

## Numerisches Programmieren

### 4. Programmieraufgabe: Freier Fall, Planetensystem, ODEs & Diskrete Fourier-Transformation

#### Einführung: Freier Fall

Dieser Teil des Aufgabenblatts steht wieder ganz unter dem Motto “Spielerisch zur numerischen Simulation“. Wir werden uns hierbei ODEs im Zusammenhang mit punktförmigen Objekten anschauen, z. B. einer kleinen Kugel. Ein solches Objekt betrachten wir uns im freien Fall und unter der Beeinflussung von diversen anderen Kräften. Die daraus entstehenden ODEs sollen mit einfachen Zeitschrittverfahren gelöst werden.

#### Physikalische Grundlagen

Betrachten wir zuerst ein stark vereinfachtes Modell des freien Falls von einer kleinen Kugel in 2D. Der Zustand unserer vereinfachten Kugel ist für einen Zeitpunkt  $t$  eindeutig über die Position  $\vec{r}(t)$  und die momentane Geschwindigkeit  $\vec{v}(t)$  bestimmt.

Nun betrachten wir die Änderung der Position und Geschwindigkeit über einen kleinen Zeitschritt  $\Delta t$ . Hierbei gilt, dass die Positionsänderung dem Integral der Geschwindigkeit  $\vec{v}$  während des nächsten Zeitschritts entspricht:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \int_t^{t+\Delta t} \vec{v}(\tau) d\tau$$

Des Weiteren gilt auch, dass sich die Geschwindigkeitsänderung über das bestimmte Integral der Beschleunigung  $a(t)$  berechnen lässt:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \int_t^{t+\Delta t} \vec{a}(\tau) d\tau$$

Zur Vereinfachung betrachten wir nun den freien Fall mit konstanter Beschleunigung durch die Gravitation mit  $g = 9.81 \frac{m}{s^2}$  und damit  $\vec{a} = (0, -g)^T$ :

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \int_t^{t+\Delta t} \vec{a}(\tau) d\tau = \vec{v}(t) + \int_t^{t+\Delta t} \begin{pmatrix} 0 \\ -g \end{pmatrix} d\tau$$

Nun können wir uns die Geschwindigkeit zu jedem beliebigen Zeitpunkt  $t$  für gegebene Anfangswerte  $\vec{v}(0)$  exakt ausrechnen indem wir die Integrale weiter auswerten:

$$\vec{v}(t) = \vec{v}(0) + \int_0^t \begin{pmatrix} 0 \\ -g \end{pmatrix} d\tau = \vec{v}(0) + \begin{pmatrix} 0 \\ -g \end{pmatrix} t$$

Mit der Auswertung des Integrals für die Positionsänderung können wir letztendlich auch die Position in Abhängigkeit von der Zeit  $t$  berechnen:

$$\vec{r}(t) = \vec{r}(0) + \int_0^t \vec{v}(\tau) d\tau = \vec{r}(0) + \int_0^t \vec{v}(0) + \begin{pmatrix} 0 \\ -g \end{pmatrix} \tau d\tau = \vec{r}(0) + \vec{v}(0)t + \begin{pmatrix} 0 \\ -g \end{pmatrix} \frac{t^2}{2}$$

Damit liegt uns eine analytische Lösung vor, die uns für gegebene Anfangswerte zu einem beliebigen Zeitpunkt  $t$  die Position zurückgibt. Nachdem wir den freien Fall nun analytisch betrachtet haben, wollen wir im Weiteren betrachten, wie man eine solche Simulation durch numerische Verfahren approximieren kann. Leider hängt z. B. in Spielen die Simulation nicht nur von der Gravitation ab, sondern auch noch von vielen anderen Einflussgrößen (Federn, Stöße durch andere Objekte, Kollisionsimpulse, etc.) die uns eine analytische Lösung erschweren bzw. unmöglich machen.

## ODE Beispielfunktionen

Wir werden ODEs in der folgenden Form untersuchen:

$$\frac{d\vec{r}}{dt} = \vec{F}(t, \vec{r}), \quad \vec{r}(0) = \vec{r}_0 = \begin{pmatrix} r_{0,x} \\ r_{0,y} \end{pmatrix}, \quad \left. \frac{d\vec{r}}{dt} \right|_{t=0} = \vec{v}_0 = \begin{pmatrix} v_{0,x} \\ v_{0,y} \end{pmatrix}.$$

Die erste ODE ist die des freien Falls:

$$\vec{F}_1(t, \vec{r}) = \vec{F}_1(t) = \begin{pmatrix} v_{0,x} \\ v_{0,y} - g \cdot t \end{pmatrix}.$$

Des Weiteren betrachten wir auch noch drei weitere, etwas komplexere Funktionen, um die Unterschiede der Verfahren besser verstehen zu können.

Die zweite Funktion in unserer Testreihe ist mit

$$\vec{F}_2(t, \vec{r}) = \begin{pmatrix} v_{0,x} + r_x(t)^2 \cdot 0.005 \\ v_{0,y} - gt - r_x(t)^2 t^2 \cdot 0.001 \end{pmatrix}$$

gegeben, wobei  $g$  die Gravitationskonstante ist. Sie ist 2. Grades in  $r_x(t)$  und  $t!$   
 Die dritte ist 4. Grades in  $t$  und gegeben mit:

$$\vec{F}_3(t, \vec{r}) = \vec{F}_3(t) = \begin{pmatrix} v_{0,x} + 4.0 \cdot t \\ v_{0,y} - 8.0 \cdot t^2 + 0.1 \cdot t^4 \end{pmatrix}$$

Die vierte Funktion ist ein steifes Beispiel, und daher nur mit impliziten Verfahren lösbar:

$$\vec{F}_4(t, \vec{r}) = \begin{pmatrix} r_y(t) - 1 \\ -100(r_x(t) - 10) - 101r_y(t) + 1 \end{pmatrix}$$

Der Unterschied zwischen impliziten und expliziten Verfahren wird in den nächsten Vorlesungen genauer behandelt. In diesen Aufgabenblatt werden wir uns auf die expliziten Verfahren konzentrieren.

## Programmoberfläche

Als Hilfestellung und zur Veranschaulichung der Funktionen gibt es eine GUI, die in Abbildung 1 genauer beschrieben ist.

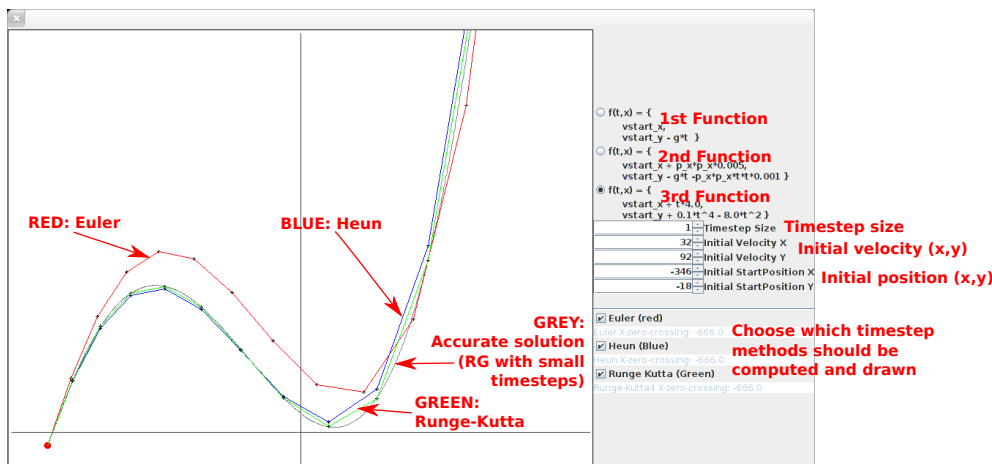


Abbildung 1: GUI for timestep methods

Auf der rechten Seite kann eine der im obigen Abschnitt beschriebenen Funktionen gewählt werden. Die Zeitschrittweite sowie Anfangswerte für Position und Geschwindigkeit lassen sich direkt darunter einstellen.

Im Zeichenfenster auf der linken Seite markiert der rote Punkt den im rechten Bereich angegebenen Startpunkt der ODE. Die Zeitschrittverfahren sind farb-kodiert (siehe Legende rechts), die Kurvenwerte an den diskreten Zeitpunkten jeweils mit einem schwarzen Kreuz markiert.

## Planetensystem: Physikalische Grundlagen

Dieser Abschnitt schildert eine Weiterführung des freien Falls. Verschiedene Objekte, in unserem Fall Planeten, sollen sich unter gegenseitigem Einfluss durch die Gravitation im Raum bewegen. Ausgehend von der Wirkung der Gravitationskraft  $\vec{F}$  zwischen zwei Objekten

$$\vec{F} = Gm_1m_2 \frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|^3}$$

mit  $G$  der Gravitationskonstante, den Massen  $m_1$  und  $m_2$  und den Positionen  $\vec{r}_1$  und  $\vec{r}_2$  der beiden Objekte, lässt sich ein System gewöhnlicher Differentialgleichungen aufstellen. Die Summe aller Kräfte auf ein Objekt wird über den Zusammenhang  $\vec{F} = m\vec{a}$  in eine Beschleunigung umgerechnet, welche der 2. Ableitung des Ortes  $r$  entspricht. Durch Einbeziehen der Geschwindigkeit  $v$  kann man nun diese Differentialgleichung 2. Ordnung in eine Differentialgleichung 1. Ordnung umwandeln:

$$\begin{pmatrix} \dot{r}_i(t) \\ \dot{v}_i(t) \end{pmatrix} = \begin{pmatrix} v_i(t) \\ a_i(t) \end{pmatrix} = \begin{pmatrix} v_i(t) \\ -G \sum_{j \neq i} m_j \frac{r_i(t) - r_j(t)}{|r_i(t) - r_j(t)|^3} \end{pmatrix} \text{ für ein Objekt } i.$$

Ab drei Körpern ist dieses Problem nicht mehr analytisch lösbar, und wir sind auf numerische Mittel wie den uns bekannten Einschrittverfahren angewiesen.

## Zeitschrittverfahren

Nun werden die in dieser Programmieraufgabe verwendeten Zeitschrittverfahren vorgestellt, welche sich alle auf das  $\mathbb{R}^n$  erweitern lassen. Folgende Informationen finden sich auch in den Vorlesungsunterlagen.

### Expliziter Euler

Mit Genauigkeitsordnung 1 ist der explizite Euler das einfachste Zeitintegrationsverfahren. Aus der Ordnung des Verfahrens können wir schließen, dass es nur für lineare Funktionen exakt ist. Ausgehend von einem Zeitpunkt  $t$  wird angenommen, dass die zu integrierende Funktion als Gerade beschrieben werden kann. Die Position zum nächsten Zeitpunkt  $t + \Delta t$  ergibt sich damit wie folgt:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \Delta t \vec{f}(t, \vec{r}(t))$$

Die Funktion  $\vec{f}$  gibt in dieser Aufgabe die Geschwindigkeit zum gegebenen Zeitpunkt  $t$  und einer Position  $\vec{r}(t)$  zurück.

## Heun

Das Verfahren von Heun ist ein Verfahren zweiter Ordnung und mit einer Ordnungsstufe genauer als das Eulerverfahren. Die erste Gleichung, die hierbei benötigt wird, approximiert den nächsten Punkt analog dem Eulerverfahren:

$$\overline{\vec{r}(t + \Delta t)} = \vec{r}(t) + \Delta t \vec{f}(t, \vec{r}(t))$$

Dieser Punkt wird aber nun nicht als neuer Punkt, sondern in einer weiteren Formel verwendet, um den neuen Punkt mit einem Genauigkeitsgrad 2 zu berechnen:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \frac{\Delta t}{2} \left( \vec{f}(t, \vec{r}(t)) + \vec{f}(t + \Delta t, \overline{\vec{r}(t + \Delta t)}) \right)$$

## Runge-Kutta

Die ersten nach ihren Erfindern Carl Runge und Martin Wilhelm Kutta benannten Methoden wurden zu Beginn des 20. Jahrhunderts beschrieben. Die grundlegende Idee hinter den Verfahren ist es, durch das Berechnen von Zwischenwerten innerhalb eines Zeitschrittes ein Verfahren höherer Ordnung zu erhalten. Sowohl das Euler- als auch das Heun-Verfahren entsprechen im übrigen Runge-Kutta-Verfahren der jeweiligen Genauigkeitsordnung.

Das hier beschriebene Verfahren stellt das sogenannte "klassische Runge-Kutta-Verfahren" dar und ist von Genauigkeitsordnung 4.

Zuerst werden die Koeffizienten  $\vec{k}_i$  bestimmt:

$$\begin{aligned}\vec{k}_1 &= \Delta t \vec{f}(t, \vec{r}(t)) \\ \vec{k}_2 &= \Delta t \vec{f}\left(t + \frac{1}{2}\Delta t, \vec{r}(t) + \frac{1}{2}\vec{k}_1\right) \\ \vec{k}_3 &= \Delta t \vec{f}\left(t + \frac{1}{2}\Delta t, \vec{r}(t) + \frac{1}{2}\vec{k}_2\right) \\ \vec{k}_4 &= \Delta t \vec{f}(t + \Delta t, \vec{r}(t) + \vec{k}_3)\end{aligned}$$

Die neue Position können wir dann mit folgender Formel berechnen:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \frac{1}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)$$

## Schnelle (inverse) Fourier-Transformation

Aus der Vorlesung und den Tutorübungen (insb. Aufgabenblatt 8) ist die diskrete und die diskrete inverse Fourier-Transformation bereits bekannt. Da die Implementierung der diskreten (inversen) Fourier-Transformation eine Laufzeit von  $O(n^2)$  hat, wird sie in vielen Programmen durch die schnelle (inverse) Fourier-Transformation mit einer Laufzeit von  $O(n \log n)$  ersetzt.

In dieser Programmieraufgabe beschänken wir uns auf die schnelle inverse Fourier-Transformation (*IFFT*). Den Pseudocode dazu finden Sie im Aufgabenblatt 8.

Das Programmgerüst enthält bereits eine Implementierung der diskreten Fourier-Transformation. Zum testen Ihres Codes können Sie sich folgende Gleichung zunutze machen:

$$IFFT(DFT(v)) = v$$

Für das Verständnis der Fourier-Transformation sind komplexe Zahlen essentiell.

## Komplexe Zahlen

Komplexe Zahlen erweitern den Zahlenbereich der reellen Zahlen, sodass die Gleichung  $x^2 = -1$  lösbar wird. Die Lösung dieser Gleichung ist die Zahl  $i$ , also  $i^2 = -1$ . Die Zahl  $i$  wird auch als imaginäre Einheit bezeichnet.

Alle komplexen Zahlen lassen sich in der Form  $a + b \cdot i$  darstellen, wobei  $a$  und  $b$  reelle Zahlen sind. Komplexe Zahlen lassen sich außerdem (ähnlich wie reelle Zahlen auf einem Zahlenstrahl) in der komplexen Ebene eintragen (siehe Abbildung 2). Dabei entspricht der reelle Anteil  $a$  dem Wert auf der “ $x$ -Achse” und der imaginäre Anteil dem Wert auf der “ $y$ -Achse”.

Neben der Form  $a + bi$  lässt sich jede komplexe Zahl auch eindeutig in Polarkoordinaten  $r \cdot e^{i\varphi}$  darstellen. Aus der komplexen Ebene ist diese Darstellung leicht Zahl ersichtlich:  $r$  entspricht dem Abstand zum Ursprung – dem Radius – und  $\varphi$  dem Winkel zur positiven reellen Achse.

Mit der folgenden Formel lassen sich aus den Polarkoordinaten einer komplexen Zahl  $z$  der reelle Teil  $a$  und der imaginäre Teil  $b$  berechnen:

$$z = r \cdot e^{i\varphi} = r \cdot (\cos(\varphi) + i \cdot \sin(\varphi))$$

In komplexen Zahlen lassen sich “Wurzeln von Eins” darstellen, die für die Fouriertransformation wichtig sind. Die  $n$ -ten Wurzeln von Eins sind die komplexen Zahlen  $z_k = e^{2\pi ik/n}$ ,  $0 \leq k < n$ . Für diese gilt:

$$1 = z_k^n$$

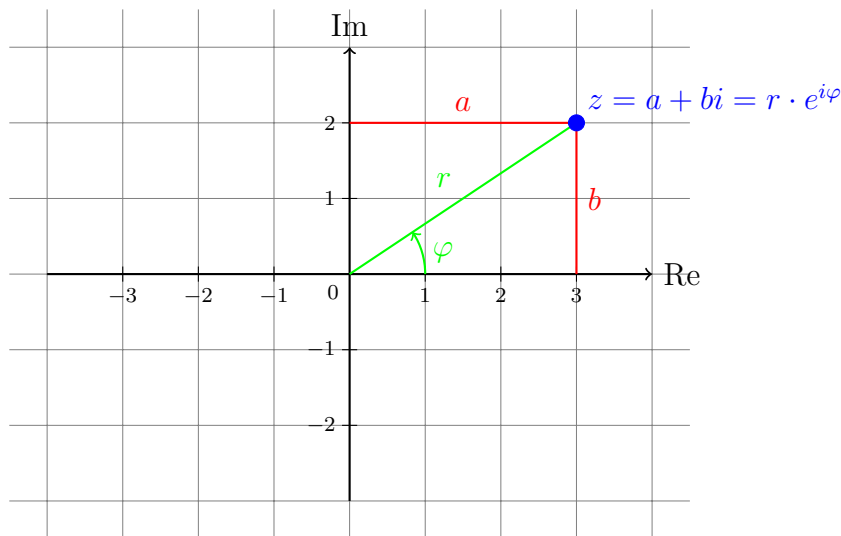


Abbildung 2: Visualisierung einer komplexen Zahl  $z$ .

## Rechenoperationen

### Addition

$$x + y = (a + bi) + (c + di) = (a + c) + (b + d)i$$

### Subtraktion

$$x - y = (a + bi) - (c + di) = (a - c) + (b - d)i$$

### Multiplikation <sup>1</sup>

$$x \cdot y = (a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$$

### Division

$$\frac{x}{y} = \frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

### Potenz mit ganzzahligem Exponenten

Mit dem Satz von Moivre (*Mathematik für Physiker, Kerner u. Wahl, 2013*) lassen sich leicht ganzzahlige Potenzen von komplexen Zahlen in Polarkoordinaten berechnen:

$$z^n = r^n \cdot (e^{i\varphi})^n = r^n \cdot (\cos \varphi + i \sin \varphi)^n = r^n \cdot (\cos(n\varphi) + i \sin(n\varphi)) = r^n \cdot e^{in\varphi}$$

Achtung: Auf Grund der Periodizität von  $\sin$  und  $\cos$  lässt sich diese Formel nicht ohne weiteres auf rationale oder reelle Exponenten erweitern!

<sup>1</sup>Aus dieser Formel können Sie leicht die Definition  $i^2 = -1$  ableiten.

# Beschreibung des Programmgerüsts

Das auf der Vorlesungsseite zur Verfügung gestellte Programmgerüst enthält:

- **freierfall:**  
Dieses Package enthält alle Klassen zur Berechnung des freien Falls und der entsprechenden Benutzeroberfläche. Zur Bearbeitung der Programmieraufgabe ist ein Verständnis dieser Klassen nicht erforderlich.
- **planeten:**  
Dieses Package enthält alle Klassen zur Simulation von Planetensystemen. Zur Bearbeitung der Programmieraufgabe ist ein Verständnis dieser Klassen nicht erforderlich.
- **ODE:**  
Dieses Interface beschreibt eine ODE der Form  $\dot{\vec{y}}(t) = f(t, \vec{y}(t))$  und ermöglicht das Auswerten der rechten Seite  $f$ .
- **ExpliziterEuler, Heun und RungeKutta4:**  
Diese Klassen ermöglichen die Berechnung eines Schrittes des entsprechenden Einschrittverfahrens.
- **Complex:**  
Die Klasse repräsentiert eine komplexe Zahl. Gespeichert wird die Zahl als Realteil und Imaginärteil. Außerdem werden alle wichtigen Operatoren (add, sub, mul, power) bereit gestellt sowie Funktion zum Konvertieren von bzw. in Polarkoordinaten.
- **DFT/IFFT:**  
Die Klassen enthalten Funktionen für die diskrete Fourier-Transformation und schnelle inverse Fourier-Transformation.
- **Test:**  
Die Test-Klasse startet die beiden Benutzeroberflächen für den freien Fall und das Planetensystem, und enthält einige aus der Übung bekannte Testbeispiele.



## Aufgaben

Im Folgenden werden die zu implementierenden Methoden aufgelistet. Details finden Sie jeweils in den Kommentaren zu den einzelnen Methoden. Überlegen Sie sich zu allen Methoden Testbeispiele und testen Sie unbedingt jede einzelne Methode direkt nach der Erstellung. Sie können zusätzlich die zur Verfügung gestellte Testklasse `Test` verwenden. Es ersetzt aber nicht das Verwenden eigener Testbeispiele.

- Programmieren Sie in den Klassen `ExpliziterEuler`, `Heun` und `RungeKutta4` den gegebenen Methodenrumpf `nextStep`.
- Programmieren Sie in der Klasse `Complex` die gegebenen Methodenrumpfe `add`, `sub`, `mul` und `fromPolar`.
- Programmieren Sie in der Klasse `IFFT` den gegebenen Methodenrumpf `ifft`. Den Pseudocode dazu finden Sie in Angabenblatt 8 (Tutorübung).

## Formalien und Hinweise

- Das Programmgerüst erhalten Sie auf den Webseiten zur Vorlesung.
- Ergänzen Sie das Programmgerüst **auf alle Fälle an den dafür vorgegebenen Stellen!** Darüber hinaus können Sie eigene Erweiterungen hinzufügen, falls Sie diese benötigen. Wichtig dabei ist, dass die Signaturen der Klassen mit ihren Methoden und Attributen erhalten bleiben, da sie den für die von uns durchgeführten Testfälle erforderlichen Code enthalten. Fügen Sie **keine** weiteren Dateien hinzu, da diese von unserem automatischen Korrekturtool nicht verarbeitet werden.
- Beseitigen Sie vor Abgabe Ihres Programms alle Ausgaben an die Konsole, die Sie eventuell zu Debugging- oder Testzwecken eingefügt haben und reichen sie Ihre Lösung bis zum **17. Juli 2015, 12:00 Uhr** über Moodle ein.
- Reichen Sie nur die Dateien `ExpliziterEuler.java`, `Heun.java`, `RungeKutta4.java`, `Complex.java` und `IFFT.java` ein. Deren Zuordnung zum Package `ode` bzw. `dft` muss nicht entfernt werden.
- Bitte laden Sie Ihre java-Dateien als flaches tgz-Archiv hoch. Der Dateiname ist beliebig wählbar, bei der Erweiterung muss es sich jedoch um **.tgz** oder **.tar.gz** handeln.

Ein solches Archiv können Sie beispielsweise mit dem Linux-Tool `tar` erstellen, indem Sie die laut Aufgabenstellung zu bearbeitenden java-Dateien in

ein sonst leeres Verzeichnis legen und dort anschließend den Befehl

```
> tar cvvzf numpro_aufg4.tgz *.java
```

ausführen.

*tgz-Archive sind keine Zip- oder Rar-Archive! Es funktioniert also nicht ein Zip- oder Rar-Archiv zu erstellen und die Endung des Dateinamens in .tgz zu ändern.*

- Für die Abgabe der Programmieraufgaben ist das Eintragen in eine Gruppe notwendig.

Bitte beachten Sie:

- *Alle Abgaben, die nicht den formalen Kriterien genügen, werden grundsätzlich mit 0 Punkten bewertet!*
- *Wir testen die Abgaben auch auf Plagiate! Sollte sich herausstellen, dass Sie bei Ihrer Abgabe abgeschrieben haben oder abschreiben haben lassen, bewerten wir die komplette Abgabe mit 0 Punkten!*