# Einführung in die Grundlagen der Numerik

Winter semester 2019/2020
Prof. Dr. Marc Alexander Schweitzer
Denis Düsseldorf

universitätbonn

## Übungsblatt 0.

Ohne Bewertung! Besprechung in der ersten Übung

- Die Webseite zur Vorlesung gibt es unter diesem Link:
  `https://ins.uni-bonn.de/teachings/ws-2019-282-v2e1-mb10-einfuhrung-in-di/`
- Für die Zulassung zur Klausur sind mindestens 50% der Gesamtpunkte der Theorieaufgaben zu erreichen. Ausserdem sind bei *JEDER* Programmieraufgabe mindestens 50% der möglichen Punkte zu erreichen!
- Die Aufgaben sind in Gruppen von 3 Personen zu bearbeiten
- Die Termine für die Tutorien finden sich auf der Webseite. Eine Registrierung erfolgt in der *ERSTEN* Vorlesung!
- Das erste Blatt dient nur der Auffrischung Ihrer Kenntnisse und dem Kennenlernen der Programmiersprache `Python` mit seinen Modulen `Numpy / matplotlib`.

## Die Programmiersprache `Python`

Dieses Blatt dient Wiederholung und der Vorbereitung für die weiteren Programmieraufgaben. Die Programmieraufgaben zur Vorlesung sollen in Python 3 (!) gelöst werden. Python ist eine interpretierte high-level Programmiersprache mit der es möglich ist komplizierte Programme in sehr kurzer Zeit (vgl. mit `C/C++`) zu schreiben. Da Python *interpretiert* und nicht *kompiliert* wird, ist es für eine hohe Performance notwendig Operationen auf die *richtige* Art und Weise auszuführen. Ein Beispiel hierfür sind die so genannten *List Comprehensions*. Schauen wir uns einmal an, wie lange es dauert die Zahlen von 1 bis 1.000.000 in einer neu angelegten Liste zu speichern. Hat man Kenntnis einer Programmiersprache wie `C / C++`, so wird man eventuell einen Loop erstellen und in jeder Iteration ein neues Element an eine ursprünglich leere Liste anhängen. Zusammen mit einer Zeitmessung sieht das dann in Python so aus:

```
# Importiere das benoetigte Modul zur Zeitmessung
import time

# Beginne Laufzeitmessung
start = time.time()

# Initialisiere leere Liste
lst = []
# Fuege neue Elemente im Loop zur Liste hinzu
for i in range(1000000):
    lst.append(i+1) # In Python zaehlt man von 0 los

# Ende der Laufzeitmessung
```

```
elapsed_time = time.time() - start

>> 0.1626739501953125
```

Das Ausführen benötigt also 0.1626739501953125 Sekunden.
Benutzt man die `List Comprehensions`, so wird der Code nicht nur kürzer und eleganter, sondern auch schneller:

```
# Importiere das benoetigte Modul zur Zeitmessung
import time

# Beginne Laufzeitmessung
start = time.time()

# Initialisiere die Liste ueber eine List Comprehension
lst = [i+1 for i in range(1000000)]

# Ende der Laufzeitmessung
elapsed_time = time.time() - start

>> 0.07152318954467773
```

Die Laufzeit hat sich halbiert!

**Hinweis:** Das war nur ein Beispiel ein vielen! Beide Programmschnipsel liefern natürlich das gleiche Resultat und funktionieren daher auch gleich 'gut'. Die Bewertung der Programmieraufgaben erfolgt anhang von Funktionalität, NICHT Effizienz. So lange der Code terminiert ist erst mal alles in Ordnung. Angewöhnen kann man sich einen guten Stil jedoch nie zu früh :-)

**Programmieraufgabe 1.** (Python / NumPy / matplotlib - Einführung)

a) Besorgen Sie sich eine Installation von `Python 3.7` mit den Paketen `NumPy` und `matplotlib`. Sie können ebenfalls die Rechner der CIP Pools in der Endenicher Allee 19b benutzen, auf denen `Python` und die notwendigen Pakete installiert sind. Eine Installation über die Distribution `Anaconda` (`https://www.anaconda.com/distribution/`) ist ebenfalls möglich. `Anaconda` installiert neben dem `Python`-Interpreter bereits unzählige Pakete mit (darunter auch `Numpy` und `matplotlib`).

b) Zur Einführung in `Python` / `NumPy` / `matplotlib`: Schreiben Sie eine rekursive Routine zur Berechnung der ersten $n$ Fibonacci zahlen.

```
import numpy as np
import matplotlib.pyplot as plt

# Routine zur Berechnung der n-ten Fibonacci Zahl
def fibonacci(n):
    # Hier kommt Ihr Code hin
    #
    #
```

c) Schreiben Sie eine Routine die die ersten $n_{max}$ Fibonacci Zahlen plottet. Geben Sie dem Plot einen passenden Titel und speichern Sie ihn als PDF Datei.

```
# Routine zum Plotten der ersten n_max Fibonacci Zahlen
def plot_fibonacci(max_n):
    # Hier kommt Ihr Code hin
    #
    #
```

d) Plotten Sie die folgenden Funktionen für $x \in \{0, 1, \ldots, 20\}$ zusammen in einem einzigen Plot und erstellen Sie eine dazugehörige Legende mit den Namen der Funktionen.

$$\begin{aligned}
\textbf{MyWeight}(x) &= 80 + cos\left(\frac{5x}{\pi}\right) \\
\textbf{StraightOne}(x) &= x + 70 \\
\textbf{StairwayToHeaven}(x) &= 50 + \lfloor x \rfloor \\
\textbf{V-Shape}(x) &= \begin{cases} 70 - x, & \text{if } x < 10 \\ 50 + x, & \text{else} \end{cases}
\end{aligned}$$

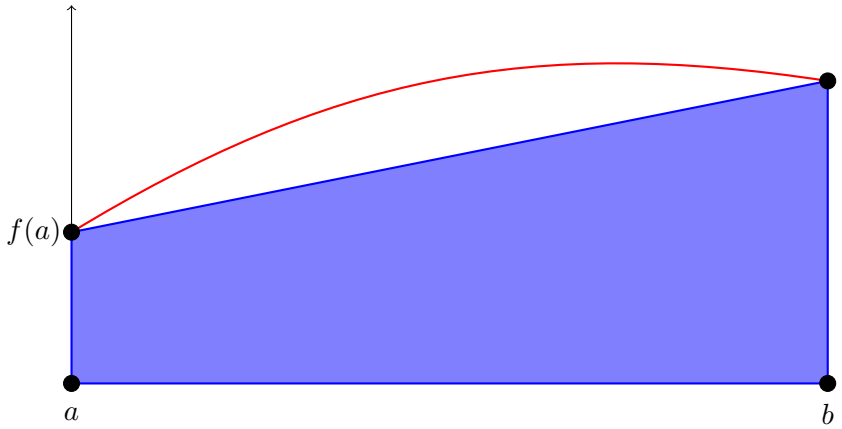
Speichern Sie den Plot mit `savefig()` als PDF Datei.

e) Erstellen Sie ein 2×2 Gitter für Subplots und plotten Sie jede Funktion aus d) in einem dieser Subplots. Der Titel des Subplots soll der Funktionenname sein. Speichern Sie die Plots wie zuvor zusammen in einer PDF Datei.

Keine Abgabe

**Programmieraufgabe 2.** (Trapezregel)

In dieser Aufgabe implementieren Sie eine der einfachsten Regeln zur Quadratur: Die *Trapezregel*. Mit dieser Quadraturformel können lineare Polynome exakt integriert werden.




Für eine Funktion $f$ berechnet sich die Approximation an das Integral $I[f] = \int_a^b f(x)\,dx$ durch

$$Q_{TP}^1[f] = \frac{b-a}{2}[f(a) + f(b)].$$

a) Schreiben Sie eine Routine zur Berechnung der Trapezregel für eine gegebene Funktion mit gegebenen Endpunkten

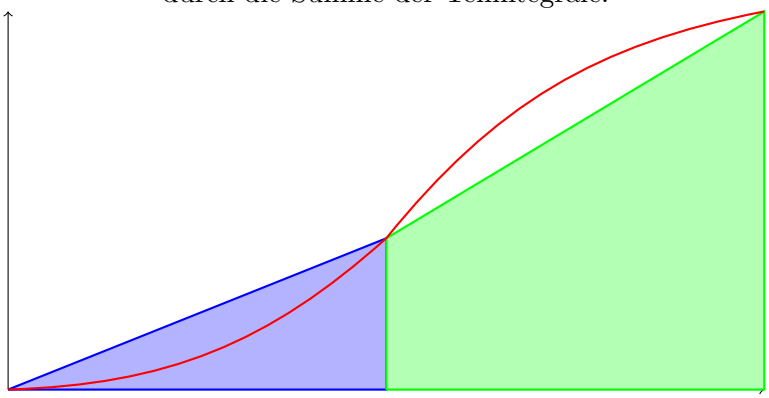
```
def trapezregel(func,a,b):
    # Hier kommt Ihr Code hin
    #
    #
```

`lambda functions` stellen eine einfache Möglichkeit dar, einfachere mathematische Funktionen zu definieren. `lambda functions` können in `Python` wie gewöhnliche Variablen an Funktionen übergeben werden. Testen Sie Ihre Trapezregel mit der Funktion

```
testfunction = lambda x: x**3 + 4
```

für das Interval $[2, 10]$. Der Wert Ihrer Approximation sollte 4064.0 (und damit ziemlich weit neben dem wahren Wert 2528) sein.

b) Jetzt verbessern wir unsere Quadraturformel: Die neue $m$-Intervall-Trapezregel splittet das Integrationsintervall erst in $m$ gleichlange Teilstücke und berechnet für jedes dieser Teilstücke dann approximativ das Teilintegral (mittels standard Trapezregel). Eine Approximation an das Gesamtintegral $I[f]$ ist dann gegeben durch die Summe der Teilintegrale.



Implementieren Sie diese neue Quadraturformel in `Python`. Rufen Sie hierzu Ihre Funktion `trapezregel`$(\cdot,\cdot,\cdot)$ aus a) $m$ mal auf.

```
def trapezregel_mInterval(func,a,b,m):
    # Hier kommt Ihr Code hin
    #
    #
```

c) Plotten Sie die Konvergenz (die Werte) Ihrer Quadraturregel für steigende $m = 1, \ldots, 100$ und speichern Sie den Plot als PDF Datei.

Keine Abgabe

**Programmieraufgabe 3.** Lineare Gleichungssysteme $Ax = b$ die aus interessanten, realen Anwendungen resultieren, setzen sich nicht selten aus mehr als $10^6$ Gleichungen und Unbekannten zusammen. Das direkte Lösen dieser Gleichungssysteme, z.B. mittels Invertieren der Systemmatrix $A$, ist daher keine Option. Einen Ausweg bieten iterative Lösungsverfahren, von denen es eine Vielzahl gibt. Ein gegebener Startwert $x^0$ wird hierbei so lange durch eine bestimmte Vorschrift geupdatet, bis er sich nicht mehr ändert bzw. das Residuum klein genug ist. In dieser Programmieraufgabe konzentrieren wir uns auf zwei der einfachsten iterativen Löser: Die *Methode von Gauss-Seidel und Jacobi.*

Im Folgenden Sei $A \in \mathbb{R}^{n\times n}$ die Matrix des linearen Gleichungssystems $Ax = b$, für ein unbekanntes $x \in \mathbb{R}^n$ und eine gegebene rechte Seite $b \in \mathbb{R}^n$. Wir Zerlegen $A$ mittels

$$A = \underbrace{\begin{bmatrix} 0 & 0 \\ \star & 0 \end{bmatrix}}_{=:L} + \underbrace{\begin{bmatrix} \star & 0 \\ 0 & \star \end{bmatrix}}_{=:D} + \underbrace{\begin{bmatrix} 0 & \star \\ 0 & 0 \end{bmatrix}}_{=:U}$$

in den lower-triangular part $L$, den diagonal part $D$ und den upper-triangular part $U$. Das **Gauss-Seidel** Verfahren berechnet eine neue Iterierte $x^{k+1}$ aus $x^k$ mittels

$$x^{k+1} = [D + L]^{-1}[-Ux^k + b]. \tag{GS}$$

Das **Jacobi** Verfahren berechnet eine neue Iterierte $x^{k+1}$ aus $x^k$ mittels

$$x^{k+1} = D^{-1}[-(L + U)x^k + b]. \tag{JA}$$

a) Implementieren Sie das Gauss-Seidel sowie das Jacobi-Verfahren in `Python`. Sehen Sie sich die `Numpy`-Routinen `triu` bzw `tril` an, um die Eingabematrix zu zerlegen. Das Paket `numpy.linalg` erleichtert Ihnen generell die Arbeit mit Matrizen! Das Abbruchkriterium beider Routinen soll ein sein, wenn das Residuum $\|Ax^k - b\|_2$ kleiner als $\varepsilon = 10^{-8}$ ist. Lassen Sie in jedem Iterationsschritt den Index der Iteration und die Größe des Residuums ausgeben. Um Divergenz zu erkennnen ist es sinnvoll ebenfalls die Iterationszahl zu beschränken. Stellen Sie dieses Limit auf 1000 Iterationen ein.

b) Testen Sie die Löser an der Matrix $A \in \mathbb{R}^{16\times 16}$ mit

$$A = \begin{bmatrix} T & -I & & \\ -I & T & -I & \\ & -I & T & -I \\ & & -I & T \end{bmatrix} \quad \text{wobei} \quad T = \begin{bmatrix} 8 & -1 & 0 & 0 \\ -1 & 8 & -1 & 0 \\ 0 & -1 & 8 & -1 \\ 0 & 0 & -1 & 8 \end{bmatrix}.$$

Benutzen Sie die rechte Seite

$$b = \begin{bmatrix} b_1 & b_2 & b_1 & b_2 \end{bmatrix}^T \quad \text{mit} \quad b_1 = \begin{bmatrix} 6 \\ 5 \\ 5 \\ 6 \end{bmatrix}, \; b_2 = \begin{bmatrix} 5 \\ 4 \\ 4 \\ 5 \end{bmatrix}$$

und den Startwert $x^0 = [0, \ldots, 0]$.

c) Erstellen Sie einen Plot der die Konvergenz beider Verfahren vergleicht. Benutzen Sie eine logarithmisch skalierte Ordinate (`semilogy`) und eine Legende.

Keine Abgabe