double u_Euler_2 = alpha
    double u_RungeK_4_2 = al
    double k1_2,k2_2,k3_2,k4
    double tmp;

    printf("# 0: Index i \n#
    printf("# 5: Runge-Kutta

    for(int i=0; i <= N; ++i
        t = a + i*h;                                         // Zeit-Parameter wird um h erhoeht
        printf("%3d %19.15f %19.15f %19.15f %19.15f %19.15f ",i, t, u_Euler_1, u_Euler_2,u_RungeK_4_1, u_RungeK_4_2); // Ausgaben der Loesungen
        printf(" %19.15f %19.15f \n",u_1_analytisch(t,alpha_1,alpha_2),u_2_analytisch(t,alpha_1,alpha_2));          // Ausgaben der Loesungen

        tmp = u_Euler_1 + h*f_1(t,u_Euler_1,u_Euler_2);                   // Euler Methode
        u_Euler_2 = u_Euler_
        u_Euler_1 = tmp;

        k1_1 = h*f_1(t,u_Run
        k1_2 = h*f_2(t,u_Run
        k2_1 = h*f_1(t+h/2,u
        k2_2 = h*f_2(t+h/2,u
        k3_1 = h*f_1(t+h/2,u
        k3_2 = h*f_2(t+h/2,u
        k4_1 = h*f_1(t+h,u_R
        k4_2 = h*f_2(t+h,u_R
        u_RungeK_4_1 = u_Run
        u_RungeK_4_2 = u_Run
    }
}
```

```
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$ g++ DGL_2_a.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$ ./a.out
# 0: Index i
# 1: t-Wert
# 2: Euler Methode y=u1
# 3: Euler Methode y'=u2
# 4: Runge-Kutta Ordnung vier y=u1
# 5: Runge-Kutta Ordnung vier y'=u2
# 6: Analytische Loesung y=u1
# 7: Analytische Loesung y'=u2
  0   0.000000000000000  -0.400000000000000  -0.600000000000000  -0.400000000000000  -0.600000000000000   -0.400000000000000  -0.600000000000000
  1   0.010000000000000  -0.406000000000000  -0.604000000000000  -0.406019763483969  -0.603928723538745   -0.406019763480423  -0.603928723536853
  2   0.020000000000000  -0.412040000000000  -0.607857981566324  -0.412078082044877  -0.607709709178302   -0.412078082037698  -0.607709709174595
  3   0.030000000000000  -0.418118579815663  -0.611566192920012  -0.418173438715645  -0.611334998889101   -0.418173438704745  -0.611334998883659
  4   0.040000000000000  -0.424234241744863  -0.615116641998630  -0.424304235726955  -0.614796389290806   -0.424304235712248  -0.614796389283716
  5   0.050000000000000  -0.430385408164850  -0.618501090718012  -0.430468792024694  -0.618085425824015   -0.430468792006092  -0.618085425815371
  6   0.060000000000000  -0.436570419072030  -0.621711049125200  -0.436665340728562  -0.621193396812236   -0.436665340705978  -0.621193396802138
  7   0.070000000000000  -0.442787529563282  -0.624737769441126  -0.442892026530753  -0.624111327412678   -0.442892026504100  -0.624111327401232
  8   0.080000000000000  -0.449034907257693  -0.627572239991585  -0.449146903033572  -0.626829973454418   -0.449146903002762  -0.626829973441739
  9   0.090000000000000  -0.455310629657609  -0.630205179025015  -0.455427930024874  -0.629339815162493   -0.455427929989821  -0.629339815148703
 10   0.100000000000000  -0.461612681447859  -0.632627028415642  -0.461732970690161  -0.631631050766446   -0.461732970650777  -0.631631050751672
 11   0.110000000000000  -0.467938951732015  -0.634827947250508  -0.468059788760183  -0.633693589991885   -0.468059788716384  -0.633693589976263
```

```
 85   0.850000000000000  -0.639660862882018   1.054523823695142  -0.616891001612558   1.114431821296771  -0.622591325165712   1.122347719338161
 86   0.860000000000000  -0.629115624645067   1.129532211568419  -0.605363775572507   1.191431165979944  -0.610983572063222   1.199621323869462
 87   0.870000000000000  -0.617820302529383   1.207027101931975  -0.593053919229975   1.270968277427929  -0.598590426648817   1.279436581086842
 88   0.880000000000000  -0.605750031510063   1.287070494442404  -0.579935740997543   1.353106145307971  -0.585386158364030   1.361856502970104
 89   0.890000000000000  -0.592879326565639   1.369725619771776  -0.565982913189885   1.437909002519349  -0.571344400331044   1.446945344392321
 90   0.900000000000000  -0.579182070367921   1.455056957596302  -0.551168459501741   1.525442343150815  -0.556438136834108   1.534768621066005
 91   0.910000000000000  -0.564631500791958   1.543130254681930  -0.535464742305851   1.615772940527428  -0.540639690621037   1.625393127578488
 92   0.920000000000000  -0.549200198245139   1.634012543061608  -0.518843449770017   1.708968865341206  -0.523920710023936   1.718886955510981
 93   0.930000000000000  -0.532860072814523   1.727772158298681  -0.501275582792442   1.805099503859777  -0.506252155898339   1.815319511635476
 94   0.940000000000000  -0.515582351231536   1.824478757830605  -0.482731441754640   1.904235576206921  -0.487604288380014   1.914761536183362
 95   0.950000000000000  -0.497337563653230   1.924203339386915  -0.463180613091184   2.006449154708622  -0.467946653458753   2.017285121179401
 96   0.960000000000000  -0.478095530259361   2.027018259475069  -0.442591955675716   2.111813682297923  -0.447248069368521   2.122963728834351
 97   0.970000000000000  -0.457825347664610   2.132997251927500  -0.420933587022628   2.220403990971623  -0.425476612793413   2.231872209989260
 98   0.980000000000000  -0.436495375145335   2.242215446502930  -0.398172869303950   2.332296320291499  -0.402599604888941   2.344086822604119
 99   0.990000000000000  -0.414073220680306   2.354749387534646  -0.374276395181044   2.447568335922434  -0.378583597118249   2.459685250283272
100   1.000000000000000  -0.390525726804959   2.470677052618158  -0.349209973450754   2.566299148199512  -0.353394356902915   2.57874662082961
(base) hanauske@hanauske-Aspire-A717-72G:~/PPROG/EigProg/V9$
```

Visualisierung und numerische Lösung mittels Python

Beim Klicken auf das untere rechte Bild gelangen Sie zu dem Jupyter Notebook DGL_2.ipynb in dem eine Visualisierung der Ergebnisse des oberen C++ Programms programmiert ist. Die untere Abbildung auf der linken Seite zeigt z.B. die Visualisierung der Daten von DGL_2.cpp. Zusätzlich wird in dem Notebook auch die numerische Lösung direkt in Python generiert (Methode "integrate.odeint()" im Python-Modul "scipy") und mit den simulierten Daten der unterschiedlichen Verfahren verglichen.



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 von Differentialgleichungen

Systeme von gekoppelten Differentialgleichungen und Differentialgleichungen zweiter Ordnung

Im Jupyter Notebook DGL_1.ipynb haben wir einige in Python implementierte Lösungsmethoden für Differentialgleichungen erster Ordnung kennengelernt. In diesem Notebook werden wir uns zunächst mit Systemen von gekoppelten Differentialgleichungen erster Ordnung befassen und dann das numerische Lösen von Differentialgleichungen zweiter Ordnung vorstellen.

Systeme von gekoppelten Differentialgleichungen

Wir betrachten zunächst das numerische Lösen eines Systems von m-gekoppelten Differentialgleichungen (DGLs) erster Ordnung der Form

$$\dot{y}_1(t) = \frac{dy_1}{dt} = f_1(t, y_1, y_2, \dots, y_m)$$
$$\dot{y}_2(t) = \frac{dy_2}{dt} = f_2(t, y_1, y_2, \dots, y_m)$$
$$\dot{y}_3(t) = \dots =$$

Studentische Projekte

Das Foucaultsche Pendel

Im Jahre 1851 gelang Jean Bernard Léon Foucault ein anschaulicher Beweis der Erdrotation. Aufgrund der, in rotierenden Bezugssystemen auftretenden Coriolisbeschleunigung, dreht sich die Schwingungsebene des Pendels langsam. Dieses Projekt ist ein Anwendungsfall der Newtonschen Mechanik in bewegten Bezugssystemen (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I3. Seite 18]) und das zugrundeliegende System von drei gekoppelten Differentialgleichungen zweiter Ordnung gilt es numerisch mittels eines C++ Programmes zu lösen und die berechneten Daten mittels Python zu visualisiern.

Das periodisch angetriebene Pendel

Das Projekt *periodisch angetriebenes Pendel* ist ein Anwendungsfall aus der klassischen Mechanik (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel VII27. Seite 496]). Das System besteht aus einem Pendel, auf welches zusätzlich eine äußere Kraft mit periodischer Zeitabhängigkeit wirkt. Außerdem soll das Pendel durch eine geschwindigkeitsabhängige Luftreibung gedämpft sein. Die zugrundeliegende Bewegungsgleichung des periodisch angetriebenen Pendels ist stark nichtlinear und die, nur auf numerischem Weg zu berechnenden Lösungen, zeigen deterministisch chaotische Bewegungen.

Das Doppelpendel

Das Projekt *Doppelpendel* ist ein Anwendungsfall aus der klassischen Mechanik (siehe untere Animation). Das System besteht aus zwei miteinander verbundenen Pendeln, wobei wir die Luftreibung zunächst vernachlässigen. Die Herleitung der Bewegungsgleichungen des Doppelpendels erfolgt am elegantesten mittels der Euler-Lagrange Gleichungen, bzw. mittels der Hamilton Theorie. Mittels der Lagrange-Gleichungen gelangt man zu zwei gekoppelten Differentialgleichungen zweiter Ordnung, die man dann in ein System von vier gekoppelten DGLs erster Ordnung umschreiben muss um es numerisch lösen zu können. Die zugrundeliegende Bewegungsgleichung des Doppelpendels ist stark nichtlinear, sodass kleine Abänderungen in den Anfangswerten, nach einiger Zeit qualitativ unterschiedliche Bewegungen zur Folge haben (deterministisches Chaos).

Die schwingende Kette

Das Projekt *Die schwingende Kette* ist ein zentrales Beispiel eines schwingenden Massensystems, ein Problem aus der klassischen Mechanik (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I7. Seite 76]). Das System besteht aus einem masselosen Faden, der mit $N+1$ Massenpunkten (Dingen, Teilchen, Perlen) besetzt ist. Die $N+1$ 'Perlen' sollen in diesem Projekt als Objekte einer Klasse programmiert werden. Jede Perle besitzt eine ganzzahlige Instanzvariable n (die Nummer der Perle) und die erste ($n=0$) und letzte Perle ($n=N+1$) wird so befestigt, dass sie sich nicht im Raum bewegen können. Die Bewegungsgleichung der *schwingenden Kette* stellen ein System von N Differentialgleichungen zweiter Ordnung dar. Es sollen die Spezialfälle $N=2$ und $N=3$ mit einem C++ Programm simuliert werden um dann durch Erhöhung der Teilchenzahl den Kontinuums-Grenzwert der *schwingenden Saite* (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I8. Seite 88]) approximativ abbilden zu können.

Planetenbewegungen

In diesem Projekt wird die Bewegung eines Körpers in einem gravitativem Zentralkraftfeld untersucht. Die Bewegungsgleichung eines Massenpunktes (Venus, Erde) im Gravitationsfeld der Sonne ist ein oft behandeltes System in der klassischen Mechanik (siehe z.B. Walter Greiner, 'Mechanik Teil 1' [5. Auflage, 1989, Kapitel 26. Seite 266])). Die Bewegungsgleichungen des betrachteten Systems in drei Dimensionen, formuliert mittels der *Newtonschen Gravitationstheorie*, stellen ein System von drei gekoppelten Differentialgleichungen 2. Ordnung dar, die man dann in ein System von sechs gekoppelten DGLs erster Ordnung umschreiben muss um es numerisch lösen zu können.

Räuber-Beute Simulationen

Das Projekt *Räuber Beute Simulation* hat Ihren Ursprung im Fachgebiet der Populationsökologie (bzw. Populations-/Evolutionsbiologie). Die Räuber-Beute Gleichung beschreibt den natürlichen Überlebenskampf mehrerer Spezies, die einander auffressen. Die Population der Räuberwesen ernährt sich von der Population der Beutewesen und sinkt die Anzahl der Beutewesen, so erniedrigt sich die Reproduktionsrate der Räuberwesen, da diese Hunger erleiden müssen. Das Projekt besteht aus zwei formell getrennten Simulationen: Eine Agenten-ähnliche Simulation auf individueller Lebewesen Formulierung und eine Simulation, die auf Basis einer analytischen *Räuber-Beute Gleichung*.

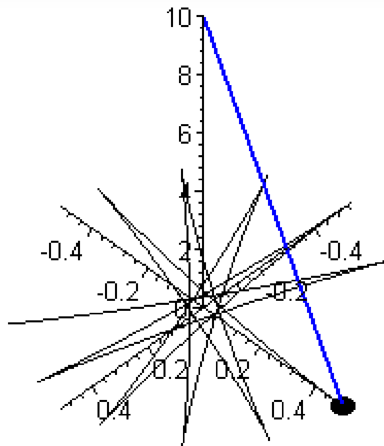
Das Foucaultsche Pendel

Das Projekt *Foucaultsche Pendel* ist ein Anwendungsfall der Newtonschen Mechanik in bewegten Bezugssystemen (siehe untere Animation). Das System besteht aus einem Pendel, welches sich auf der rotierenden Erde befindet und sich deshalb in einem rotierenden Koordinatensystem befindet (Luftreibung wird im Folgenden vernachlässigen). Betrachtet man sich die Bewegung eines Körpers (beschrieben durch $\vec{r}(t)$) in einem mit der Winkelgeschwindigkeit $\vec{\omega}$ rotierenden Koordinatensystem, so treten aufgrund der Rotation zusätzliche Komponenten in den Grundgleichungen der Mechanik auf. Die Gleichungen der Mechanik eines Körpers der Masse m auf den eine Kraft $\vec{F}$ in einem rotierenden Bezugssystem einwirkt, lauten (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I1. Seite 5]):

$$m\frac{d^2\vec{r}}{dt^2} = \vec{F} - \underbrace{m\frac{d\vec{\omega}}{dt}\times\vec{r}}_{\text{lineare Beschleunigung}} - \underbrace{2m\,\vec{\omega}\times\vec{v}}_{\text{Coriolisbeschleunigung}} - \underbrace{m\,\vec{\omega}\times(\vec{\omega}\times\vec{r})}_{\text{Zentripedalbeschleunigung}}$$

Weitere Links

- Maple Worksheet: Das Foucaultsche Pendel
- Wikipedia: Das Foucaultsche Pendel



Da sich das *Foucaultsche Pendel* auf der Erde befindet, und diese ja weitgehendst mit konstanter Winkelgeschwindigkeit rotiert ($\frac{d\vec{\omega}}{dr} \approx 0$), kann man den Term der die 'lineare Beschleunigung' aufgrund der Erdrotation beschreibt vernachlässigen. Auch ist die Winkelgeschwindigkeit der Erde klein gegenüber der Schwingungsdauer des Pendels, sodass der Term der 'Zentripedalbeschleunigung' ebenfalls klein ist, da dieser $\sim \omega^2$ geht. Die auf den Pendelkörper einwirkende Kraft besteht aus der Zugkraft $\vec{T}$ des Pendelfadens und der nach unten gerichteten Gravitationskraft ($\vec{F} = \vec{T} + m\vec{g}$). Man erhält somit in Näherung

$$m\frac{d^2\vec{r}}{dt^2} = \vec{T} + m\vec{g} - 2m\,\vec{\omega}\times\vec{v} \quad .$$

Aufgrund des Terms der 'Coriolisbeschleunigung' ist die obere Gleichung ein gekoppeltes System von drei Differentialgleichungen zweiter Ordnung, welches man zunächst in ein System aus sechs Differentailgleichungen erster Ordnung umschreiben muss, um es numerisch lösen zu können.

Mögliche Teilaufgaben des Projektes

Das Foucaultsche Pendel

Im Jahre 1851 gelang Jean Bernard Léon Foucault ein anschaulicher Beweis der Erdrotation. Aufgrund der, in rotierenden Bezugssystemen auftretenden Coriolisbeschleunigung, dreht sich die Schwingungsebene des Pendels langsam. Dieses Projekt ist ein Anwendungsfall der Newtonschen Mechanik in bewegten Bezugssystemen (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I3. Seite 18]) und das zugrundeliegende System von drei gekoppelten Differentialgleichungen zweiter Ordnung gilt es numerisch mittels eines C++ Programmes zu lösen und die berechneten Daten mittels Python zu visualisieren.

Die Gleichungen der Mechanik eines Körpers der Masse m auf den eine Kraft F in einem rotierenden Bezugssystem einwirkt, lauten

$$m \frac{d^2\vec{r}}{dt^2} = \vec{F} - \underbrace{m \frac{d\vec{\omega}}{dt} \times \vec{r}}_{\text{lineare Beschleunigung}} - \underbrace{2m\,\vec{\omega} \times \vec{v}}_{\text{Coriolisbeschleunigung}} - \underbrace{m\,\vec{\omega} \times (\vec{\omega} \times \vec{r})}_{\text{Zentripedalbeschleunigung}}$$

Da sich das *Foucaultsche Pendel* auf der Erde befindet, und diese ja weitgehendst mit konstanter Winkelgeschwindigkeit rotiert, kann man den Term der die 'lineare Beschleunigung' aufgrund der Erdrotation beschreibt vernachlässigen. Auch ist die Winkelgeschwindigkeit der Erde klein gegenüber der Schwingungsdauer des Pendels, sodass der Term der 'Zentripedalbeschleunigung' ebenfalls klein ist, da dieser quadratisch eingeht. Die auf den Pendelkörper einwirkende Kraft besteht aus der Zugkraft T des Pendelfadens und der nach unten gerichteten Gravitationskraft. Man erhält somit in Näherung das folgende System von DGLs:

$$\vec{F} = \vec{T} + m\vec{g}$$

$$m \frac{d^2\vec{r}}{dt^2} = \vec{T} + m\vec{g} - 2m\,\vec{\omega} \times \vec{v}$$

Mögliche Teilaufgaben des Projektes

Schreiben Sie zunächst die Bewegungsgleichung in ein System von erster Ordnung Differentialgleichungen um (gerne mittels Python-Jupyter + SymPy).

Lösen Sie das System aus sechs Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Betrachten Sie unterschiedliche Anfangsbedingungen und visualisieren Sie ihre Ergebnisse mittels eines Python Skriptes bzw. Jupyter Notebooks.

Die Bewegung des Foucaultschen Pendels hängt davon ab, an welcher Stelle man sich auf der Erde befindet. Zeigen Sie dies mittels einer numerischen Simulation.

Vernachlässigen Sie nun nicht mehr die Terme der 'lineare Beschleunigung' und 'Zentripedalbeschleunigung' und nehmen eine große und sich zeitlich verändernde Winkelgeschwindigkeit des rotierenden Koordinatensystems an. Wie schnell muss das System rotieren, dass die Terme nicht mehr zu vernachlässigen sind.

Wie würde die Bewegung des Pendels auf dem Mond oder auf einem Neutronenstern aussehen?

Schreiben Sie Ihr Programm so um, dass das *Foucaultsche Pendel* eine Klasse (mit eigenen Daten-Member, Member Funktionen und Konstruktoren) repräsentiert.

Das periodisch angetriebene Pendel

Das Projekt *periodisch angetriebenes Pendel* ist ein Anwendungsfall aus der klassischen Mechanik (siehe z.B. Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel VII27. Seite 496]). Das System besteht aus einem Pendel, auf welches zusätzlich eine äußere Kraft mit periodischer Zeitabhängigkeit wirkt. Außerdem soll das Pendel durch eine geschwindigkeitsabhängige Luftreibung gedämpft sein. Die zugrundeliegende Differentialgleichung besitzt die folgende Form

$$\frac{d^2\phi(t)}{dt^2} + \frac{\beta}{m}\frac{d\phi(t)}{dt} + \frac{g}{l} \cdot \sin\left(\phi(t)\right) = f_0 \cdot \cos\left(\Omega \cdot t\right) \quad ,$$

wobei m und l die Masse und Länge des Pendels, β der Stokessche Reibungskoeffizient und f_0 und Ω die Amplitude und Frequenz der äußeren Kraft ist. Die zugrundeliegende Bewegungsgleichung des periodisch angetriebenen Pendels ist stark nichtlinear und die, nur auf numerischem Weg zu berechnenden Lösungen, zeigen deterministisch chaotische Bewegungen.

Mögliche Teilaufgaben des Projektes

Schreiben Sie die Bewegungsgleichung zweiter Ordnung in ein System von zwei miteinander gekoppelten Differentialgleichungen erster Ordnung um.

Lösen Sie das System von Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Betrachten Sie unterschiedliche Anfangsbedingungen und Parameterwerte (m, l, β, f_0 und Ω) und visualisieren Sie ihre Ergebnisse mittels eines Python Skriptes bzw. Jupyter Notebooks.

Die zugrundeliegende Bewegungsgleichung des periodisch angetriebenen Pendels ist stark nichtlinear, sodass kleine Abänderungen in den Anfangswerten, nach einiger Zeit qualitativ unterschiedliche Bewegungen zur Folge haben. Fertigen Sie zwei Simulationen an, die sich in ihren Anfangswerten (z.B. der Anfangsauslenkung des Pendels $\phi(0)$) nur um einen Epsilonbetrag $\Delta\epsilon$ unterscheiden und betrachten Sie, ab wann die Pendel-Trajektorien sich qualitativ unterscheiden.

Stellen Sie die Phasenraumtrajektorien ($\phi(t)$ - $\frac{d\phi(t)}{dt}$ Diagramm) einiger Simulationen dar und vergleichen Sie diese mit den Abbildungen auf Seite 497 in Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel VII27. Seite 496].

Vernachlässigen Sie die periodisch äußere Kraft $f_0 = 0$ und simulieren Sie die Fälle: schwache Dämpfung, aperiodischer Grenzfall und überdämpftes System.

Vergleichen Sie Ihre Simulation mit der linearen Näherung des *periodisch angetriebene Pendel* (siehe z.B. Walter Greiner, 'Mechanik Teil 1' [5. Auflage, 1989, Kapitel 23. Seite 236])) und untersuchen Sie die *Resonanzkatastrophe* indem Sie Amplitude der erzwungenen gedämpften Schwingung als Funktion der Erregerfrequenz Ω darstellen.

Schreiben Sie Ihr Programm so um, dass das *periodisch angetriebene Pendel* eine Klasse (mit eigenen Daten-Member, Member Funktionen und Konstruktoren) repräsentiert.

Das Doppelpendel

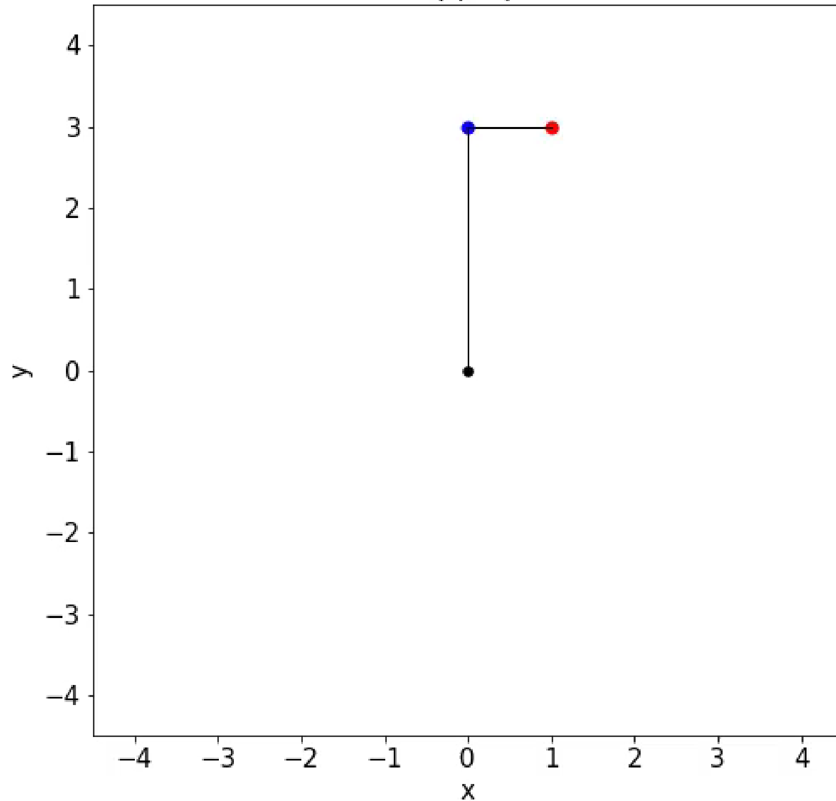
Das Projekt *Doppelpendel* ist ein Anwendungsfall aus der klassischen Mechanik (siehe untere Animation). Das System besteht aus zwei miteinander verbundenen Pendeln, wobei wir die Luftreibung zunächst vernachlässigen. Die Herleitung der Bewegungsgleichungen des Doppelpendels erfolgt am elegantesten mittels der Euler-Lagrange Gleichungen, bzw. mittels der Hamilton Theorie. In der Euler-Lagrange Theorie beschreibt man das Doppelpendel mittels zweier generalisierten Koordinaten $\theta_1(t)$ und $\theta_2(t)$, welche die Ausschläge der beiden Pendel als Pendelwinkel zur Vertikalen beschreiben. Die generalisierten Geschwindigkeiten $\dot{\theta}_1(t)$ und $\dot{\theta}_2(t)$ stellen die Winkelgeschwindigkeiten der beiden Pendel dar. Betrachtet man die Bewegung des Doppelpendels in der x-y-Ebene, so ergibt sich die folgende Lagrangefunktion des Systems (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel V15. Seite 269], beachte leicht abgeänderte Koordinatenwahl)

$$L = T - V = \frac{1}{2} m_1 \left[\dot{x}_1(t)^2 + \dot{y}_1(t)^2 \right] + \frac{1}{2} m_2 \left[\dot{x}_2(t)^2 + \dot{y}_2(t)^2 \right] - V$$

$$\text{mit:} \quad x_1 = l_1 \sin(\theta_1)\,,\ y_1 = -l_1 \cos(\theta_1) \quad,\ x_2 = l_1 \sin(\theta_1) + l_2 \sin(\theta_2)\,,\ y_2 = -l_1 \cos(\theta_1) - l_2 \cos(\theta_2)$$

$$V = V(\theta_1, \theta_2) = m_1\, g \left[l_1 + l_2 - l_1 \cos(\theta_1) \right] + m_2\, g \left[l_1 + l_2 - \left(l_1 \cos(\theta_1) + l_2 \cos(\theta_2) \right) \right] \quad ,$$

wobei m_1 und m_2 die Massen und l_1 und l_2 die Längen des Pendels darstellen.



Mittels der Lagrange-Gleichungen

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) - \frac{\partial L}{\partial \theta_1} = 0 \quad \text{und} \quad \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_2}\right) - \frac{\partial L}{\partial \theta_2} = 0$$

gelangt man zu zwei gekoppelten Differentialgleichungen zweiter Ordnung, die man dann in ein System von vier gekoppelten DGLs erster Ordnung umschreiben muss um es numerisch lösen zu können.

Mögliche Teilaufgaben des Projektes

Berechnen Sie zunächst auf analytischem Weg, mittels eines Jupyter Notebooks unter Verwendung der SymPy Bibliothek, die Euler-Lagrange Gleichungen des Doppelpendels und schreiben Sie dann die Bewegungsgleichung in ein System von erster Ordnung Differentialgleichungen um.

Lösen Sie das System von Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Betrachten Sie unterschiedliche Anfangsbedingungen und Parameterwerte (m_1, m_2, l_1 und l_2) und visualisieren Sie ihre Ergebnisse mittels eines Python Skriptes bzw. Jupyter Notebooks.

Die zugrundeliegende Bewegungsgleichung des Doppelpendels ist stark nichtlinear, sodass kleine Abänderungen in den Anfangswerten, nach einiger Zeit qualitativ unterschiedliche Bewegungen zur Folge haben. Fertigen Sie zwei Simulationen an, die sich in ihren Anfangswerten (z.B. der Anfangsauslenkung des ersten Pendels $\theta_1(0)$) nur um einen Epsilonbetrag $\Delta\epsilon$ unterscheiden und betrachten Sie sich, ab wann die Pendel-Trajektorien sich qualitativ unterscheiden.

Vernachlässigen Sie nun nicht mehr die Luftreibung und bauen Sie einen zusätzlichen Reibungsterm in die Lagrangedichte ein. Lösen die dann die resultierenden Bewegungsgleichungen numerisch.

Schreiben Sie Ihr Programm so um, dass das *Doppelpendel* eine Klasse (mit eigenen Daten-Member, Member Funktionen und Konstruktoren) repräsentiert.

Weitere Links

- Maple Worksheet: Hamilton Theorie des Doppelpendels
- Maple Worksheet: Euler-Lagrange Theorie des Doppelpendels
- Wikipedia: Das Doppelpendel

Die schwingende Kette

Das Projekt *Die schwingende Kette* ist ein zentrales Beispiel eines schwingenden Massensystems, ein Problem aus der klassischen Mechanik (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I7. Seite 76]). Das System besteht aus einem masselosen Faden, der mit $N+1$ Massenpunkten (Dingen, Teilchen, Perlen) besetzt ist. Die $N+1$ 'Perlen' sollen in diesem Projekt als Objekte einer Klasse programmiert werden. Jede Perle besitzt eine ganzzahlige Instanzvariable n (die Nummer der Perle) und die erste ($n=0$) und letzte Perle ($n=N+1$) wird so befestigt, dass sie sich nicht im Raum bewegen können. Beschreibt man das System im zweidimensionalen Raum und spannt die erste Perle bei ($x_0 = 0$,$y_0 = 0$) und die letzte Perle bei ($x_0 = L$,$y_0 = 0$) ein, so soll eine über den gesamten Faden konstante Fadenspannung T entstehen. Die Perlen seien in äquidistanten Abständen a auf dem Faden angeordnet und die Auslenkung aus der Ruhelage in y-Richtung sei relativ klein, sodass die geringfügige Auslenkung in x-Richtung zu vernachlässigbar ist. Betrachten Sie somit zunächst den Fall einer ausschließlich vertikalen Auslenkung der Teilchen.

Die y-Auslenkung des n-ten Teilchens y_n wird durch die Auslenkung des $(n-1)$-ten Teilchens y_{n-1} und des $(n+1)$-ten Teilchens y_{n+1} beeinflusst. Die rücktreibenden Kräfte $\vec{F}_{n-1}$ und $\vec{F}_{n+1}$ besitzen die folgende Form

$$\vec{F}_{n-1} = -\left(T \cdot \sin(\alpha_{n-1})\right) \vec{e}_y \underbrace{\approx}_{\text{lineare Näherung}} -T \left(\frac{y_n - y_{n-1}}{a} \right) \vec{e}_y$$

$$\vec{F}_{n+1} = -\left(T \cdot \sin(\alpha_{n+1})\right) \vec{e}_y \underbrace{\approx}_{\text{lineare Näherung}} -T \left(\frac{y_n - y_{n+1}}{a} \right) \vec{e}_y \quad ,$$

wobei α_{n-1} der mit der Horizontalen eingeschlossene Auslenkungswinkel zum $(n-1)$-ten Teilchen und α_{n+1} der Winkel zu seinem anderen Nachbarn, dem$(n-1)$-ten Teilchen ist. Die Bewegungsgleichung des n-ten Teilchens in der linearen

seinem anderen Nachbarn, dem$(n-1)$-ten Teilchen ist. Die Bewegungsgleichung des n-ten Teilchens in der linearen Näherung lautet somit:

$$m_n \frac{d^2 y_n}{dt^2}\,\vec{e}_y = \vec{F}_{n-1} + \vec{F}_{n+1} = -T\left(\frac{y_n - y_{n-1}}{a}\right)\vec{e}_y - T\left(\frac{y_n - y_{n+1}}{a}\right)\vec{e}_y \qquad \Leftrightarrow$$

$$\frac{d^2 y_n(t)}{dt^2} = \frac{T}{m_n\, a}\left(y_{n-1} - 2\,y_n + y_{n+1}\right) \quad ,$$

wobei m_n die Masse des n-ten Teilchens ist und der Index n von $n=1$ bis $n=N$ läuft. Die Bewegungsgleichung der *schwingenden Kette* stellt somit ein System von N Differentialgleichungen zweiter Ordnung dar. Die Bewegungsgleichung für die erste und letzte freie Perle ($n=1$ und $n=N$) vereinfacht sich, da die ganz äußeren Eckperlen eingespannt sind (Randbedingung: $y_0(t)=0$ und $y_{N+1}(t)=0\,,\ \forall\, t \in \mathbb{R}$)

$$\frac{d^2 y_1(t)}{dt^2} = \frac{T}{m_1\, a}\left(-2\,y_1 + y_2\right) \qquad \text{und} \qquad \frac{d^2 y_N(t)}{dt^2} = \frac{T}{m_1\, a}\left(y_{N-1} - 2\,y_N\right) \quad .$$

Mögliche Teilaufgaben des Projektes

Schreiben Sie das System von N Bewegungsgleichung in ein System von $2N$ Differentialgleichungen erster Ordnung um.

Entwerfen Sie ein C++ Programm mit einer Klasse, welche die Perlenkette als einen vector-Container bestehend aus Perlen-Objekten einer anderen Klasse (Klasse Perle) implementiert hat. Innerhalb dieser Klasse soll auch das System von Differentialgleichungen erster Ordnung mittels des Runge-Kutta Ordnung vier Verfahrens gelöst werden.

Betrachten Sie die Fälle $N = 2$ und $N = 3$ für eine Kette bestehend aus Teilchen gleicher Masse $m_n = m$ und visualisieren Sie ihre Ergebnisse mittels eines Python Skriptes bzw. Jupyter Notebooks (zum Vergleich siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel III7. Seite 84]).

Simulieren Sie nun eine Kette bestehend aus einer großen Anzahl an Perlen und vergleichen Sie Ihre Ergebnisse mit dem Kontinuums-Grenzwert der *schwingenden Saite* (siehe Walter Greiner, 'Klassische Mechanik II' [8. Auflage, 2008, Kapitel I8. Seite 88]).

Betrachten Sie den Fall ungleicher Massen an einem Beispiel.

Betrachten Sie nun den allgemeinen Fall $\vec{F}_{n-1} = -\left(T \cdot \sin(\alpha_{n-1})\right) \vec{e}_y$ und $\vec{F}_{n+1} = -\left(T \cdot \sin(\alpha_{n+1})\right) \vec{e}_y$ und beziehen zusätzlich auch die Bewegungen der Perlen in x-Richtung in Ihre Berechnungen ein. Erstellen Sie dann ein Beispiel-Simulation mit großen Perlenausschlägen und vergleichen die Ergebnisse mit den Ergebnissen der linearen Näherung. Bauen Sie auch diesen Anwendungsfall sinnvoll in die Klassenstruktur Ihres Programmes ein.

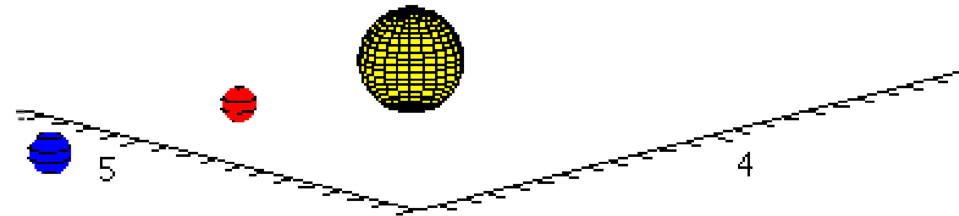
Planetenbewegungen

In diesem Projekt wird die Bewegung eines Körpers in einem gravitativem Zentralkraftfeld untersucht. Die Bewegungsgleichung eines Massenpunktes (Venus, Erde) im Gravitationsfeld der Sonne ist ein oft behandeltes System in der klassischen Mechanik (siehe z.B. Walter Greiner, 'Mechanik Teil 1' [5. Auflage, 1989, Kapitel 26. Seite 266])). Die Bewegungsgleichungen des betrachteten Systems, formuliert mittels der *Newtonschen Gravitationstheorie* lauten:

$$\frac{d^2\vec{r}(t)}{dt^2} = \frac{G \cdot M_\odot}{r^2} \cdot \frac{\vec{r}(t)}{r} \quad ,$$

wobei $\vec{r}(t)$ der Ortsvektor des betrachteten Planeten in Bezug auf unser Zentralgestirn, die Sonne (Masse $M_\odot$) und G die Gravitationskonstante ist. In drei Dimensionen ($\vec{r} = (x, y, z)$) stellt die Bewegungsgleichung ein System von drei DGLs 2. Ordnung dar.

Mögliche Teilaufgaben des Projektes



Schreiben Sie das System von Bewegungsgleichungen zweiter Ordnung in ein System von Differentialgleichungen erster Ordnung um.

Lösen Sie das System von Differentialgleichungen erster Ordnung mittels eines C++ Programms und benutzen Sie das Runge-Kutta Ordnung vier Verfahren.

Betrachten Sie die Bewegung von unterschiedlichen Planeten um die Sonne (unterschiedliche Anfangsbedingungen) und berechnen Sie z.B. die Bahn der Erde, der Venus und des Merkur. Visualisieren Sie ihre Ergebnisse mittels eines Python Skriptes bzw. Jupyter Notebooks.

Am 8.Juni 2004 um ca. 7.30 Uhr ereignete sich der Venustransit und konnte auch in Frankfurt beobachtet werden. Simulieren Sie mittels Ihres Planetenbewegung-Programmes dieses Ereignis. Als Vorlage können Sie das weiter unten angegebene Maple Worksheet Der Transits der Venus am 8.Juni 2004 (Newtonsche Version) verwenden.

Simulieren Sie nun auch den Merkurtransit, der sich am 9.Mai 2016 um ca. 13.12 Uhr in Frankfurt ereignete. Das unten angegebene Maple Worksheet Der Merkur-Transit (allgemein-relativistisch) kann Ihnen dabei vielleicht helfen, obwohl die Berechnungen hier allgemein-relativistisch formuliert wurden.

Erstellen Sie ein fiktives Planetensystem mit zumindest einem Planeten in einem kreisförmigen Orbit und einem Planeten mit einer stark elliptischen Bahn. Simulieren Sie zusätzlich die Bahn eines Asteroiden, der das Planetensystem auf einer hyperbolischen Bahn passiert.

Schreiben Sie Ihr Programm so um, dass das die *Planetenbewegungen* eine Klasse (mit eigenen Daten-Member, Member Funktionen und Konstruktoren) repräsentiert.

Projekt: Planetenbewegungen

Weitere Links

- View Maple Worksheet: Der Merkur-Transit (allgemein-relativistisch)
- Download Maple Worksheet: Der Merkur-Transit (allgemein-relativistisch)
- View Maple Worksheet: Der Transits der Venus am 8.Juni 2004 (Newtonsche Version)
- Download Maple Worksheet: Der Transits der Venus am 8.Juni 2004 (Newtonsche Version)

Räuber-Beute Simulationen

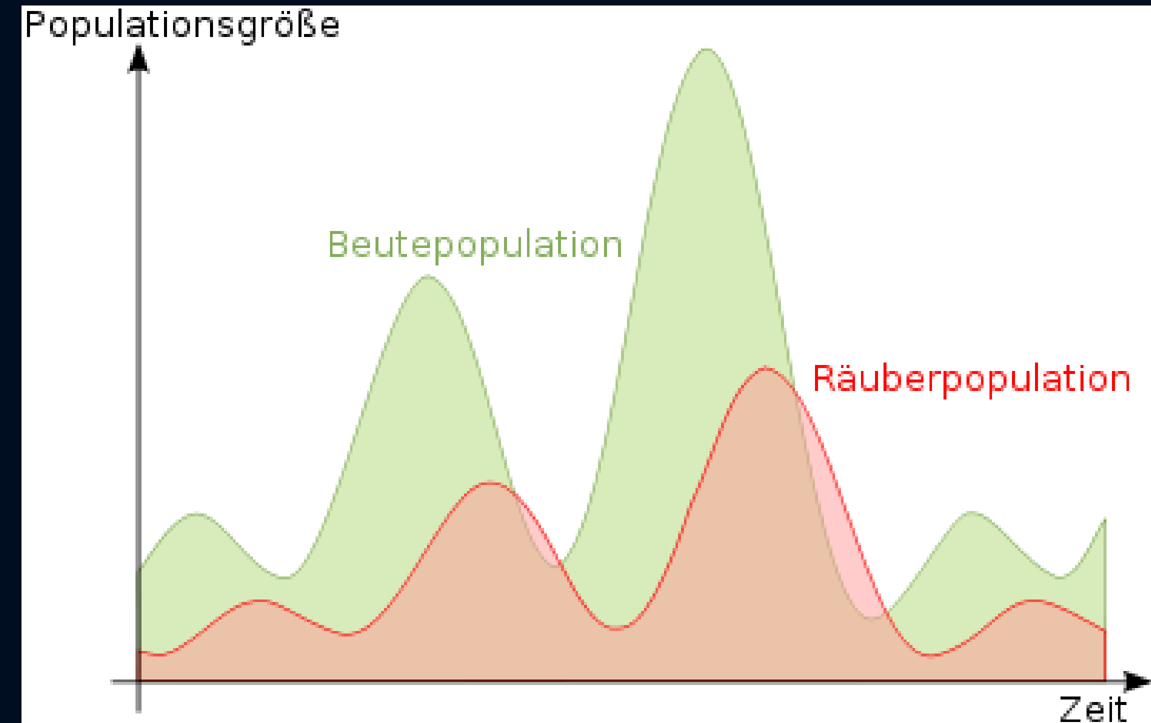
Das Projekt *Räuber Beute Simulation* hat Ihren Ursprung im Fachgebiet der Populationsökologie (bzw. Populations-/Evolutionsbiologie). In der Ökologie werden die Beziehungen/Interaktionen von Lebewesen (Organismen) untereinander und zu ihrer unbelebten Umwelt untersucht. Die zeitliche Entwicklung der einzelnen Populationsgrößen ist Gegenstand der Populations- und Evolutionsbiologie, welche ihren mathematischen Ansatz in der sogenannten *Räuber-Beute Gleichung* (Lotka-Volterra Gleichung) findet. Die Räuber-Beute Gleichung beschreibt den natürlichen Überlebenskampf mehrerer Spezies, die einander auffressen. Die Population der Räuberwesen ernährt sich von der Population der Beutewesen und sinkt die Anzahl der Beutewesen, so erniedrigt sich die Reproduktionsrate der Räuberwesen, da diese Hunger erleiden müssen. Das Projekt besteht aus zwei formell getrennten Simulationen: Eine Agenten-ähnliche Simulation auf individueller Lebewesen Formulierung und eine Simulation, die auf Basis einer analytischen *Räuber-Beute Gleichung*.

Die Räuber-Beute-Gleichung (Lotka-Volterra Gleichung) für N-Populationen besteht aus dem folgenden System von Differentialgleichungen

$$\frac{dx_i(t)}{dt} = \left(r_i + \sum_{j=1}^{N} b_{ij}\, x_j(t) \right) x_i(t) \quad ,$$

wobei $x_i(t)$ die Anzahl der Lebewesen in der Population i zur Zeit t beschreibt, r_i die Werte der intrinsischen Reproduktions-Sterberaten der Population i sind (wie viele Lebewesen der Population i werden pro Zeiteinheit dt gebohren minus die pro Zeiteinheit versterbenden Lebewesen der Population i) und b_{ij} die Interaktionsmatrix der Population i zur Population j darstellt (Erhöhung der Reproduktionsrate der Räuber pro Beutelebewesen bzw. Erniedriegung der Reproduktionsrate der Beutetiere pro Räuberlebewesen). Details finden Sie z.B. in den unten angegebenen Büchern von 1) Hofbauer und Sigmund und 2) Nowak. Das betrachtete Problem stellt ein System von N gekoppelten Differentialgleichungen erster Ordnung dar und kann mittels der in dieser Vorlesung erlernten numerischen Verfahren gelöst werden.

Das Räuber-Beute Spiel



- Der oben beschriebene deterministische Zugang zu dem betrachteten System setzt voraus, dass jedes Lebewesen der einen Population stets mit einem Lebewesen der anderen Population in Kontakt kommen kann und die Überlebenswahrscheinlichkeit des Beutewesens wird durch eine, über die gesamte Population gemittelten Wert (Eintrag in der b_{ij}-Interaktionsmatrix) beschrieben. Eine Agenten-ähnliche Simulation auf individueller Lebewesen Formulierung ist hingegen eine Art von stochastischer Simulation und jedes Lebewesen soll hierbei durch ein Objekt einer C++ Lebewesen-Klasse formuliert werden. Pro Iterationsschritt (Zeitschritt Δt) sollen die sich in einem C++ Container befindlichen einzelnen Lebewesen nun in Kontakt treten und der individuelle Überlebenskampf soll mittels einer Member-Funktion definiert werden.

Die Lotka-Volterra-Gleichung (Räuber-Beute-Gleichung) für N-Populationen

Anzahl der Räuber/Beute Wesen der i-ten Population zur Zeit t

$$\frac{dx_i(t)}{dt} = \left(r_i + \sum_{j=1}^{N} b_{ij} \, x_j(t) \right) x_i(t)$$

Reproduktions- bzw. Sterberaten

Interaktionsmatrix

Mögliche Teilaufgaben des Projektes

Entwerfen Sie ein C++ Programm, das die deterministische Räuber-Beute-Gleichung für den Fall $N = 2$ löst. Das System von zwei Differentialgleichungen erster Ordnung soll hierbei mittels des Runge-Kutta Ordnung vier Verfahrens gelöst und mit einem Python Skript bzw. Jupyter Notebook visualisiert werden. Benutzen Sie hierbei die intrinsischen Reproduktions-Sterberaten $r_1 = 0.9$ und $r_2 = -0.8$ und die folgenden Werte der Interaktionsmatrix b_{ij} ($b_{11} = -0.01$, $b_{12} = -0.02$, $b_{21} = 0.25$ und $b_{22} = -0.05$) und vergleichen Sie Ihre Ergebnisse mit dem Maple Worksheet: Die Populationsökologie (bzw. Populations-/Evolutionsbiologie) als Anwendungsfeld der evolutionären Spieltheorie.

Betrachten Sie den Fall $N = 2$ mit $r_1 = 0.8$, $r_2 = -0.5$ und $b_{11} = -0.01$, $b_{12} = -0.09$, $b_{21} = 0.4$ und $b_{22} = -0.01$) und vergleichen Sie Ihre Ergebnisse mit dem Maple Worksheet: Die Populationsökologie (bzw. Populations-/Evolutionsbiologie) als Anwendungsfeld der evolutionären Spieltheorie.

Betrachten Sie den Fall $N = 2$ und entwerfen ein C++ Programm, dass eine Agenten-ähnliche Simulation auf individueller Lebewesen Formulierung beinhaltet. Das folgende Java-Programm kann Ihnen als Vorlage dienen: Niklas Götz, Individuenbasierte Räuber-Beute-Simulation. Vergleichen Sie Ihre Ergebnisse mit den numerischen Ergebnissen der deterministischen Simulationen.

Betrachten Sie den Fall $N = 3$.

Weitere Links

- Hofbauer, Josef, and Karl Sigmund. Evolutionary games and population dynamics. Cambridge university press, 1998
- Martin A. Nowak, Evolutionary Dynamics - Exploring the Equations of Life, 2006
- Maple Worksheet: Die Populationsökologie (bzw. Populations-/Evolutionsbiologie) als Anwendungsfeld der evolutionären Spieltheorie
- Niklas Götz, Individuenbasierte Räuber-Beute-Simulation

Eigene Projektthemen sind auch möglich

Ab der nächsten Vorlesung (Vorlesung 10) werden die Studierenden dann an eigenen Projekten arbeiten, wobei viele dieser Projekte Probleme behandeln, die man nur mittels einer numerischen Lösung adäquat beschreiben kann. Neben den vorgestellten Projekten, können auch eigene Projektthemen behandelt werden. Bei Interesse können auch Projekte aus dem Bereich der [Physik der sozio-ökonomischen Systeme] (z.B. Replikatordynamik der Evolutionären Spieltheorie, Simulationen von komplexen Netzwerken), oder Projekte im Themengebiet der allgemeinen Relativitätstheorie (z.B. Bewegung um ein schwarzes Loch mittels der Geodätengleichung, näheres siehe [Allgemeine Relativitätstheorie mit dem Computer]) behandelt werden. Die studentischen Projekte können alleine oder in Gruppen (bis zu drei Personen) durchgeführt werden.