Zusammenfassung
Numerische Methoden für Physiker

Alisha Dütschler & Kevin Kazuki Huguenin-Dumittan

1. Juni 2017

Inhaltsverzeichnis

Vorwort

Diese Zusammenfassung dient zur Begleitung der Vorlesung "numerische Methoden für Physiker" von Dr. Vasile Gradinaru. Sie wurde ursprünglich für das Frühjahressemester 2016 geschrieben. Sie basiert inhaltlich auf der Zusammenfassung von Tim Engel, geschrieben für den PVK 2015. Sie sollte die wichtigsten Konzepte und Algorithmen der Vorlesung und des Vorlesungsskripts enthalten, dient aber keinesfalls als komplette Ersetzung dafür. Aus diesem Skript stammen auch die meisten Abbildungen, die in dieser Zusammenfassung verwendet werden. Zusätzlich werden im Laufe des Semesters einige Beispielprogramme mit den wichtigsten Codes abgegeben. Das Ziel dabei ist, dass die Studenten mehr Zeit für das Anwenden und selbstständige Bearbeiten von Aufgaben haben, anstatt im Skript mühselig die wichtigsten Codefragmente zusammensuchen zu müssen.

Die Autoren sowie die ETH Zürich nehmen keine Haftung für inhaltliche Korrektheit und Vollständigkeit. Trotzdem versuchen wir natürlich, das Skript so gut wie möglich zu gestalten. Kritik und Verbesserungsvorschläge jeglicher Art, von gravierenden inhaltlichen Fehlern bis zu kleinen Rechtschreibfehlern oder alternativen Formatierungsmöglichkeiten, sind immer willkommen.

Konvention für Python-Befehle

Die Beispielcodes innerhalb dieser Zusammenfassung wurden ursprünglich für die Python Version 2.7 geschrieben, sollten aber alle auf Version 3.6 angepasst worden sein. Bei importierten Bibliotheken gilt folgende Konvention:

- Direkte Befehle aus **numpy** sowie **matplotlib.pyplot** werden ohne Angabe des vollständigen Importierungspfades angegeben (z.B. exp, linspace, polyfit, plot)
- Alle anderen Befehle werden jeweils mit dem vollständigen Importierungspfad angegeben (z.B. numpy.linalg.solve, numpy.random.rand, numpy.fft.fft, scipy.optimize.fsolve)

Bei den Beispielprogrammen hingegen wird aus Übersichtsgründen vieles zu Beginn importiert. Diese enthalten jeweils in den ersten Zeilen eine Beschreibung des Programmes sowie allfällige Hinweise, wie man mit diesen Programmen am besten lernen kann.

1 Nullstellensuche

Geg: $F : U \subset \mathbb{R}^n \longrightarrow \mathbb{R}^n$ eine beliebige Funktion
Ges: $x^* \in \mathbb{R}^n$ s.d. $F(x^*) = 0$

1.1 Grundbegriffe: Iteratives Verfahren

Definition (Iteratives Verfahren)
Ein Iteratives Verfahren ist ein Algorithmus definiert durch:
- Einen Startwert x_0
- Eine Iterationsvorschrift $x_{k+1} = \phi(x_k)$
- Eine Abbruchbedingung

Somit erzeugt es eine Folge $x_0, x_1, \ldots, x_N$ von approximierten Lösungen zu einem Problem.

Sei $lim_{k\to\infty} x_k = x^*$ für ein x^*
Fehler $e_k := \| x^* - x_k \|$
In der Regel ist x^* nicht bekannt. Dann: $x^* \approx x_N$ für $N \gg 1$, $e_k \approx \| x_N - x_k \|$

Konvergenzordnung p $\exists c > 0 : e_{k+1} \leq c\, e_k^p$
Berechnung: $e_{k+1} \approx c\, e_k^p \Rightarrow p \approx \frac{\log(e_{k+1}/e_k)}{\log(e_k/e_{k-1})}$

Konvergenzrate L Falls $p = 1$ (lineare Konvergenz): $L \equiv c < 1$ s.d. $e_{k+1} \leq L\, e_k$
Berechnung: $e_k \approx L\, e_{k-1} \approx L^k\, e_0 \Rightarrow \log(e_k) \approx k\, \log(L) + \log(e_0)$

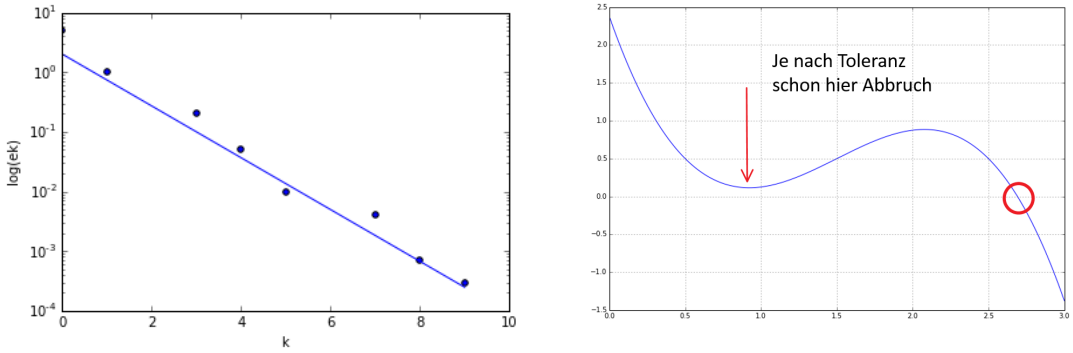
```
# Konvergenzordnung
p = log(e[2:]/e[1:-1])/log(e[1:-1]/e[:-2])
paverage = sum(p)/size(p)

# Konvergenzrate, falls p = 1
semilogy(k,e) #: Dies muss eine Gerade sein
fit = polyfit(k,log(e),1) # siehe das Kapitel 5. fuer polyfit
L = exp(fit[0])
```

Abbruchkriterien Bsp: Nullstellensuche, $F(x^*) = 0$.
1. Abbruch nach fixer Anzahl Schritte (schlecht, Notlösung damit kein ∞ loop).
2. Naives Abbruchkriterium: $\| F(x_k) \| < tol$ (Siehe Abbildung 1b $\rightarrow$ Problem)
3. Absolute Änderung: $\| x_k - x_{k-1} \| < tol$ (keine Angaben über Grössenordnung)
4. (Relative) Änderung: $\| x_k - x_{k-1} \| / \| x_k \| < tol$ (Problem bei $x_k = 0$)

Kein Kriterium ist also perfekt. In der Regel wird das Kriterium 4 mit einigen anderen kombiniert, um allfällige Probleme zu beheben. Eine Alternative, falls an der Prüfung oder in Serien zu übungszwecken der exakte Wert x^* bekannt sein sollte:
5. $\| x_k - x^* \| < tol$

```
tol = 1e-10
maxit = 1000
for k in range(1,maxit+1):  # Kriterium 1 als Notloesung
    x[k] = ...
    if numpy.linalg.norm(x[k]) < 1e-13:     # Kriterium 3, falls x zu nahe an 0
        if numpy.linalg.norm(x[k]-x[k-1]) < tol:
            break
    elif numpy.linalg.norm(x[k] -x[k-1]) < tol * norm(x[k]): # Kriterium 4, sonst
        break
```



(a) Lin-log-Plot (b) Versagen des Abbruchkriteriums

Abbildung 1

1.2 Fixpunktiteration

Idee: Nullstellenproblem $\Rightarrow$ Fixpunktproblem $F(x^*) = 0 \Rightarrow \phi(x^*) = x^*$

Vorgehen (Fixpunktiteration)
Vorbereitung: Wähle ein $\phi(x)$ (an der Prüfungs meistens bereits gegeben)
Iteratives Verfahren mit Iterationsvorschrift:
- $x_{k+1} = \phi(x_k)$

Bsp: $F(x) = xe^x - 1$
$\phi_1 = e^{-x}$ (lineare Konvergenz)
$\phi_2 = (1+x)/(1+\exp(x))$ (quadratische Konvergenz)
$\phi_3 = x + 1 - x\exp(x)$ (Divergenz)
Es gibt also sehr viele unterschiedlich gute Möglichkeiten, die Funktion ϕ zu wählen.

Konvergenz der Fixpunktiteration

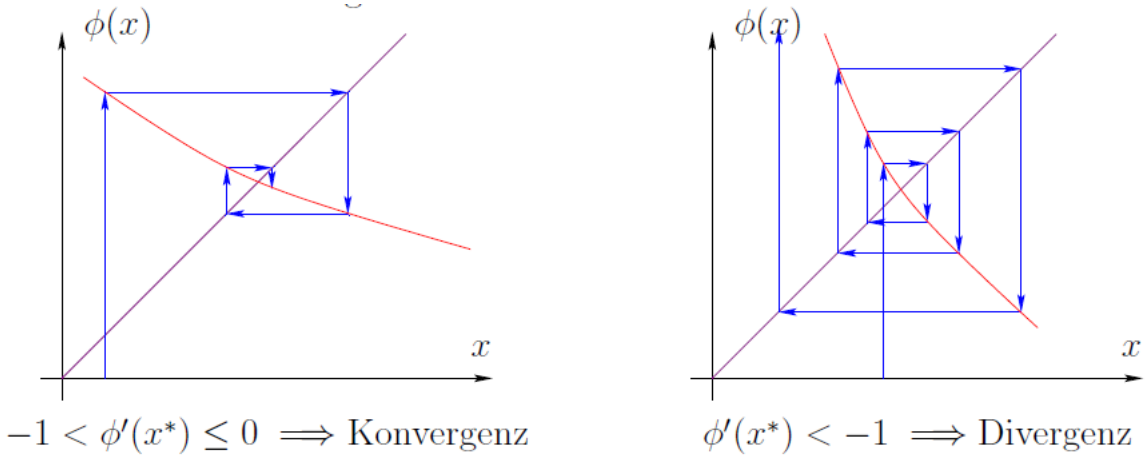
Hilfssatz 1.3.11: Sei $\phi : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ mit $\phi(x^*) = x^*$ stetig differenzierbar und $\| D\phi(x^*) \| < 1 \Rightarrow$ lokal mind. lineare Konvergenz. (Siehe Abbildung 2.)
Hilfssatz 1.3.14: Sei I $\subset \mathbb{R}$ ein Intervall und $\phi : I \longrightarrow \mathbb{R}$ (m+1)-mal differenzierbar mit $\phi(x^*) = x^* \in I$. Sei weiter $\phi^{(k)}(x^*) = 0$ für k=1,2,..,m ($m \geq 1$). $\Rightarrow$ lokale Konvergenz gegen x^* mit Ordnung $p \geq m + 1$.
Bsp:

1. $\phi(x) = \exp(-x), \phi'(x) = -\exp(-x). \mid \phi'(x) \mid < 1 \overset{(1.3.11)}{\Rightarrow} x_{k+1} = \phi(x_k)$ konvergiert lokal (mind.) linear. Aber: $\phi'(x) \neq 0$ d.h. keine Voraussagen mit 1.3.14.

2. $\phi(x) = x - F(x)/F'(x)$ wobei $F(x^*) = 0, \phi'(x) = F(x)F''(x)/(F'(x))^2 \Rightarrow \phi(x^*) = 0$, falls $F'(x^*) \neq 0 \Rightarrow$ lokale Konvergenz min. der Ordnung 2.

3. $\phi(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) = c \begin{pmatrix} cos(x_1) - sin(x_2) \\ cos(x_1) - 2sin(x_2) \end{pmatrix}, D\phi(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) = -c \begin{pmatrix} sin(x_1) & cos(x_2) \\ sin(x_1) & 2cos(x_2) \end{pmatrix}$
Wahl geeignete Norm: Zeilensummennorm $\| A \|_\infty = max_{i=1,\ldots,m} \sum_{j=1}^{n} \mid a_{ij} \mid$
$\Rightarrow \| D\phi \|_\infty < 3c \overset{(1.3.11)}{\Rightarrow}$ Für $c < 1/3$: lokale min. lineare Konvergenz.

Abbildung 2: Beispiele für die Konvergenz /Divergenz



1.3 Intervallhalbierungsverfahren (Bisektionsverfahren)

Idee: Zwischenwertsatz:
$F : [a, b] \longrightarrow \mathbb{R}$ stetig mit $F(a)F(b) < 0 \Rightarrow \exists x^* \in [a, b] : F(x^*) = 0$

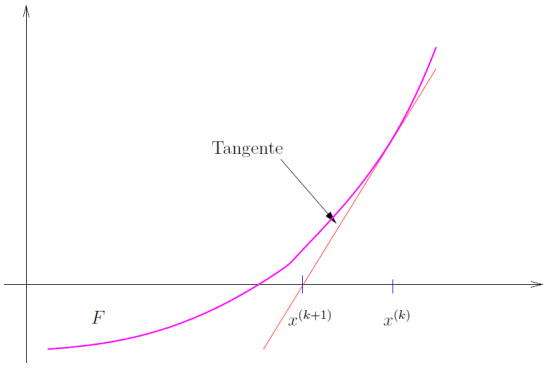
Vorgehen (Bisektionsverfahren)
Iteratives Verfahren mit der Iterationsvorschrift:
- Intervall halbieren $m = \frac{a+b}{2}$
- Suche Intervall mit unterschiedlichem Vorzeichen: $F(a)F(m) < 0$ oder $F(m)F(b) < 0$
- Weiter mit Intervall "< 0"

```
xstar = scipy.optimize.bisect(F,a,b)
```

Lineare Konvergenz (langsam). Keine Verallgemeinerung in mehrere Dimensionen.

1.4 Newtonverfahren

Abbildung 3: Newtonverfahren



Idee: Approximation von F durch Tangente bei x_k (Taylor):
$F(x) \approx \tilde{F}(x) := F(x_k) + F'(x_k) \cdot (x - x_k) \stackrel{!}{=} 0$

Vorgehen (Newtonverfahren)
Iteratives Verfahren mit Iterationsvorschrift:
- $x_{k+1} = x_k - F(x_k)/F'(x_k)$

Newtonverfahren (mehrdimensional)

Gleiche Idee: $x_{k+1} = x_k - DF(x_k)^{-1}F(x_k) =: x_k - s_k$ (s_k Newton Korrektur)

Aber: $DF(x) := \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix} \rightarrow$ "Berechnen Sie nie das Inverse einer Matrix!"

Vorgehen (Mehrdimensionales Newtonverfahren)
Iteratives Verfahren mit Iterationsvorschrift:
- $\vec{s}_k$ = Lösung des linearen Gleichungssystems $DF(\vec{x}_k)\vec{s}_k = F(\vec{x}_k)$
- $\vec{x}_{k+1} = \vec{x}_k - \vec{s}_k$

Newtonverfahren = Fixpunktiteration mit $\phi(x) = x - F(x)/F'(x)$ (siehe Kap.1.2 Bsp.2).
$\Rightarrow$ lokale, (mindestens) quadratische Konvergenz.

Bsp: $F(x) = \begin{pmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2 \rightarrow F(\begin{pmatrix} 1 \\ 1 \end{pmatrix}) = 0, DF(x) = \begin{pmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{pmatrix}$

Sekantenverfahren

Falls F' unbekannt: $F'(x) \approx \frac{F(x_k)-F(x_{k-1})}{x_k - x_{k-1}}$
Konvergenzordnung $p \approx 1.62$.
Beachte: 2 Startwerte benötigt. Verallgemeinerung in mehrere Dimensionen (im Gegensatz zu Newton) nicht möglich.

1.5 Gedämpftes Newtonverfahren

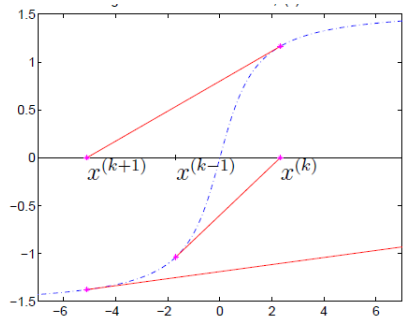


Abbildung 4: Divergenz Newtonverfahrens

Ob das Newtonverfahren konvergiert hängt vom Startwert x_0 ab! Es gibt Fälle, in denen die Newton Korrektur s_k zu gross wird und das Verfahren deshalb divergiert.

Bsp: Für $f(x) = \arctan(x)$ konvergiert das Newtonverfahren nur für Startwerte $x_0 \in [-1.39, 1.39]$. (Siehe Abbildung 4.)

Idee: Dämpfung von s_k mit einem Dämpfungsparameter $\lambda_k \in (0, 1]$.

Es wird das maximale λ_k gewählt, so dass $\| \bar{s}(\lambda_k) \|_2 \leq (1 - \frac{\lambda_k}{2}) \| s_k \|_2$ $(*)$ mit $\bar{s}(\lambda_k) := DF(x_k)^{-1}F(x_k - \lambda_k s_k)$.
Praxis: $\lambda_k = 1 \rightarrow \frac{1}{2} \rightarrow \frac{1}{4} \rightarrow ... \rightarrow$ bis die Bedingung $(*)$ erfüllt ist.

Vorgehen (Gedämpftes Newtonverfahren)
Iteratives Verfahren mit Iterationsvorschrift:
- $\vec{s}_k$ = Lösung von $DF(\vec{x}_k)\vec{s}_k = F(\vec{x}_k)$ oder in 1D: $s_k = F(x_k)/F'(x_k)$
- $\lambda_k = max\{\frac{1}{2^n} \,|\, n \in \mathbb{N}_0$ und $(*)$ ist erfüllt$\}$ (in jedem Iterationschritt neu berechnen!)
- $\vec{x}_{k+1} = \vec{x}_k - \lambda_k \vec{s}_k$

1.6 Zusammenfassung

	Vorteile	Nachteile
Fixpunktiteration	⋆ unterschiedliche Konvergenzordnungen	⋆ für jede Funktion neu berechnen ⋆ kann divergieren
Bisektionsverfahren	⋆ konvergiert immer → braucht keinen guten Startwert!!	⋆ langsam ⋆ nur in 1D
Newtonverfahren	⋆ auch in mehreren Dimensionen ⋆ mind. quadratische Konvergenz ($p \geq 2$)	⋆ braucht Ableitung
Sekantenverfahren	⋆ wenn Ableitung unbekannt ⋆ p $\approx$ 1.62 auch nicht schlecht	⋆ nur in 1D braucht 2 Startwerte

Ein gemeinsamer Nachteil aller Verfahren bis auf das Bisektionsverfahren ist die nur lokale Konvergenz. Professionelle Codes wie fsolve bauen oft auf das Gedämpfte Newtonverfahren auf.

2 Numerische Lineare Algebra

2.1 Matrixzerlegungen

2.1.1 LU-Zerlegung

Sei A $\in \mathbb{R}^{n\times n}$ invertierbar, dann existieren P, L, U $\in \mathbb{R}^{n\times n}$, so dass PA = LU. Wobei L eine untere Dreiecksmatrix mit Einsen auf der Diagonalen, U eine obere Dreiecksmatrix und P eine Permutationsmatrix sind.
Anwendung: Lösen von Gleichungssystemen
$Ax = b \Leftrightarrow LUx = Pb \Leftrightarrow Lz = Pb$ (Vorwärtssubstitution) & $Ux = z$ (Rückwärtssubstitution)

```
P, L, U = scipy.linalg.lu(A)
```

2.1.2 Cholesky-Zerlegung

Ist A symmetrisch ($A = A^T$) und positiv definit, dann existiert eine Zerlegung $A = LL^T = U^TU$, wobei L / U eine untere / obere Dreiecksmatrix mit strikt positiven Diagonaleinträgen ist.

```
L = numpy.linalg.cholesky(A) # L untere Dreiecksmatrix mit dot(L,L.T) == A

# Loesen von Gleichungssystemen:
Lspez, P = scipy.linalg.cho_factor(A) # speziell formatiertes L; nur fuer LGS geeignet
x = scipy.linalg.cho_solve((Lspez,P),b)
```

2.1.3 QR-Zerlegung

Sei A $\in \mathbb{R}^{m\times n}$ mit $m \geq n$, dann existiert ein $\hat{Q} \in \mathbb{R}^{m\times n}$, $\hat{R} \in \mathbb{R}^{n\times n}$, so dass $A = \hat{Q}\hat{R}$, wobei $\hat{Q}$ orthonormale Spalten hat und $\hat{R}$ eine obere Dreiecksmatrix ist. (reduzierte QR-Zerlegung)

Bem: $A = QR$, wobei $Q := \begin{pmatrix} \hat{Q} & q_{n+1} & \dots & q_m \end{pmatrix} \in \mathbb{R}^{m\times m}, R := \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} \in \mathbb{R}^{m\times n}$. Mit $Q^TQ = \mathbb{1}$ (orthogonal). (vollständige QR-Zerlegung)

```
Qhat, Rhat = numpy.linalg.qr(A) # reduzierte QR-Zerlegung
Q, R = numpy.linalg.qr(A,mode='complete') # vollständige QR-Zerlegung
```

Anwendung: Ausgleichsrechung (siehe Kapitel 3), Lösen von linearen Gleichungssystemen:
$Ax = b \leftrightarrow QRx = b \leftrightarrow Rx = Q^Tb = z$

2.1.4 Singulärwertzerlegung

Sei A $\in \mathbb{R}^{m\times n}$, dann existiert ein U $\in \mathbb{R}^{m\times m}$, V $\in \mathbb{R}^{n\times n}$ und $\Sigma \in \mathbb{R}^{m\times n}$, so dass A=$U\Sigma V^T$ wobei V und U orthogonal und $\Sigma = diag(\sigma_1, \dots, \sigma_p)$ mit $p = min\{m, n\}$. $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$ sind die Singulärwerte von A.
Anwendung: Ausgleichsrechnung (siehe Kapitel 3)

```
U,s,Vt = scipy.linalg.svd(A) # U,Vt (transponierte von V) sind Matrizen, s ist ein Vektor mit
den Singulärwerten.
# Die Matrix Sigma erhält min wie folgt:
Sigma = zeros((m,n))
p = min(m,n)
Sigma[:p,:p] = diag(s)
```

Sind $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0$, ist r der Rang der Matrix. Da Numerische Verfahren nicht immer ganz korrekt sind, muss dies nicht immer mit dem numerischen Rang ($\tilde{r} := max\{i \mid \sigma_i > \epsilon\}$) übereinstimmen.

```
r = 1+where(s/s[0]> eps)[0].max() # s mit svd wie im vorherigen Code
```

2.2 Kondition eines Problems

f : $\mathbb{R}^n \longrightarrow \mathbb{R}^n$ mathematisches Problem — x: Exakte Anfangsdaten
$\tilde{f}$: Algorithmus zu f — $\tilde{x}$: gestörte Anfangsdaten

Fehler = $\| f(x) - \tilde{f}(\tilde{x}) \| \leq \| f(x) - f(\tilde{x}) \| + \| f(\tilde{x}) - \tilde{f}(\tilde{x}) \|$ (Dreiecksungleichung)

- Kondition: $\| f(x) - f(\tilde{x}) \|$, Wie robust ist das Problem gegenüber Störungen?
- Konsistenz: $\| f(\tilde{x}) - \tilde{f}(\tilde{x}) \|$, Wie gut ist der Algorithmus (bzgl. den gestörten Daten)?

Eine weitere wichtige Grösse ist:

- Stabilität: $\| \tilde{f}(x) - \tilde{f}(\tilde{x}) \|$ Wie empfindlich ist der Algorithmus gegenüber Störungen der gegebenen Daten?

Spezialfall: Lineare Abbildungen f(x)=Ax mit A $\in \mathbb{R}^{n \times n}$ invertierbar.
$cond(A) := \| A^{-1} \| \| A \|$, $cond_2(A) := \| A^{-1} \|_2 \| A \|_2 = \frac{\sigma_1}{\sigma_n}$

```
numpy.linalg.cond(A)
```

2.2.1 Inversion einer Matrix

Geg: Lineares Gleichungssystem $Ax = b \Leftrightarrow x = A^{-1}b$
$cond(A) = cond(A^{-1}) \approx 1$: kleine Rundungsfehler, stabil unter Störungen von b
$cond(A) = cond(A^{-1}) \gg 1$: grosse Rundungsfehler, instabil unter Störungen von b

Eigentlich kann Python die Inverse einer $n \times n$ Matrix A mit numpy.linalg.inv(A) berechnen. Dies ist aber nicht sehr effizient, und zeigt oft grosse Rundungsfehler. Deshalb sollte dies mit numpy.linalg.solve oder scipy.linalg.solve getan werden.

```
invA = numpy.linalg.solve(A,eye(A.ndim)) # A n x n Matrix!
```

3 Ausgleichsrechunug

Geg: Daten $(t_i, y_i) \in \mathbb{R}^2, i \in \{1, \ldots, m\}$, Modell $f_{\vec{x}} : \mathbb{R} \longrightarrow \mathbb{R}, f_{\vec{x}}(t) = \vec{y}$
Ges: Parameter $\vec{x} = (x_1, \ldots, x_n)$, so dass die Daten und das Modell *am besten* passen.

$$\text{Wir wollen: } \begin{pmatrix} f_{\vec{x}}(t_1) \\ \vdots \\ f_{\vec{x}}(t_m) \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \Leftrightarrow \underbrace{\begin{pmatrix} |f_{\vec{x}}(t_1) - y_1| \\ \vdots \\ |f_{\vec{x}}(t_m) - y_m| \end{pmatrix}}_{:=\vec{R}=Residuum/Residuenvektor} = \vec{0}$$

Bei der Ausgleichsrechnung: $m > n$. Das erste Gleichungssystem ist somit überbestimmt und es gibt keine Lösung. Stattdessen erhalten wir das Minimierungsproblem:

Least Squares Problem: $\vec{x}^\star = argmin_{\vec{x}} \sum_{i=1}^{m} \mid f_{\vec{x}}(t_i) - y_i \mid^2 = argmin_{\vec{x}} \parallel \vec{R} \parallel_2^2$

3.1 Lineare Ausgleichsrechnung

Falls $f_{\vec{x}}$ linear im Parameter $\vec{x}$, also $f_{\vec{x}}(t) = x_1 b_1(t) + \cdots + x_n b_n(t)$ für Basisfunktionen $b_j(t)$ (nicht notwendigerweise linear in t), dann handelt es sich um lineare Ausgleichsrechnung. Für diesen Spezialfall lässt sich das Problem umschreiben zu:

$$\underbrace{\begin{pmatrix} b_1(t_1) & \ldots & b_n(t_1) \\ \vdots & \vdots & \vdots \\ b_1(t_m) & \ldots & b_n(t_m) \end{pmatrix}}_{:=A\ m\times n Matrix} \underbrace{\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}}_{:=\vec{x}} = \underbrace{\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}}_{:=\vec{b}} \Leftrightarrow A\vec{x} = \vec{b} \Leftrightarrow \parallel A\vec{x} - \vec{b} \parallel = 0$$

Da $m > n$ ist das Gleichungssystem $A\vec{x} = \vec{b}$ wie im allgemeinen Fall überbestimmt. Wir lösen also stattdessen das für den linearen Fall umschriebene Least Squares Problem:
$\vec{x}^* = argmin_{\vec{x}} \parallel A\vec{x} - \vec{b} \parallel_2^2 \quad (*)$

```
xstar = numpy.linalg.lstsq(A,b)[0]
```

Bsp:

Daten:				
T[C] (t_i)	98.269	0	-194.96
p[$mmHg$] (y_i)	852.7	624.5	172.7

Modell: (Gesetz von Gay-Lussac) $p(T) = \alpha T + \beta$
$f_{\vec{x}}(t) = x_1 b_1(t) + x_2 b_2(t)$ wobei $\vec{x} = (x_1, x_2) = (\alpha, \beta)$ und $b_1(t) = t\ , b_2(t) = 1$

$$A = \begin{pmatrix} 98.269 & 1 \\ 0 & 1 \\ -194.96 & 1 \end{pmatrix}, \vec{b} = \begin{pmatrix} 852.7 \\ 624.5 \\ 172.7 \end{pmatrix} \text{ Frage: } \vec{x}^\star = argmin \parallel A\vec{x} - \vec{b} \parallel_2^2 ?$$

3.1.1 Normalengleichung

Idee: Bestimme Minimum durch: $grad(\parallel A\vec{x} - \vec{b} \parallel_2^2) \stackrel{!}{=} 0$
$\vec{x}^\star = argmin \parallel A\vec{x} - \vec{b} \parallel_2^2 \Leftrightarrow \mathbf{A^T A \tilde{x}^\star = A^T \tilde{b}}$

```
ATA = dot(A.T,A)
ATb = dot(A.T,b)
xstar = numpy.linalg.solve(ATA,ATb)
```

Bem: $\text{cond}(A^T A) = \text{cond}\ (V\Sigma^T U^T U \Sigma V^T) = \text{cond}(V\Sigma^2 V^T) = \frac{\sigma_1^2}{\sigma_n^2} = (\text{cond(A)})^2 \geq \text{cond(A)}$.
Die Normalengleichung ist also schlecht konditioniert und wenn möglich zu vermeiden.

3.1.2 QR-Zerlegung

Idee: A = QR = $\hat{Q}\hat{R}$ (vollständige und reduzierte QR- Zerlegung)

$$\| A\vec{x} - \vec{b} \|_2^2 = \| QR\vec{x} - \vec{b} \|_2^2 = \| R\vec{x} - Q^T\vec{b} \|_2^2 = \| \hat{R}\vec{x} - \hat{Q}^T\vec{b} \|_2^2 + \| \begin{pmatrix} q_{n+1} \\ \vdots \\ q_m \end{pmatrix} \vec{b} \|_2^2$$

$$R\vec{x} - Q^T\vec{b} = \left(\begin{array}{c} \hat{R} \\ \hline 0 \\ \vdots \\ 0 \end{array}\right) \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} - \left(\begin{array}{c} \hat{Q}^T \\ \hline \\ q_{n+1} \\ \vdots \\ q_m \end{array}\right) \begin{pmatrix} b_1 \\ b_2 \\ \\ \vdots \\ \\ b_m \end{pmatrix} = \left(\begin{array}{c} \hat{R}\vec{x} - \hat{Q}^T\vec{b} \\ \hline \\ \begin{pmatrix} q_{n+1} \\ \vdots \\ q_m \end{pmatrix} \vec{b} \end{array}\right)$$

Minimiere den von $\vec{x}$ abhängigen Teil exakt: $\vec{x}^\star = argmin \| A\vec{x} - \vec{b} \|_2^2 \Leftrightarrow \mathbf{\hat{R}\tilde{x}^\star = \hat{Q}^T\tilde{b}}$

```
Qhat,Rhat = numpy.linalg.qr(A)
xstar = numpy.linalg.solve(Rhat,dot(Qhat.T,b))
```

3.1.3 Singulärwertzerlegung

Idee: $A = U\Sigma V^T = \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \left(\begin{array}{c} V_1^T \\ \hline \\ V_2^T \end{array}\right)$

Dabei ist r = Rang(A) = Anzahl Singulärwerte $\neq 0$, $U_1 \in \mathbb{R}^{m\times r}$,$V_1 \in \mathbb{R}^{n\times r}$

Analoge Überlegungen wie bei QR liefert: $\vec{x}^\star = argmin \| A\vec{x} - \vec{b} \|_2^2 \Leftrightarrow \mathbf{\tilde{x}^\star = V_1\Sigma_r^+U_1^T\tilde{b}}$

($V_1\Sigma_r^+U_1^T$ heisst Pseudoinverse von A, wobei $\Sigma_r^+ = \Sigma_r^{-1} = diag(\sigma_1^{-1}, ..., \sigma_r^{-1})$)

```
psinvA = numpy.linalg.pinv(A) # Pseudoinverse
xstar = dot(psinvA,b)
```

Vorgehen (Lineare Ausgleichsrechnung)
- Vorbereitung: $\vec{x}$ und Basisfunktionen bestimmen
- Matrix A erstellen
- Bestimme die optimalen Parameter $\vec{x}$
- Rekonstruiere die Fitfunktion mit den optimalen Parametern

3.2 Nicht-lineare Augleichsrechunung

Least squares problem: $\vec{x}^\star = argmin_{\vec{x}} \sum_{i=1}^{m} |f_{\vec{x}}(t_i) - y_i|^2 \ \ (*)$

Definiere: $F(\vec{x}) := \begin{pmatrix} f_{\vec{x}}(t_1) - y_1 \\ \vdots \\ f_{\vec{x}}(t_m) - y_m \end{pmatrix}$, $\phi(\vec{x}) := \frac{1}{2} \| F(\vec{x}) \|_2^2 \rightarrow (*) \Leftrightarrow \vec{x}^\star = argmin\ \phi(\vec{x})$

```
xstar = scipy.optimize.leastsq(F,x0)[0]
# F: F wie oben definiert x0: Startwert
```

3.2.1 Newton Verfahren

Idee: $\phi(\vec{x})$ minimal $\Leftrightarrow$ $grad(\phi(\vec{x})) = 0 \rightarrow$ bestimme Nullstelle mit Newtonverfahren
Wir brauchen dazu auch die Ableitung von $grad(\phi(\vec{x}))$, also die Hessematrix.

Vorgehen
- $grad\phi(\vec{x}) := (DF(\vec{x}))^T F(\vec{x})$
- $Hess\phi(\vec{x}) := (DF(\vec{x}))^T DF(\vec{x}) + \sum_{j=1}^{m} F_j(\vec{x}) HessF_j(\vec{x})$
- $\vec{x}^*$ = Newton($grad\phi$, $Hess\phi$, $\vec{x}_0$)

Nachteil: Diese Berechnungen sind sehr mühsam.

3.2.2 Gauss-Newton Verfahren

Idee: Taylor: $F(\vec{x}) \approx F(\vec{x}_k) + DF(\vec{x}_k)(\vec{x} - \vec{x}_k) =: F(\vec{x}_k) + DF(\vec{x}_k)\vec{s}$
Minimiere also stattdessen $\| F(\vec{x}) \| \approx \| F(\vec{x}_k) + DF(\vec{x}_k)\vec{s} \| = \| DF(\vec{x}_k)\vec{s} - (-F(\vec{x}_k)) \|$
Lineares Ausgleichsproblem lösen!

Vorgehen (Gauss-Newton-Verfahren)
Iteratives Verfahren mit Iterationsvorschrift:
- $\vec{s}_k$ = Lösung des linearen Ausgleichsproblems $\vec{s} = argmin_{\vec{s}} \| DF(\vec{x}_k)\vec{s} - (-F(\vec{x}_k)) \|$
- $\vec{x}_{k+1} = \vec{x}_k + \vec{s}_k$

Bsp: Daten:

t[min] (t_i)	0	5	10	15	20	25
I[mA] (y_i)	9.66	18.8	22.36	24.07	24.59	24.91

Modell: $I(t) = \alpha - \beta e^{-\gamma t} \rightarrow f_{\vec{x}}(t) = x_1 - x_2 e^{-x_3 t}$

$$F(\vec{x}) = \begin{pmatrix} x_1 - x_2 e^{-x_3 t_1} - y_1 \\ \vdots \\ x_1 - x_2 e^{-x_3 t_m} - y_m \end{pmatrix} \Rightarrow DF(\vec{x}) = \begin{pmatrix} 1 & -e^{-x_3 t_1} & t_1 x_2 e^{-x_3 t_1} \\ \vdots & \vdots & \vdots \\ 1 & -e^{-x_3 t_m} & t_m x_2 e^{-x_3 t_m} \end{pmatrix} \in \mathbb{R}^{m \times 3}.$$

3.3 Zusammenfassung

Lineare Ausgleichsrechnung

	Vorteile	Nachteile
Normalengleichung	⋆ $A \in \mathbb{R}^{m \times n}$ dünn besetzt und m,n gross (braucht Modifikation)	⋆ schlecht konditioniert $A^T A$ für grosse m,n aufwendig
QR-Verfahren	⋆ $A \in \mathbb{R}^{m \times n}$ vollbesetzt und n klein ($m >> n$) ⋆ numerisch stabil	⋆ A nicht vollrängig ⋆ kann dünne Besetzung nicht ausnützen
SVD-Verfahren	⋆ $A \in \mathbb{R}^{m \times n}$ vollbesetzt und n klein ($m >> n$) ⋆ numerisch stabil ⋆ A muss nicht vollrängig sein	⋆ kann dünne Besetzung nicht ausnützen

Viele Professionelle Codes bauen auf SVD auf

Nicht-Lineare Ausgleichsrechnung

	Vorteile	Nachteile
Newton	⋆ quadratische Konvergenz	⋆ sehr aufwendig ⋆ Braucht zweite Ableitung
Gauss-Newton	⋆ weniger Bedingungen an F ⋆ viel einfacher	⋆ eventuelles overshooting

Viele Professionelle Codes verwenden eine gedämpfte Modifikation des Gauss-Newton Verfahrens

4 Eigenwertprobleme

Geg: $A \in \mathbb{R}^{n \times n}$
Ges: $\sigma(A) := \{\lambda : \lambda \textit{ Eigenwert von } A\}$ (Spektrum von A).

```
w , V = scipy.linalg.eig(A) # eigh fuer hermitesche Matrizen
# w: Vektor mit Eigenwerten
# V: Matrix mit V[:,i] (= i-te Spalte) Eigenvektor zum Eigenwert w[i]
```

Verwendet wird der QR-Algorithmus. Er ist langsam für grosse Matrizen ($O(n^3)$), aber ist für kleine Matrizen sehr gut. Der QR-Algorithmus kann dünne Besetzungen nicht ausnützen. Für grosse dünn besetzte Matrizen lässt sich $A^k x$ schnell berechnen, was wir im Folgenden ausnützen wollen:

4.1 Krylov-Verfahren

(L-ter) Krylov-Raum $K_l(A, v) := span\{v, Av, ..., A^{l-1}v\}$ mit $v \neq 0 \in \mathbb{R}^n$ beliebig
Idee: Man verwendet eine Projektion von $\mathbb{R}^n$ nach K_l mit $l < n$. Anschliessend den QR-Algorithmus in K_l. Dies liefert uns l<n Eigenwerte (nicht alle!).
Man verwendet das Arnoldi-Verfahren (ein modifiziertes Gram-Schmidt-Verfahren), um eine Orthonormalbasis von $K_l(A, v)$ zu finden.

Vorgehen (Arnoldi-Verfahren)
$v_1 = v/ \parallel v \parallel$
für $k = 1, 2, ..., l$:
    $\tilde{v} = Av_k$
    für $j = 1, 2, ..., k$:
        $h_{j,k} =< v_j, \tilde{v} > (= v_j^H \tilde{v})$
        $\tilde{v} = \tilde{v} - h_{j,k} v_j$
    $h_{k+1,k} =\parallel \tilde{v} \parallel$
    $v_{k+1} = \tilde{v}/h_{k+1,k}$
    Frühzeitiger Abbruch, falls: $h_{k+1,k} = 0$, da dann $K_k(A, v) = K_l(A, v)$

Daraus definieren wir $V_l := (v_1|...|v_l)$ mit $\{v_1, ..., v_l\}$ ONB von K_l und:
Hessenberg-Matrizen: $H_l := \begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,l} \\ h_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & h_{l,l-1} & h_{l,l} \end{pmatrix} \in \mathbb{R}^{l \times l}$, $\tilde{H} := \begin{pmatrix} & & H_l & \\ & & & \\ 0 & \dots & 0 & h_{l+1,l} \end{pmatrix}$
Es gilt: $H_l = V_l^H A V_l$ (Projektion in $K_l(A, v)$ mit Transformationsmatrix V_l)
Falls $A = A^H$ (A hermetisch) $\Rightarrow H_l^H = H_l \Rightarrow H_l = \begin{pmatrix} h_{1,1} & h_{1,2} & 0 \\ h_{1,2} & \ddots & \ddots \\ 0 & \ddots & \ddots \end{pmatrix}$ (Tridiagonal)
Somit hat die innere Schleife des Arnoldi-Verfahrens die konstante Länge 2. Dies führt zum Lanczos-Verfahren, bei welchem die zweite for-Schleife ersetzt wird.

Genauigkeit der Eigenwerte Seien $h_{l+1,l} = 0$, $h_{j+1,j} \neq 0$ für j = 1, ... , l-1. Ist λ ein Eigenwert von H_l (Ritz-Wert), so ist λ auch ein Eigenwert von A.
Sei hingegen $h_{j+1,j} \neq 0$ für j= 1, ... , l. Dann approximieren die Eigenwerte von H_l diejenigen von A. Je grösser l, desto besser die Approximation.

Vorgehen (Berechnung von EWen mit dem Krylov-Verfahren)
- Bestimme H_l mit dem Arnoldi-Verfahren (oder falls $A^H = A \rightarrow$ Lanczos-Verfahren)
- Bestimme EWe von H_l mit QR- Algorithmus (scipy.linalg.eig)

4.2 Eigenwertprobleme im Zusammenhang mit DGL

4.2.1 Finite Differenzen

Approximation der Ableitungsoperatoren durch Differenzialquotienten. Zwei mögliche Varianten für die erste Ableitung:
Vorwärts: $\frac{df}{dx}(x_i) \approx \frac{f(x_{i+1})-f(x_i)}{h}$ oder rückwärts: $\frac{df}{dx}(x_i) \approx \frac{f(x_i)-f(x_{i-1})}{h}$
Mögliche Approximation der zweiten Ableitung: $\frac{d^2f}{dx^2}(x_i) \approx \frac{f(x_{i+1})-2f(x_i)+f(x_{i-1})}{h^2}$

Bsp: $\frac{d^2}{dx^2}\Psi(x) = \lambda\Psi(x)$ mit unbekanntem $\lambda \in \mathbb{R}$
Randbedingung: $\Psi(a) = \Psi(b) = 0$
Idee: Partition $x_0, ...x_N$ von $[a, b] \Rightarrow \Psi(x_0) = \Psi(x_N) = 0$
Noch zu bestimmen: $\Psi(x_1), \ldots, \Psi(x_{N-1})$ und λ. Dazu:

$$\frac{d^2}{dx^2}\begin{pmatrix} \Psi(x_1) \\ \vdots \\ \vdots \\ \vdots \\ \Psi(x_{N-1}) \end{pmatrix} \approx \frac{1}{h^2}\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & & 1 \\ & & & 1 & -2 \end{pmatrix}\begin{pmatrix} \Psi(x_1) \\ \vdots \\ \vdots \\ \vdots \\ \Psi(x_{N-1}) \end{pmatrix} = \lambda\begin{pmatrix} \Psi(x_1) \\ \vdots \\ \vdots \\ \vdots \\ \Psi(x_{N-1}) \end{pmatrix}$$

Die gesuchten Lösungen $\Psi(x_1), \ldots, \Psi(x_{N-1})$ sind somit die Eigenvektoren der Matrix A und die unbestimmten Werte λ die zugehörigen Eigenwerte.
Bem: Dieses A ist dünnbesetzt $\rightarrow$ Krylov für grosse N nützlich

5 Interpolation

Geg: Daten $\{x_j, y_j\}_{j=0}^n$ und ein Modell $\tilde{f}(x) := \sum_{j=0}^n \alpha_j b_j(x)$
Ges: Koeffizienten α_j s.d. $\tilde{f}(x_j) = y_j$.
Die x_j heissen Stützstellen und $b_j(x)$ Basisfunktionen.

Anwendung:"komplizierte" Funktion $f \to$ "einfache" Funktion $\tilde{f} \approx f$ (z.B. Polynom).

Interpolationsbedingung $\begin{pmatrix} b_n(x_0) & \dots & b_0(x_0) \\ \vdots & & \vdots \\ b_n(x_n) & \dots & b_0(x_n) \end{pmatrix} \begin{pmatrix} \alpha_n \\ \vdots \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$

Die Anzahl der Stützstellen und Basisfunktionen ist gleich. Somit hat das Gleichungssystem, im Gegensatz zur Ausgleichsrechung, eine (eindeutige) Lösung.

5.1 Polynomiale Interpolation

Interpolation mit einem Polynom n-ten Grades.
Raum der Polynome von Grad $\leq$ n: $P_n := span\{1, x, x^2, ..., x^n\}$
$p \in P_n$ ist eindeutig bestimmt durch (n+1) Punkte $\{(x_j, y_j = p(x_j))\}_{j=0}^n$

5.1.1 Monombasis

Zur Bestimmung der Koeffizienten $\alpha_n, ..., \alpha_0$ ist folgendes Gleichungssystem zu lösen:
$\begin{pmatrix} x_0^n & x_0^{n-1} & \dots & 1 \\ \vdots & \vdots & & \vdots \\ x_n^n & x_n^{n-1} & \dots & 1 \end{pmatrix} \begin{pmatrix} \alpha_n \\ \vdots \\ \alpha_0 \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}$

```
c = numpy.polyfit(x,y,n)
# x,y: (n+1) Datenpunkte; n Grad des Interpolationspolynoms
# c = [alpha[n], ... , alpha[0]] faengt beim hoechsten Koeffizienten an!
ytilde = numpy.polyval(c,xtilde)
# ytilde : p ausgewertet bei xtilde
```

Vorteile: sehr anschaulich
Nachteile: Vandermonde Matrix schlecht konditioniert! Beim Hinzufügen neuer Punkte müssen alle Koeffizienten neu berechnet werden.

5.1.2 Newton-Basis

Newton-Polynome $\{N_j(x)\}_{j=0}^n$ zu den Stützstellen $\{x_j\}_{j=0}^{n-1}$ sind definiert durch:
$N_0(x) := 1$, $N_1(x) := (x - x_0)$, ..., $N_n(x) := \prod_{i=0}^{n-1}(x - x_i)$ und bilden eine Basis von P_n.
Dividierte Differenzen
$y[x_j] := y_j$, $y[x_j, ..., x_{j+k}] := \frac{y[x_{j+1},...,x_{j+k}] - y[x_j,...,x_{j+k-1}]}{x_{j+k} - x_j}$

Das Interpolationspolynom zu $\{(x_j, y_j)\}_{j=0}^n$ ist gegeben als:
$p_n(x) = y[x_0] + y[x_0, x_1](x - x_0) + ... + y[x_0, ..., x_n](x - x_0) \cdots (x - x_{n-1}) = \sum_{i=0}^n y[x_0, ..., x_i] N_i(x)$
Dies eliminiert die beiden Nachteile der Monombasis!

5.1.3 Lagrange-Basis

Lagrange-Polynome $\{l_j(x)\}_{j=0}^n$ zu den Stützstellen $\{x_j\}_{j=0}^n$ sind definiert durch:
$l_j(x) := \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}$. Sie bilden eine Basis von P_n und es gilt: $p_n(x) = \sum_{i=0}^n y_i l_i(x)$

Bem: $l_i(x_j) = \delta_{ij}$
Baryzentrische Interpolationsformel Das Interpolationspolynom zu $\{(x_j, y_j)\}_{j=0}^n$ kann auch geschrieben werden als: $p_n(x) = \frac{\sum_{k=0}^{n} \frac{\lambda_k}{x-x_k} y_k}{\sum_{k=0}^{n} \frac{\lambda_k}{x-x_k}}$ mit $\lambda_k := \prod_{j \neq k} \frac{1}{x_k - x_j}$

Beachte: Alle bisherigen Verfahren liefern das gleiche Polynom. Wir haben nur eine Darstellung bezüglich verschiedenen Basen betrachtet. Die Lagrange-Basis erleichtert uns die Berechnung der Fehler:

Fehlerschätzung Ist $f : [a,b] \to \mathbb{R} \in C^{n+1}$ und p_n das Interpolationspolynom zu $\{(x_j, y_j)\}_{j=0}^n$, dann ist $|f(x) - p_n(x)| \leq \frac{1}{(n+1)!} max_{\xi \in (a,b)} |f^{(n+1)}(\xi)(x - x_0) \cdots (x - x_n)|$

Auswirkung von Messfehlern Seien $p_n(x)$ bzw. $\tilde{p}_n(x)$ die Interpolationspolynome zu $\{(x_j, y_j)\}_{j=0}^n$ bzw. $\{(x_j, \tilde{y}_j)\}_{j=0}^n$, dann ist $|p_n(x) - \tilde{p}_n(x)| \leq \Lambda_n max_{j=0,...,n}(|y_j - \tilde{y}_j|)$ mit $\Lambda_n := max_{x \in [a,b]} \sum_{j=0}^n |l_j(x)|$.
Für äquidistante Stützstellen gilt $\Lambda_n \sim \frac{2^{n+1}}{nlog(n)}$ (exponentiell $\to \infty$, sehr schlecht)

5.2 Chebychev-Interpolation

Chebychev-Polynome 1.Art $T_n(x) := cos(n \cdot arccos(x))$, $x \in [-1, 1]$
Die T_n sind Polynome. Es gilt $T_0(x) = 1$, $T_1(x) = x$, $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$
Optimale Stützstellen Seien $\{\tilde{x}_j\}_{j=0}^n$ die (n+1) Nullstellen von $T_{n+1}(x)$ (Chebychev Knoten). Dann minimieren diese $|(x - x_0) \cdots (x - x_n)|$. Verwendet man die Chebychev Knoten als Stützstellen gilt: $\Lambda_n \sim \frac{2}{\pi} log(n)$.
Dasselbe gilt auch für die Extrema von T_n (Chebychev Abszissen).

Chebychev Knoten skaliert auf $[a,b]$: $x_k = a + \frac{1}{2}(b-a)(cos(\frac{2k+1}{2(n+1)}\pi) + 1)$
Chebychev Abszissen skaliert auf $[a,b]$: $x_k = a + \frac{1}{2}(b-a)(cos(\frac{k}{n}\pi) + 1)$

```
# Nullstellen von T_{n+1} skaliert auf [a,b]
xcheb = a + 0.5*(b-a)*(cos((arange(0,n+1)+0.5)/(n+1)*pi)+1)
```

Idee: Chebychev-Polynome als Basisfunktionen: $b_j(x) = T_j(x)$ mit Chebychev Knoten
Das Interpolationspolynom zu f und den Nullstellen $\{x_j\}_{j=0}^n$ von $T_{n+1}(x)$ als Stützstellen ist gegeben als:
$p_n(x) = \frac{1}{2}c_0 + c_1 T_1(x) + ... + c_n T_n(x)$ mit $c_k = \frac{2}{n+1} \sum_{j=0}^n f(x_j) cos(k \frac{2j+1}{n+1} \frac{\pi}{2})$
Clenshaw-Algorithmus Seien $d_{n+2} = d_{n+1} = 0$ und $d_k = c_k + 2xd_{k+1} - d_{k+2}$ für $k = n, ..., 0$, dann ist $p_n(x) = \frac{1}{2}(d_0 - d_2)$ (Auswertung durch Rückwärtsrekursion)

Trick für Auswertung auf einem beliebigen Intervall $[a,b]$:
1. Schritt: c = chebexp(y) auf $[a,b]$
2. Schritt: $\tilde{y}$ = clenshaw(c, $\frac{2\tilde{x}-a-b}{b-a}$) mit $\tilde{x} \in [a,b]$
$\Rightarrow p_n(\tilde{x}) = \tilde{y}$

```
# Chebychev-Interpolation : Code 6.6.1 (mit Cheb. Knoten) (oder Code 6.6.2 (mit Cheb. Abszissen))
(Benutzen DFT)
c = chebexp(y)
# y: (n+1) Funktionswerte mit yi = f(xi) mit xi in [a,b] den Skalierten Chebychev Knoten
# c : Koeffizienten c = (1/2 c_0, c_1, ..., c_n)
# Clenshaw-Algorithmus : Code 5.4.25
ytilde = clenshaw(c, (2 * xtilde-a-b)/(b-a))
# ytilde : pn(x) mit Koeffizienten aus c ausgewertet bei xtilde in [-1,1]
```

5.3 Trigonometrische Interpolation

5.3.1 Fourierreihe

Raum der quadratisch integrablen Funktionen:
$L^2(0,L) := \{f : (0,L) \to \mathbb{C} | \parallel f \parallel_{L^2(0,L)} < \infty\}$
Skalarprodukt auf L^2 :$< f,g >_{L^2(0,L)} := \int_0^L f(x)\bar{g}(x)dx$, $\parallel f \parallel_{L^2(0,L)} := \sqrt{< f,f >_{L^2}}$
Sei $f \in L^2(0,L)$, dann gilt: $f(x) = \sum_{k=-\infty}^{\infty} \hat{f}(k) e^{\frac{2\pi ikx}{L}}$, mit den Fourierkoeffizienten $\hat{f}(k) := \frac{1}{L}\int_0^L f(x)e^{-\frac{2\pi ikx}{L}}dx$.
Je glatter f, desto schneller fallen $\hat{f}(k)$ für $k \to \infty$ gegen 0.

Bsp: $f(x) = cos(2\pi x) + cos(4\pi x)$, $x \in [0,2]$
$\Rightarrow f(x) = \frac{1}{2}(e^{2\pi ix} + e^{-2\pi ix}) + \frac{1}{2}(e^{4\pi ix} + e^{-4\pi ix}) \stackrel{!}{=} \sum_{k=-\infty}^{\infty} \hat{f}(k) e^{\pi ikx}$
$\Rightarrow \hat{f}(k) = \begin{cases} \frac{1}{2} & , k = -4, -2, 2, 4 \\ 0 & sonst \end{cases}$

5.3.2 Diskrete Fouriertransformation

In der Praxis ist $\hat{f}(k)$ oft nicht analytisch berechenbar.
Idee: Verwende die Trapezregel (siehe Kapitel 6). Somit ist $\hat{f}(k) \approx \gamma_k := \frac{1}{N}\sum_{l=0}^{N-1} f(\frac{Ll}{N}) e^{-\frac{2\pi ikl}{N}}$
Sei dazu $\omega_N := e^{-\frac{2\pi i}{N}}$ die N-te Einheitswurzel und definiere:
Fouriermatrix $F_N := (\omega_N^{jk})_{j,k=0,\dots N-1} = \begin{pmatrix} \omega_N^0 & \dots & \omega_N^0 \\ \vdots & & \vdots \\ \omega_N^0 & \dots & \omega_N^{(N-1)^2} \end{pmatrix}$, $F_N^{-1} = \frac{1}{N}\bar{F}_N$

Diskrete Fouriertransformation (DFT) $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^n$, $\mathcal{F}(\vec{y}) := F_N \vec{y}$.
Sei $\vec{z} = \frac{1}{N}\mathcal{F}(\vec{y})$ mit $(\vec{y})_l = f(L\frac{l}{N})$, l= 0, ... , N-1, dann gilt:
für N=2n : $\gamma_k = \begin{cases} z_k & , 0 \le k \le n-1 \\ z_{k+N} & , -n \le k < 0 \end{cases}$
für N=2n+1 :$\gamma_k = \begin{cases} z_k & , 0 \le k \le n \\ z_{k+N} & , -n \le k < 0 \end{cases}$

Fast Fourier Transformation (FFT) Die DFT hat eine Laufzeitordnung von $O(N^2)$, was nicht ideal ist. Die FFT unterteilt das Problem in viele kleinere Teilprobleme und erreicht eine Laufzeitordnung $O(Nlog(N))$.

```
z = 1./N * numpy.fft.fft(y) # FFT
gamma = numpy.fft.fftshift(z) # Verschiebt die Koeffizienten, dass sie wie oben sind
# Fuer die Inverse Fouriertransformation : numpy.fft.ifft
```

5.3.3 Trigonometrische Interpolation

Idee: Verwende als Basisfunktionen der Interpolation $b_j(x) = e^{\frac{2\pi ijx}{L}}$.
Das trigonometrische Interpolatiospolynom zu $f \in L^2(0,L)$ und den äquidistanten Stützstellen $\{x_l = L\frac{l}{N}\}_{l=0}^{N-1}$ ist gegeben als:
N=2n : $f_{2n}(x) = \sum_{k=-n}^{n-1} \gamma_k e^{\frac{2\pi ikx}{L}}$
N=2n+1 : $f_{2n+1}(x) = \sum_{k=-n}^{n} \gamma_k e^{\frac{2\pi ikx}{L}}$
Bem: f_{2n} und f_{2n+1} erfüllen die Interpolationsbedingung: $f_{2n}(x_l) = f_{2n+1}(x_l) = f(x_l)$.

```
ytilde = evaliptrig(y,M) # Code 6.4.3 bzw 6.4.4
# y: 2n Funktionswerte {y_j = f(x_j)}_{j=0}^{2n-1} mit x_j ∈ [a,b] äquidistant
```

```
# ytilde: Trig. Interpol. Polynom, zu {(x_j, y_j)}_{j=0}^{2n-1} ausgewertet an M äquidistanten Punkten
auf [a,b]
```

5.4 Zusammenfassung

	Vorteile	Nachteile
Polynominterpolation (Newton Basis)	⋆ freie Wahl der Stützstellen ⋆ Punkte können später hinzugefügt werden	⋆ grosse Fehler am Rand, vor allem für grosse n
Chebychevinterpolation	⋆ optimale Stützstellen → kleine Fehler	⋆ Funktionswerte an Chebychev Knoten müssen bekannt sein
Trig. Interpolation	⋆ Zusatzinfo: Frequenzkomponenten ⋆ sehr wichtig in der Technik	

6 Numerische Quadratur

Geg: $f : [a_1, b_1] \times ... \times [a_d, b_d] \subset \mathbb{R}^d \longrightarrow \mathbb{R}$
Ges: $\int_{a_1}^{b_1} ... \int_{a_d}^{b_d} f(x_1, ..., x_d) dx_. ..dx_d$

Idee: Approximation der Funktion durch Interpolation:
$\int_a^b f(x)dx \approx Q_n(f; a, b) := \sum_{i=1}^n w_i f(c_i)$ wobei die w_i Gewichte und $c_i \in [a, b]$ die Stützstellen der Interpolation sind. Dabei handelt es sich um eine n-Punkte Quadraturformel (QF).

6.1 Grundlagen

Ordnung $Q_n(f; a, b)$ hat die Ordnung m, falls sie für alle Polynome mit Grad $\leq m - 1$ (also Polynome mit m Freiheitsgraden) exakt ist.

Fehler Quadraturfehler $E(n) := |\int_a^b f(x)dx - Q_n(f; a, b)|$.

Algebraische Konvergenz $\exists c > 0, p > 0$ s.d. $\forall n \in \mathbb{N} : E(n) \leq \frac{c}{n^p}$.
Merkmal: $E(n) \approx cn^{-p} \Leftrightarrow \log(E(n)) \approx \log(c) - p \log(n)$
$\Leftrightarrow$ Plot von E gegen n in einem LogLog-Plot ist eine Gerade

Exponentielle Konvergenz falls $\exists 0 < q < 1, c > 0$ s.d. $\forall n \in \mathbb{N} : E(n) \leq cq^n$.
Merkmal: $E(n) \approx cq^n \Leftrightarrow \log(E(n)) \approx \log(c) - n \log(q)$
$\Leftrightarrow$ Plot von E gegen n in einem LinLog-Plot ist eine Gerade

```
# n und E sind arrays mit verschiedenen n, bzw E(n)
loglog(n,E) # ≙ Gerade → alg. Konv.
semilogy(n,E) # ≙ Gerade → exp. Konv.

p = - polyfit(log(n),log(E),1)  [0] # Bei algebraischer Konvergenz
q = exp(polyfit(n,log(E),1)  [0]) # Bei exponentieller Konvergenz
```

Referenzintervalle Für manche Verfahren muss man die Funktion umskalieren:
Auf $[0, 1]$: $\int_a^b f(x)dx = (b - a) \int_0^1 f(\tilde{x})dx$ mit $\tilde{x} = (1 - x)a + xb$
Auf $[-1, 1]$: $\int_a^b f(x)dx = \frac{(b-a)}{2} \int_{-1}^1 f(\tilde{x})dx$ mit $\tilde{x} = \frac{(1-x)a+(1+x)b}{2}$

6.2 Quadratur mit äquidistanten Stützstellen

6.2.1 Newton-Cotes-Quadraturformeln

Betrachte zuerst eindimensionale $f : [a, b] \subset \mathbb{R} \longrightarrow \mathbb{R}$
Idee: Approximiere f durch ein Polynom (Polynomiale Interpolation)
Die ersten drei Newton-Cotes-Quadraturformeln sind:

Mittelpunktsregel (MR): n=1 Stützstellen; Ordnung 2
Approximation durch ein Polynom 0. Grades: $(p_0(x) = f(\frac{a+b}{2}))$
$\int_a^b f(x)dx \approx Q_1^M(f; a, b) = (b - a)f(\frac{a+b}{2})$

Trapezregel (TR): n=2 Stützstellen; Ordnung 2
Approximation durch ein Polynom 1. Grades: $(p_1(x) = f(a) + \frac{f(b)-f(a)}{b-a}(x - a))$
$\int_a^b f(x)dx \approx Q_2^T = \frac{b-a}{2}(f(a) + f(b))$

Simpsonregel (SR): n=3 Stützstellen; Ordnung 4
Approximation durch ein Polynom 2. Grades
$\int_a^b f(x)dx \approx Q_3^S = \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b))$

Allgemein gilt für die Newton-Cotes-Quadraturformeln:
Ordnung: Mindestens so gross wie die Anzahl Stützstellen n, da es sich um polynomiale Interpolation handelt. Ist n gerade, so ist die Ordnung gerade n, andernfalls n+1 aufgrund der Symmetrie.
Fehlerschätzung: $\forall f \in C^n([a,b]) : E(n) \leq \frac{1}{n!}(b-a)^{n+1} max_{x\in[a,b]}\{f^{(n)}(x)\}$

6.2.2 Zusammengesetzte Newton-Cotes-Quadraturformeln

Idee: Das Intervall wird in N Teilintervalle aufgespalten und auf jedes Teilintervall werden die Newton-Cotes Formeln angewendet: $\int_a^b f(x)dx = \sum_{i=0}^{N-1} \int_{x_i}^{x_{i+1}} f(x)dx \approx \sum_{i=0}^{N-1} Q_n(f; x_i, x_{i+1})$

zusammengesetzte MR: $\int_a^b f(x)dx \approx h \sum_{i=0}^{N-1} f(\frac{x_i+x_{i+1}}{2})$

zusammengesetzte TR: $\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b))$

zusammengesetzte SR: $\int_a^b f(x)dx \approx \frac{h}{6}(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + 4\sum_{i=1}^{N} f(\frac{x_{i-1}+x_i}{2}) + f(b))$

Für diese Formeln werden jeweils N gleichlange Teilintervalle der Länge $h = \frac{b-a}{N}$ gewählt.

Fehlerschätzung: Sei $Q_n(f; a, b)$ eine Quadraturformel der Ordnung p. Dann gilt für die dazugehörige zusammengesetzte Quadraturformel:
$\forall f \in C^p([a,b])\ \exists c > 0$ s.d. $E(N) \leq c\tilde{h}^p max_{x\in[a,b]}\{f^{(p)}\}$, wobei $\tilde{h} := max_{i=1,...N}\{h_i\}$
Für äquidistante Stützstellen: $E(N) \leq c\frac{(b-a)^p}{N^p} max_{x\in[a,b]}\{f^{(p)}\} \leq \frac{C'}{N^p}$ (alg. Konv.)

```
x, h = linspace(a,b,N+1,retstep=True) # N Teilintervalle mit Schrittweite h
mid =  0.5*(x[:-1] + x[1:]) # Mittelpunkte

# Zusammengesetzte Mittelpunktsregel:
IzMR = h * sum(f(z) for z in mid)
# Zusammengesetzte Trapezregel:
IzTR = h/2. * (f(a) + 2*sum(f(z) for z in x[1:-1]) + f(b))
# Zusammengesetzte Simpsonregel:
IzSR = h/6. * (f(a) + 2*sum(f(z) for z in x[1:-1]) + 4*sum(f(z) for z in mid) + f(b))
```

6.3 Quadratur mit nicht äquidistanten Stützstellen

6.3.1 Adaptive Quadratur

Äquidistante Stützstellen können lokale Strukturen der Funktion nicht ausnützen.
Idee: Intervalle werden dort verfeinert, wo die Funktion kompliziert ist, und somit grosse Fehler auftreten/erwartet sind. Dazu nimmt man zwei QF Q^{p_1} und Q^{p_2} unterschiedlicher Ordnungen $p_1 > p_2$. Als Mass für die lokale Komplexität der Funktion dient:
$\Delta := |Q^{p_1}(f; a, b) - Q^{p_2}(f; a, b)|$. Ist Δ gross, so ist der Fehler gross und das Intervall wird halbiert. Dies geht weiter, bis Δ kleiner als eine gewisse Toleranz ist.

6.3.2 Gauss-Legendre Quadratur

Eine Quadraturformel mit n Knoten hat 2n Freiheitsgrade, weil man die n Gewichte w_i auch frei wählen kann. Theoretisch sollte durch eine perfekte Wahl aller Koeffizienten w_i und c_i eine solche QF die Ordnung 2n erreichen können.
Für n=2,3 Stützstellen sind die QF mit optimalen Parametern auf dem Intervall [-1,1]:

Spezialfall für n=2 Stützstellen Ordnung: 4
$\int_{-1}^{1} f(x)dx \approx Q_2^G = 2 * (\frac{1}{2} f(\frac{-1}{\sqrt{3}}) + \frac{1}{2} f(\frac{1}{\sqrt{3}}))$

Spezialfall für n=3 Stützstellen Ordnung: 6
$\int_{-1}^{1} f(x)dx \approx Q_3^G = 2 * (\frac{5}{18} f(\frac{-\sqrt{15}}{5}) + \frac{8}{18} f(0) + \frac{5}{18} f(\frac{\sqrt{15}}{5}))$

Allgemein $\int_{-1}^{1} f(x)dx \approx 2 * \sum_{i=1}^{n} w_i f(c_i)$

Die Sützstellen c_i sind die EWe der Matrix: $\begin{pmatrix} 0 & b_1 & & & \\ b_1 & 0 & b_2 & & \\ & b_2 & \ddots & \ddots & \\ & & \ddots & & b_{n-1} \\ & & & b_{n-1} & 0 \end{pmatrix}, b_j = \frac{j}{\sqrt{4j^2-1}}$

Die Gewichte kann man aus den dazugehörigen Eigenvektoren gewinnen, indem man die erste Komponente der normierten Eigenvektoren quadriert. Man beachte, dass diese Formeln nur für Integrale über dem Invertall $[-1, 1]$ gelten.

6.3.3 Clenshaw-Curtis Formeln

Idee: Schreibe f durch ihre Fourierreihe um. Dies führt zur folgenden Formel:
$\int_{-1}^{1} f(x)dx = \int_0^{\pi} f(cos(\theta))sin(\theta)d\theta = \sum_{kgerade} \frac{2a_k}{1-k^2}$
Die Fourierkoeffizienten a_k lassen sich durch FFT oder DCT gewinnen. Als optimale Stützstellen der FFT/DCT werden die Extrema der Chebychev-Polynome 1.Art oder die Nullstellen der Chebychev-Polynome 2.Art gewählt.

Viele professionelle Codes verwenden eine Kombination aus diesen drei Verfahren mit nicht äquidistanten Stützstellen, um verschiedenste Integrale sehr gut zu approximieren.

```
scipy.integrate.quad(f,a,b)[0]
```

6.4 Quadratur in mehreren Dimensionen

Idee: Iteration der QF in 1D. $\int_{a_1}^{b_1} ... \int_{a_d}^{b_d} f(x_1, ..., x_d)dx_1...dx_d \approx \sum_{i_1=1}^{n_1} ... \sum_{i_d=1}^{n_d} w_{i_1}^1 ... w_{i_d}^d f(c_{i_1}^1, ..., c_{i_d}^d)$
Bem: Es handelt sich nicht um Potenzen sondern um Indizes.

Bsp: (2D) $I = \int_a^b \int_c^d f(x, y)dxdy = \int_a^b F(y)dy$
wobei $F(y) := \int_c^d f(x, y)dx \approx \sum_{i_1=1}^{n_1} w_i^1 f(c_{i_1}^1, y)$ also ist $I \approx \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} w_{i_1}^1 w_{i_2}^2 f(c_{i_1}^1, c_{i_2}^2)$

Man benötigt also insgesamt n^d Funktionsauswertungen. Insbesondere für Funktionen, die nur durch eine komplizierte Routine wie das Lösen von Differentialgleichungen gegeben sind, können die Funktionsauswertungen sehr viel Zeit kosten. Wenn der zugrunde liegende Algorithmus in einer Dimension die Konvergenzrate r hat, gilt für die Konvergenzrate des

mehrdimensionalen Verfahrens $O(N^{-\frac{r}{d}})$ Dies ist somit nur realistisch brauchbar für sehr glatte Funktionen mit einem entsprechend guten Verfahren, das die Glattheit ausnützen kann, und für relativ kleine Dimensionen.

```
scipy.integrate.nquad(f,array([[a1,b1],...,[ad,bd]]))[0]
```

6.5 Monte-Carlo-Quadratur

6.5.1 Grundverfahren

Idee: $\int_a^b f(x)dx = (b-a)\int_a^b f(x)\frac{1}{b-a}dx = (b-a)\mathbb{E}(f(x))$ mit $X \sim U([a,b])$.
Gesetz der grossen Zahlen: $\mathbb{E}(f(x)) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$ mit $x_i \in [a,b]$ zufällig und N gross.

Vorgehen
- Erzeuge N Zufallszahlen $\vec{x}_1, ..., \vec{x}_N$ aus $[a_1, b_1] \times ... \times [a_d, b_d]$
- Berechne das Integrationsvolumen $Vol := \Pi_{i=1}^{d}(b_i - a_i) = (b_1 - a_1) \cdot ... \cdot (b_d - a_d)$
- Berechne das Integral: $\int_{a_1}^{b_1} ... \int_{a_d}^{b_d} f(x)dx \approx I_N := Vol\frac{1}{N}\sum_{i=1}^{N} f(\vec{x_i})$
- Berechne den Schätzer für die Standardabweichung: $\tilde{\sigma}_N = \sqrt{\left(\frac{\frac{1}{N}(Vol^2\sum_{i=1}^{N} f(x_i)^2) - I_N^2}{N-1}\right)}$

```
#erzeugt Zufallszahlen in [0,1),
diese kann man anschliessend auf das gegebene Intervall umskalieren
x = numpy.random.rand(N,d)
```

```
#erzeugt Zufallszahlen in [a_1, b_1] × ... × [a_d, b_d]
#a=[a_1, ..., a_d] und b=[b_1, ..., b_d]
x = numpy.random.uniform(a,b,(N,d))
```

Vertrauensintervall Der exkate Wert I liegt mit ca. 68.3 % in $[I_N - \tilde{\sigma}_N, I_N + \tilde{\sigma}_N]$.

Der Wert I_N hängt natürlich von den Zufallszahlen ab. Somit erhalten wir jedes mal ein anderes Resultat, auch bei einer fix gewählten Anzahl Zufallszahlen N. Um zu sehen, um wie viel I_N zwischen verschiedenen Versuchen variiert, kann man die Berechnung von I_N weitere M male wiederholen und daraus weitere statistische Schlüsse ziehen.

Vorteil: Unabhängig von der Dimension und der Glattheit von f.
Nachteil: Sehr langsame Konvergenz ($O(N^{-\frac{1}{2}})$) $\rightarrow$ wollen Varianz reduzieren

6.5.2 Methoden zur Reduktion der Varianz

Control variates
$\int_a^b f(x)dx = \int_a^b (f(x) - \phi(x))dx + \int_a^b \phi(x)dx \approx (b-a)\frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \phi(x_i)) + I_\phi$.
Der erste Term wird mit der Monte-Carlo-Methode berechnet. Der Zweite hingegen direkt (analytisch). ϕ wird dabei so gewählt, dass $\phi(x) \approx f(x)$ und $I_\phi := \int_a^b \phi(x)dx$ möglichst einfach zu berechnen ist. Daraus folgt, dass $\tilde{\sigma}_N(f - \phi) < \tilde{\sigma}_N(f)$. Eine mögliche Wahl sind die ersten Glieder der Taylorentwicklung.
Importance sampling
$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{g(x)}g(x)dx = \mathbb{E}_g(\frac{f(x)}{g(x)}) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{g(x_i)}$ mit $x_i \sim g(x)$. g wird so gewählt, dass $\int_a^b g(x)dx = 1$ (Normierung) und $g(x) \approx \alpha f(x)$. Daraus folgt, dass $\tilde{\sigma}_N(\frac{f}{g}) < \tilde{\sigma}_N(f)$.

Dies kann man wiederum mit einigen Gliedern der Taylorreihe mal einer Normierungskonstanten bewirken.
Es ist allerdings nicht trivial Zufallszahlen $x_i \sim g(x)$ zu erzeugen. Dies kann zum Beispiel durch eine Substitution umgangen werden.

Bsp: $\int_{-1}^{1} e^{\frac{x}{2}} dx$, Taylor: $e^{\frac{x}{2}} = 1 + \frac{x}{2} + \frac{x^2}{8} + ...$
a) control variates: $\phi(x) = 1 + \frac{x}{2}$, $I_\phi = \int_{-1}^{1} \phi(x)dx = \int_{-1}^{1} 1 + \frac{x}{2}dx = 2$
$\int_{-1}^{1} e^{\frac{x}{2}} - (1 + \frac{x}{2})dx$ durch Monte-Carlo
b) importance sampling: $g(x) = \frac{\phi(x)}{2} = \frac{1}{2} + \frac{x}{4}$ (Taylorglieder mit Normierung)
Möglichkeit $x_i \sim g(x)$ zu umgehen: Substitution: $g(x)dx \stackrel{!}{=} dy$
$\rightarrow y(x) = \int_0^x g(x')dx' = \frac{1}{2}x + \frac{1}{8}x^2 \Rightarrow x = 2(-1 + \sqrt{1 + 2y})$
$\int_{-1}^{1} \frac{f(x)}{g(x)} g(x)dx = \int_{-\frac{3}{8}}^{\frac{5}{8}} \frac{2e^{-1+\sqrt{1+2y}}}{\sqrt{1+2y}} dy =: \int_{-\frac{3}{8}}^{\frac{5}{8}} h(y)dy \approx \frac{1}{N} \sum_{i=1}^{N} h(y_i)$ mit $y_i \sim U([-\frac{3}{8}, \frac{5}{8}])$.

6.6 Zusammenfassung

	Vorteile	Nachteile
Zusammengesetzte Newton-Cotes QF	⋆ sehr schnell ⋆ sehr einfach	⋆ abhängig von der Glattheit ⋆ schlecht für höhere Dimensionen ⋆ kann Struktur der Funktion nicht ausnützen
Adaptive Quadratur	⋆ adaptiv → kann lokale Glattheiten ausnützen ⋆ Kontrolle des Gesamtfehlers	⋆ schlecht für höhere Dimensionen
Gauss Quadratur	⋆ Optimale Stützstellen → maximale Ordnung ⋆ Stützstellen muss man nur einmal zu Beginn berechnen → danach sehr schnell	⋆ Funktionswerte an den optimalen Sützstellen nicht immer bekannt ⋆ Bei adaptiver Quadratur können fast keine Stützstellen im nächsten Schritt wiederverwendet werden.
Monte Carlo	⋆ dimensionsunabhängig ⋆ unabhängig von der Glattheit	⋆ nur statistischen Schätzwert ⋆ langsame Konvergenz $O(N^{-\frac{1}{2}})$

7 Gewöhnliche Differentialgleichungen

Problemstellung (Anfangswertproblem n-ter Ordnung):
Gegeben: Eine Differentialgleichung n-ter Ordnung mit Anfangswerten
(i) $\vec{y}^{(n)} = \vec{f}(t, \vec{y}, ..., \vec{y}^{(n-1)})$
(ii) $\vec{y}(t_0) = \vec{y}_0,\ ...,\ \vec{y}^{(n-1)}(t_0) = \vec{y}_0^{(n-1)}$
Gesucht: Die (eindeutige) Lösung y(t) des Anfangswertproblems für $t\epsilon[t_0, t_{End}]$

Idee: Wir unterteilen das Intervall $[t_0, t_{End}]$ in N Stücke mit N+1 Randpunkten $t_0, t_1, ..., t_N$ und approximieren $\vec{y}$ durch N+1 Punkte $\vec{y}_0, \vec{y}_1, ..., \vec{y}_N$, wobei $\vec{y}_k \approx \vec{y}(t_k)$ gelten sollte.

7.1 Grundlagen

7.1.1 Umwandlung in Differentialgleichung erster Ordnung

Die meisten Verfahren sind für Differentialgleichungen erster Ordnung gedacht. Man kann aber jede Differentialgleichung höherer Ordnung als Vektor umschreiben, und sie so als Differentialgleichung erster Ordnung darstellen.
Gegeben: $\vec{y}^{(n)} = \vec{f}(t, \vec{y}, ..., \vec{y}^{(n-1)})$ mit $\vec{y} \in \mathbb{R}^m$
Vorgehen: Man definiere $\vec{z} \in \mathbb{R}^{n \cdot m}$ wie folgt:
1. Schritt: Schreibe einen Vektor mit Einträgen $\vec{y}(t)...\vec{y}^{(n-1)}(t)$

$$\vec{z}(t) = \begin{pmatrix} \vec{z}_0(t) \\ \vdots \\ \vec{z}_{n-1}(t) \end{pmatrix} := \begin{pmatrix} \vec{y}(t) \\ \vdots \\ \vec{y}^{(n-1)}(t) \end{pmatrix}$$

2. Schritt: Leite diesen Vektor ab und setze die Differentialgleichung ein:

$$\frac{d}{dt}\vec{z}(t) = \begin{pmatrix} \vec{y}^{'}(t) \\ \vdots \\ \vec{y}^{(n)}(t) \end{pmatrix} = \begin{pmatrix} \vec{z}_1(t) \\ \vdots \\ \vec{z}_{n-1}(t) \\ \vec{f}(t, \vec{y}, ..., \vec{y}^{(n-1)}) \end{pmatrix} =: \vec{g}(t, \vec{z})$$

Somit ist $\vec{y}^{(n)} = \vec{f}(t, \vec{y}, ..., \vec{y}^{(n-1)}) \Leftrightarrow \dot{\vec{z}}(t) = \vec{g}(t, \vec{z})$ (ODE erster Ordnung)
Wir schreiben ab jetzt für die Ableitung dieses verallgemeinerten Vektors $\dot{\vec{y}}$ bzw. $\dot{\vec{z}}$.
Bsp: $y^{''} = 4(y^{'} - y) = f(y)$ (Lineare ODE zweiter Ordnung)

$$\vec{z}(t) = \begin{pmatrix} z_1(t) \\ z_2(t) \end{pmatrix} := \begin{pmatrix} y(t) \\ y^{'}(t) \end{pmatrix} \text{ und } \dot{\vec{z}}(t) = \begin{pmatrix} y^{'}(t) \\ y^{''}(t) \end{pmatrix} = \begin{pmatrix} z_2(t) \\ 4(z_2(t) - z_1(t)) \end{pmatrix} =: \vec{g}(z_1, z_2).$$

7.1.2 Autonomisierung einer Differentialgleichung

Hat man eine DGL erster Ordnung $\dot{\vec{y}} = \vec{f}(t, \vec{y})$ mit $\vec{y} \in \mathbb{R}^n$, so kann man die Zeitabhängigkeit in die Differentialgleichung einpacken. Das Vorgehen ist genau analog zur Umwandlung in eine DGL erster Ordnung. Dazu wird $\vec{z} \in \mathbb{R}^{n+1}$ eingeführt. $\vec{z}(t) := \begin{pmatrix} \vec{y}(t) \\ t \end{pmatrix} = \begin{pmatrix} \tilde{\vec{z}} \\ z_{n+1} \end{pmatrix}$
Somit erhalten wir die autonome Differentialgleichung:

$$\dot{\vec{z}}(t) = \begin{pmatrix} \vec{f}(z_{n+1}, \tilde{\vec{z}}) \\ 1 \end{pmatrix} =: \vec{g}(\vec{z})$$

7.1.3 Konvergenzordnung eines Verfahrens (analog zu Quadratur)

Wir approximieren $\vec{y}(t)$ auf $[t_0, t_{End}]$ mit N Punkten durch ein Verfahren. Dann approximiert $\vec{y}_N$ den Funktionswert zum Zeitpunkt t_{end}. Wir definieren:
Fehler zum Endzeitpunkt $e(N) := \| \vec{y}_{exact}(t_{end}) - \vec{y}_N \|$

Konvergenzordnung p des Verfahrens $p > 0$ s.d.
$\forall N \in \mathbb{N} : E(N) \leq \frac{c}{N^p}$ bzw. $E(h) \leq ch^p$ mit $h := \frac{t_{end}-t_0}{N}$ und einer Konstanten c
Berechnung: $E(N) \approx cN^{-p} \Leftrightarrow \log(E(N)) \approx \log(c) - p\log(N)$
$\Leftrightarrow$ Plot von E gegen N in einem LogLog-Plot ist eine Gerade mit Steigung (-p)

Vorgehen (Bestimmung der Konvergenzordnung)
- Verwende das Verfahren und löse die DGL mehrmals für verschiedene N
- Berechne für jedes N: $e(N) :=\| \vec{y}_{exact}(t_{end}) - \vec{y}_N \|$
- Berechne p = -polyfit(log(N),log(e),1)[0] = polyfit(log(h),log(e),1)[0]

7.1.4 Erhaltungsgrössen

Eine Abbildung $I : \mathbb{R}^n \to \mathbb{R}^n$ heisst erstes Integral (oder Invariante bzw. Erhaltungsgrösse) vom AWP $\dot{\vec{y}}(t) = \vec{f}(t, \vec{y})$ $(\star)$, falls gilt: Für alle Lösungen $\vec{y}(t)$ von $(\star)$ ist $I(\vec{y})(t) = const.$
Es gilt: I erstes Integral von $(\star) \Leftrightarrow grad(I(\vec{y}))\vec{f}(t, \vec{y}) = 0$

Bsp 1: Norm der Lösung:
Für manche Problemstellungen ist die Norm der Lösung eine Erhaltungsgrösse:
Trick: $\frac{d}{dt} \| \vec{y}(t) \| = 0 \Leftrightarrow \frac{d}{dt} \| \vec{y}(t) \|^2 = 0$.
$\frac{d}{dt} \| \vec{y}(t) \|^2 = \frac{d}{dt} (\vec{y}(t) \cdot \vec{y}(t)) = 2 \cdot \vec{y}(t) \cdot \dot{\vec{y}}(t) \stackrel{!}{=} 0 \Leftrightarrow \vec{y}(t) \perp \dot{\vec{y}}(t)$
Die Ableitung gegeben durch die Funktion f muss also senkrecht zur Position stehen!

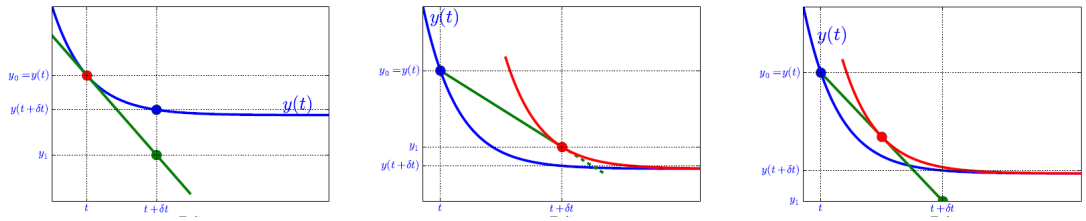
Bsp 2: Hamiltonsche Systeme:
Sei $H : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ mit $H = H(q, p)$ stetig differenzierbar, dann ist das autonome Hamilton-System das folgende System von Differentialgleichungen:

$$\begin{cases} \dot{q}(t) = \frac{\partial H}{\partial p}(q, p) \\ \dot{p}(t) = -\frac{\partial H}{\partial q}(q, p) \end{cases}$$

Die Hamilton Funktion H ist ein erstes Integral des dazugehörigen autonomen Hamilton-Systems. Die mehrdimensionale Verallgemeinerung der Hamilton-Systeme lautet:

$$\begin{cases} \dot{\vec{q}}(t) = \nabla_p H(\vec{q}, \vec{p}) \\ \dot{\vec{p}}(t) = -\nabla_q H(\vec{q}, \vec{p}) \end{cases}$$

Zusätzlich kann der Hamiltonian $H(q, p, t)$ explizit von der Zeit abhängen. Die Differentialgleichungen bleiben gleich, doch dann ist H nicht mehr erhalten.



(a) explizites Eulerverfahren (b) implizites Eulerverfahren (c) implizite Mittelpunktsregel

Abbildung 5

7.2 Polygonzugverfahren

Geg: $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, $\vec{y}(t_0) = \vec{y}_0$
Ges: Lösung y(t) des AWP 1.Ordnung

7.2.1 explizites Eulerverfahren (eE)

Approximation durch Tangente zum Anfangszeitpunkt t_k (Abbildung 5a)
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(t_k, \vec{y}_k)$

```
y = zeros((N+1,d)) # d = Dimension der Vektoren
y[0,:] = y0 # Startwert initialisieren
t, h = linspace(tstart,tend,N+1, retstep=True)
for k in range(N):
    y[k+1,:] = y[k,:]+h*f(t[k],y[k,:])
```

7.2.2 implizites Eulerverfahren (iE)

Approximation durch Tangente zum nächsten Zeitpunkt t_{k+1} (Abbildung 5b)
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(t_{k+1}, \vec{y}_{k+1})$
$\Rightarrow \vec{y}_{k+1}$ Nullstelle von: $F(\vec{X}) := \vec{X} - \vec{y}_k - h_k \vec{f}(t_{k+1}, \vec{X}))$
Guter Startwert für Nullstellensuche: Schritt aus eE
Implizit bedeutet, dass für $\vec{y}_{k+1}$ eine (i.a. nichtlineare) Gleichung aufgelöst werden muss.

```
for k in range(N):
    F = lambda x: x-y[k,:]-h*f(t[k+1],x)
    y[k+1,:] = fsolve(F,y[k,:]+h*f(t[k],y[k,:])
```

7.2.3 implizites Mittelpunktsverfahren (iM)

Approximation durch Tangente zum Zeitpunkt $(t_k + t_{k+1})/2$ (Abbildung 5c)
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\vec{y}_k + \vec{y}_{k+1}))$
$\Rightarrow \vec{y}_{k+1}$ Nullstelle von: $F(\vec{X}) := \vec{X} - \vec{y}_k - h_k \vec{f}(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\vec{y}_k + \vec{X}))$
Guter Startwert für Nullstellensuche: Schritt aus eE

```
for k in range(N):
    F = lambda x: x - y[k,:] - h*f(0.5*(t[k] + t[k+1]), 0.5*(y[k,:] + x))
    y[k+1,:] = fsolve(F, y[k,:] + h*f(t[k],y[k,:]))
```

Bei iM ist im Gegensatz zu eE und iE die Gesamtenergie erhalten! Die Konvergenzordnungen der Verfahren sind $O(h)$ für die Eulerverfahren und $O(h^2)$ für iM.

7.3 Störmer-Verlet-Verfahren

Geg: $\ddot{\vec{y}} = \vec{f}(t, \vec{y})$ (keine $\dot{\vec{y}}$ Abhängigkeit!), $\vec{y}(t_0) = \vec{y}_0$, $\dot{\vec{y}}(t_0) = \vec{v}_0$
Ges: Lösung $\vec{y}(t)$ des AWP 2. Ordnung

7.3.1 Zwei-Schritt Formulierung

Idee: Approximation durch eine Parabel und äquidistante t_k
$\vec{y}_{k+1} := -\vec{y}_{k-1} + 2\vec{y}_k + h^2 f(t_k, \vec{y}_k)$
mit zweitem Startwert $\vec{y}_1 = \vec{y}_0 + h\vec{v}_0 + \frac{1}{2}h^2\vec{f}(t_0, \vec{y_0})$.

```
y[0,:] = y0
y[1,:] = y0+h*v0 + 0.5*h**2*f(t0,y0)
for k in xrange(1,N):
    y[k+1,:] = - y[k-1,:] + 2*y[k,:] + h**2*f(t[k],y[k,:])
```

7.3.2 Ein-Schritt Formulierung

Idee: $\ddot{\vec{y}} = \vec{f}(t, \vec{y}) \Leftrightarrow \dot{\vec{y}} = \vec{v}$ und $\dot{\vec{v}} = \vec{f}(t, \vec{y})$.
Verwende das explizite Eulerverfahren für $\dot{\vec{y}}(t) = \vec{v}(t)$ und $\dot{\vec{v}}(t) = \vec{f}(t, \vec{y})$

Leap-Frog-Methode

$\vec{v}_{k+\frac{1}{2}} := \vec{v}_{k-\frac{1}{2}} + h\vec{f}(t_k, \vec{y}_k)$
$\vec{y}_{k+1} := \vec{y}_k + h\vec{v}_{k+\frac{1}{2}}$
mit Startwert $\vec{v}_{\frac{1}{2}} = \vec{v}_0 + \frac{h}{2}\vec{f}(t_0, \vec{y}_0)$

```
y[0,:] = y0
vtemp = v0 + 0.5 * h * f(t0,y0)
for k in range(N):
    y[k+1,:] = y[k,:] + h * vtemp
    vtemp += h * f(t[k+1],y[k+1,:])
```

Velocity-Verlet

$\vec{y}_{k+1} := \vec{y}_k + h\vec{v}_k + \frac{h^2}{2}\vec{f}(t_k, \vec{y}_k)$
$\vec{v}_{k+1} := \vec{v}_k + \frac{h}{2}(\vec{f}(t_k, \vec{y}_k) + \vec{f}(t_{k+1}, \vec{y}_{k+1}))$

```
y[0,:] = y0
v[0,:] = v0
for k in range(N):
    y[k+1,:] = y[k,:] + h * v[k,:] + 0.5 * h**2 * f(t[k],y[k,:])
    v[k+1,:] = v[k,:] + 0.5 * h * (f(t[k],y[k,:]) + f(t[k+1],y[k+1,:]))
```

Im Gegensatz zu Leap-Frog liefert Velocity-Verlet die Geschwindigkeiten bei t_k.
Die Einschritt-Verfahren sind numerisch stabiler.
In allen Störmer-Verlet-Verfahren ist die Energie erhalten.

7.4 Splittingverfahren

Evolutionsoperator $\Phi^{t_0,t} : D \subseteq \mathbb{R}^n \to D$ heisst Evolutionsoperator zur DGL $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, wenn $\forall\, \vec{y}_0 \in D$ gilt: $\Phi^{t_0,t}\vec{y}_0 = \vec{y}(t)$ eine Lösung des AWP $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, $\vec{y}(t_0) = \vec{y}_0$ ist.
Für autonome ODE gilt: $\Phi^{t_0,t} = \Phi^{0,t-t_0} =: \Phi^{t-t_0} =: \Phi^h$
Numerische Verfahren liefern den diskreten Evolutionsoperator $\Psi^h \approx \Phi^h$.

Bsp 1: $\dot{y} = \lambda y \Rightarrow y(t) = e^{\lambda(t-t_0)}y_0 =: e^{\lambda h}y_0 \Rightarrow \Phi^h y_0 = e^{\lambda h}y_0$.
eE liefert: $y(t_0 + h) \approx (1 + \lambda h)y_0 \Rightarrow \Psi^h = 1 + \lambda h$, also der Taylor 1. Ordnung von Φ^h
Bsp 2: $\dot{y} = y^2 \Rightarrow y(t) = \frac{y_0}{1-ty_0} \Rightarrow \Phi^h y_0 = \frac{y_0}{1-hy_0}$

Geg: Autonome, separierte Differentialgleichung $\dot{\vec{y}} = \vec{f_a}(\vec{y}) + \vec{f_b}(\vec{y})$, $\vec{y}(t_0) = \vec{y}_0$.
Mit bekannten Evolutionsoperatoren Φ_a^h zu $\dot{\vec{y}} = \vec{f_a}(\vec{y})$ und Φ_b^h zu $\dot{\vec{y}} = \vec{f_b}(\vec{y})$
Ges: Lösung $\vec{y}(t)$ des AWP 1. Ordnung
Idee: Zerlege ein kompliziertes Problem in zwei Teilprobleme.

Lie-Trotter-Splitting $\Psi_1^h = \Phi_a^h \circ \Phi_b^h$ oder $\Psi_1^h = \Phi_a^h \circ \Phi_b^h$.
Strang-Splitting $\Psi_2^h = \Phi_a^{h/2} \circ \Phi_b^h \circ \Phi_a^{h/2}$ oder $\Psi_2^h = \Phi_b^{h/2} \circ \Phi_a^h \circ \Phi_b^{h/2}$
Allgemein $\Psi_s^h = \prod_{i=1}^s \Phi_a^{a_i h} \circ \Phi_b^{b_i h}$ mit $\sum_{i=1}^s a_i = \sum_{i=1}^s b_i = 1$.
Bem: Das Splittingverfahren kann als Verallgemeinerung des Störmer-Verlet Verfahrens betrachtet werden. Sind Φ_a und Φ_b nicht bekannt, können diskrete Evolutionsoperatoren Ψ_a und Ψ_b verwendet werden.

Bsp 3: $\dot{y} = \lambda y + y^2 =: f_a(y) + f_b(y) \Rightarrow \Phi_a^h y_0 = e^{\lambda h}y_0, \Phi_b^h y_0 = \frac{y_0}{1-hy_0}$ (Bsp. 1 und 2)
$\Psi_{1,AB}^h y_0 = (\Phi_a^h \circ \Phi_b^h)y_0 = \Phi_a^h(\frac{y_0}{1-hy_0}) = e^{\lambda h}\frac{y_0}{1-hy_0}$
$\Psi_{1,BA}^h y_0 = (\Phi_b^h \circ \Phi_a^h)y_0 = \Phi_b^h(e^{\lambda h}y_0) = \frac{e^{\lambda h}y_0}{1-he^{\lambda h}y_0}$
Die beiden Varianten von Lie-Trotter sind also nicht identisch!

Bsp 4: Wichtigstes Beispiel in der Physik: Hamiltonsche Systeme
In der Mechanik hat der Hamiltonian oft folgende Form: $H(q,p,t) = T(p) + V(q,t)$ Dann:

$$\frac{d}{dt}\vec{z} := \frac{d}{dt}\begin{pmatrix} q \\ p \\ t \end{pmatrix} = \begin{pmatrix} \frac{\partial H}{\partial p}(q,p,t) \\ -\frac{\partial H}{\partial q}(q,p,t) \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{dT}{dp}(p) \\ -\frac{\partial V}{\partial q}(q,t) \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{dT}{dp}(p) \\ 0 \\ 1 \end{pmatrix}}_{:=f_a(\vec{z})} + \underbrace{\begin{pmatrix} 0 \\ -\frac{\partial V}{\partial q}(q,t) \\ 0 \end{pmatrix}}_{:=f_b(\vec{z})}$$

Beide Teile separat exakt lösen:

$$\frac{d}{dt}\begin{pmatrix} q \\ p \\ t \end{pmatrix} = \begin{pmatrix} \frac{dT}{dp}(p) \\ 0 \\ 1 \end{pmatrix} \quad \Rightarrow \quad \Phi_a^h \begin{pmatrix} q_0 \\ p_0 \\ t_0 \end{pmatrix} = \begin{pmatrix} q_0 + h\frac{dT}{dp}(p_0) \\ p_0 \\ t_0 + h \end{pmatrix}$$

$$\frac{d}{dt}\begin{pmatrix} q \\ p \\ t \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{\partial V}{\partial q}(q,t) \\ 0 \end{pmatrix} \quad \Rightarrow \quad \Phi_b^h \begin{pmatrix} q_0 \\ p_0 \\ t_0 \end{pmatrix} = \begin{pmatrix} q_0 \\ p_0 - h\frac{\partial V}{\partial q}(q_0,t_0) \\ t_0 \end{pmatrix}$$

Erste Variante des Lie-Trotter-Splittings:

$$\Psi_1^h \begin{pmatrix} q_0 \\ p_0 \\ t_0 \end{pmatrix} = (\Phi_a^h \circ \Phi_b^h) \begin{pmatrix} q_0 \\ p_0 \\ t_0 \end{pmatrix} = \Phi_a^h \begin{pmatrix} q_0 \\ p_0 - h\frac{\partial V}{\partial q}(q_0,t_0) \\ t_0 \end{pmatrix} = \begin{pmatrix} q_0 + h\frac{dT}{dp}(p_0 - h\frac{\partial V}{\partial q}(q_0,t_0)) \\ p_0 - h\frac{\partial V}{\partial q}(q_0,t_0) \\ t_0 + h \end{pmatrix}$$

7.5 Runge-Kutta-Verfahren

Geg: $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, wobei $\vec{y}(t_0) = \vec{y}_0$
Ges: Lösung $\vec{y}(t)$ des AWP 1. Ordnung

Idee: Schreibe das Differentialgleichungsproblem in ein Integralproblem um $\vec{y}(t_1) = \vec{y}(t_0) + \int_{t_0}^{t_1} \vec{f}(t, \vec{y}(t))dt$ und verwende eine Quadraturformel zur Approximation des Integrals (siehe Abschnitt 6) : $\vec{y}(t_1) \approx \vec{y}_0 + h \sum_{i=1}^{s} b_i \vec{f}(t_0 + c_i h, \vec{y}(t_0 + c_i h))$ mit Gewichten b_i und Stützstellen $c_i \in [0, 1]$ und $h = t_1 - t_0$.

Einfache Beispiele

Explizite Mittelpunktsregel Wir wählen die Mittelpunktsregel als QF. Der noch unbekannte Funktionswert in der Mitte $y(t_0 + h/2.)$ wird durch eE approximiert.

```
for i in range(N):
    y[i+1,:] = y[i,:] + h*f(t[i]+h/2., y[i,:]+h/2.*f(t[i],y[i,:]))
```

Mit der Notation für allgemeine Runge-Kutta Verfahren lässt sich dies schreiben als: $\vec{k}_1 := \vec{f}(t_0, \vec{y}(t_0))$ und $\vec{k}_2 := \vec{f}(t_0 + \frac{h}{2}, \vec{y}_0 + \frac{h}{2}\vec{k}_1)$. Dann ist $\vec{y}_1 = \vec{y}_0 + h\vec{k}_2$.

Explizite Trapezregel: Wir wählen die Trapezregel als QF. Der noch unbekannte Funktionswert $y(t_0 + h)$ wird durch eE approximiert.
$\vec{k}_1 := \vec{f}(t_0, \vec{y}(t_0))$ und $\vec{k}_2 := \vec{f}(t_0 + h, \vec{y}(t_0) + h\vec{f}(t_0, \vec{y}_0))$, dann ist $\vec{y}_1 = \vec{y}_0 + \frac{h}{2}(\vec{k}_1 + \vec{k}_2)$

```
for i in range(N):
        k1 = f(t[i], y[i,:])
        k2 = f(t[i]+h, y[i,:]+h*k1)
        y[i+1,:] = y[i,:] + h/2.*(k1+k2)
```

Polygonzugverfahren: Die drei Polygonzugverfahren sind jeweils s=1-stufige Runge-Kutta Verfahren und sind somit die drei einfachsten Beispiele.

s-stufiges Runge-Kutta-Verfahren

Gegeben sind b_i und a_{ij} in $\mathbb{R}$ mit i,j = 1, ..., s. Dann ist $c_i := \sum_{j=1}^{s} a_{ij}$ und:

Vorgehen (Allgemeines Runge-Kutta-Verfahren)
Berechne: $\vec{k}_i := \vec{f}(t_0 + c_i h, \vec{y}_0 + h \sum_{j=1}^{s} a_{ij}\vec{k}_j)$ i = 1, 2, ..., s (Inkremente)
Daraus: $\vec{y}_{j+1} = \vec{y}_j + h \sum_{i=1}^{s} b_i \vec{k}_i$

Butcher-Tableau Zur Darstellung eines Runge-Kutta-Verfahrens:

$$\begin{array}{c|ccc} c_1 & a_{11} & \dots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \dots & a_{ss} \\ \hline & b_1 & \dots & b_s \end{array} =: \begin{array}{c|c} \vec{c} & \mathrm{A} \\ \hline & \vec{b}^T \end{array}$$

Ein Runge-Kutta-Verfahren heisst:

- **explizit**, falls A eine strikte untere Dreiecksmatrix ist. Da sich jedes k_i aus den früher berechneten k_i's bestimmen lässt, müssen keine impliziten Gleichungen gelöst werden.
- **diagonal implizit**, falls A eine nicht-strikte untere Dreiecksmatrix ist. Es kann nacheinander eine Gleichung für das neuste k_i aufgestellt und gelöst werden.

- **implizit**, sonst. Es muss eine grosse Gleichung mit allen k_i's als Unbekannten auf einen Schlag gelöst werden. Dies benötigt den grössten und kompliziertesten Rechenaufwand.

Beispiele:

Expliter Euler: $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ Impliziter Euler: $\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$ Implizite Mittelpunktsregel: $\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$

Explizite Trapezregel: $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ Explizite Mittelpunktsregel : $\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$

Gauss-Kollokation

Wie bei der Gauss-Legendre Quadratur (Kapitel 6) kann man sich fragen, was für ein gegebenes s die maximal mögliche Ordnung eines Runge-Kutta Verfahrens ist. Es gilt:

- Explizite Verfahren: Ordnung p $\leq$ s
- Implizite Verfahren: Ordnung p $\leq$ 2s

Die Gauss-Kollokation liefert uns ein implizites Verfahren, das für ein gegebenes s die maximale Ordnung 2s liefert. Das Runge-Kutta Verfahren selber wird also ganz normal durchgeführt. Wir verwenden lediglich ein ganz bestimmtes Butcher-Tableau.

7.6 Adaptive Verfahren

In den meisten bisherigen Verfahren wurden konstante Schrittweiten h verwendet.
Sinnvoller: Kleinere Schritte, wo die Funktion kompliziert ist und somit grosse Fehler in der Approximation auftreten.
Dazu verwendet man zwei Evolutionsoperatoren (also zwei Verfahren) mit unterschiedlicher Ordnung. Sei die Ordnung von $\tilde{\Psi}$ grösser als die von Ψ. Ist die Differenz beider Verfahren: $\Delta := \| \tilde{\Psi}^{t,t+h} y(t_k) - \Psi^{t,t+h} y(t_k) \|$ kleiner als eine vorgegebene Toleranz (kleiner Fehler), so kann die Schrittweite h dort akzeptiert werden und um einen Faktor vergrössert werden. Ist Δ grösser als die Toleranz (grosser Fehler) wird die Schrittweite halbiert.

ode45

Als Referenzalgorithmus wird oft das Verfahren ode45 verwendet. Dies ist ein professionell entwickelter Algorithmus. Es ist ein adaptives Verfahren basierend auf zwei expliziten Runge-Kutta-Verfahren der Ordnung 4 und 5.

```
t,y = ode45(f,[t0,tEnd],y0)
# Input: f: Funktion f(t,y) der zu lösenden ODE 1.Ordnung y' = f(t,y)
# t0,tend = Start- und Endzeit; y0: Anfangswert y0
# Output: t: Zeiten t0,...,tend
# y: Approximierte Funktionswerte. y[k,:] = Lösungsvektor zum Zeitpunkt t[k]
```

Man beachte, dass die Funktion f von der Zeit t abhängen muss. Wenn wir also eine autonome Differentialgleichung der Form $\dot{\vec{y}} = \vec{f}(\vec{y})$ haben, müssen wir eine formelle Zeitabhängigkeit hinzufügen: $\dot{\vec{y}} = \vec{g}(t, \vec{y})$ mit $\vec{g}(t, \vec{y}) := \vec{f}(\vec{y})$.

```
g = lambda t,y: f(y) # genau gleiche Funktion, aber formell zeitabhaengig
```

8 Steife Differentialgleichungen

8.1 Definition

Eine Differentialgleichung ist steif, falls sich das Verhalten der numerischen Lösung eines expliziten Verfahrens ab einem bestimmten N bzw. h komplett verändert. (typischerweise: Konvergenz $\rightarrow$ Divergenz)

Bsp 1: Die Testgleichung $\dot{y} = \lambda y =: f(t,y) = f(y)$
Exakte Lösung: $y(t) = y_0 e^{\lambda t}$ für $\lambda < 0$ gilt somit: $|y(t)| \overset{t\to\infty}{\longrightarrow} 0$
Numerische Lösung durch eE: $y_k = y_{k-1} + hf(t_{k-1}, y_{k-1}) = (1+h\lambda)y_{k-1} = (1+h\lambda)^k y_0$
$|y_k| \overset{k\to\infty}{\longrightarrow} \left\{ \begin{array}{ccc} \infty & , & |1+h\lambda| > 1 \\ |y_0| & , & |1+h\lambda| = 1 \\ 0 & , & |1+h\lambda| < 1 \end{array} \right\}$
Korrekte Konvergenz nur für $|1+h\lambda| < 1 \Leftrightarrow h \in \left]0, \frac{2}{|\lambda|}\right[$
Mit $z := h\lambda$ können wir auch schreiben: $|1+z| < 1 \Leftrightarrow z \in]-2, 0[$

Bsp 2: Mehrdimensionales Lineares AWP:
Geg: $\left\{ \begin{array}{c} \dot{u} = 998u + 1998v \\ \dot{v} = -999u - 1999v \end{array} \right\}$ mit $\begin{pmatrix} u(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} =: \vec{z}_0$, also in Matrixform:
$\frac{d}{dt}\vec{z} = \frac{d}{dt}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} = A\begin{pmatrix} u \\ v \end{pmatrix} = A\vec{z}$
Exakte Lösung: $\begin{pmatrix} u(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} 2e^{-t} - e^{-1000t} \\ -e^{-t} + e^{-1000t} \end{pmatrix}$
Numerische Lösung mit eE: $\vec{z}_k = (1+hA)^k \vec{z}_0$
Trick: $\vec{z}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \vec{v}_1 - \vec{v}_2$ mit den Eigenvektoren $\vec{v}_1 = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$ und $\vec{v}_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$
zu den Eigenwerten $\lambda_1 = -1$ und $\lambda_2 = -1000$. $\Rightarrow \vec{z}_k = (1+hA)^k\vec{v}_1 - (1+hA)^k\vec{v}_2 = (1+h\lambda_1)^k\vec{v}_1 - (1+h\lambda_2)^k\vec{v}_2 = (1-h)^k\vec{v}_1 - (1-1000h)^k\vec{v}_2$
Damit: $\| \vec{z}_k \| \overset{k\to\infty}{\longrightarrow} \left\{ \begin{array}{ccc} \infty & , & h > 2/1000 = 0.002 \\ \| \vec{v}_2 \| = \sqrt{2} & , & h = 0.002 \\ 0 & , & h < 0.002 \end{array} \right\}$
Somit muss für eE eine Schrittweite h<0.002 gewählt werden.

Vorgehen (Wie zeigt man, dass eine DGL steif ist?)
Wir müssen zeigen, dass sich ab einem h bzw. N das Verhalten komplett verändert:

- **Analytisch:** Direkt die Definition eines expliziten Verfahrens (hier am einfachsten: eE) verwenden und dies soweit umformen, bis man eine Fallunterscheidung des Konvergenzverhaltens in Abhängigkeit von h hat.
- **Numerisch:** Man implementiert ein beliebiges explizites Verfahren (am besten: eM, eE geht auch) und probiert verschiedene N bzw. h aus. (z.B. N = 10 bis 10^6)

Bei der numerischen Variante: y-Achse einschränken, dass die exakte Lösung sinnvoll darin liegt. (exakte Lösung analytisch oder mit impliziten Verfahren, am besten iM, berechnen)

8.2 Stabilitätsbegriffe

Testgleichung: Die eindimensionale DGL $\dot{y} = \lambda y =: f(y)$ mit $\lambda \in \mathbb{R}$ (oder allgemeiner: $\lambda \in \mathbb{C}$) heisst die Testgleichung. Damit definieren wir:

Stabilitätsfunktion: Die Funktion $S : D \subset \mathbb{C} \to \mathbb{C}$ heisst Stabilitätsfunktion eines Verfahrens, falls für einen Zeitschritt des Verfahrens angewandt auf die Testgleichung $\dot{y} = \lambda y$ gilt: $y_{k+1} = S(z)y_k$ mit $z := \lambda \cdot h$.
Formeller: Sei Ψ_λ^h die diskrete Evolution eines Einschrittverfahrens zum gegebenen AWP mit der Testgleichung $\dot{y} = \lambda y$. Dann muss gelten: $\Psi_\lambda^h y = S(z)y$ mit $z := \lambda \cdot h$.
Stabilitätsgebiet $S_\Psi := \{z \in D : |S(z)| < 1\} \subset \mathbb{C}$.
Für ein s-stufiges Runge-Kutta-Einschrittverfahren mit Butcher-Tableau $\begin{array}{c|c} \vec{c} & \mathrm{A} \\ \hline & \vec{b}^T \end{array}$ gilt:

$S(z) = \frac{det(\mathbb{I} - zA + z\vec{1}\cdot\vec{b}^T)}{det(\mathbb{I} - zA)}$, wobei $\vec{1} := (1, ..., 1)^T \in \mathbb{R}^n$.

Bsp 3: Stabilitätsfunktion und Stabilitätsgebiet von eE: Tableau = $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$

$y_{k+1} = y_k + hf(y_k) = y_k + h\lambda y_k = (1 + h\lambda)y_k =: (1 + z)y_k \overset{!}{=} S(z)y_k \Rightarrow S(z) = 1 + z$
Oder mit der Formel: $S(z) = \frac{det(1 - z\cdot 0 + z\cdot 1\cdot 1)}{det(1 - z\cdot 0)} = \frac{det(1+z)}{det(1)} = 1 + z$

Stabilitätsgebiet: $|S(z)| = |1 + z| = |z - (-1)| \overset{!}{<} 1$
Somit: $S_\Psi = \{z \in \mathbb{C} : |z - (-1)| < 1\}$ der Kreis um (-1) mit Radius 1.
Für $z \in \mathbb{R}$ gilt: $S_\Psi \cap \mathbb{R} =]-2, 0[$ wie wir bereits gesehen haben.

Bsp 4: Stabilitätsfunktion und Stabilitätsgebiet von iM: Tableau = $\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$

$S(z) = \frac{det(1 - z\frac{1}{2} + z\cdot 1\cdot 1)}{det(1 - z\frac{1}{2})} = \frac{1+z/2}{1-z/2}$ Es gilt: $|S(z)| < 1 \Leftrightarrow |S(z)|^2 < 1$

$|S(z)|^2 = S(z) \cdot \overline{S(z)} = \frac{1+z/2+\bar{z}/2+z\bar{z}/4}{1-z/2-\bar{z}/2+z\bar{z}/4} \overset{!}{<} 1 \Leftrightarrow z + \bar{z} < 0 \Leftrightarrow Re(z) < 0$
Somit: $S_\Psi = \{z \in \mathbb{C} : Re(z) < 0\}$ die linke komplexe Halbebene. Für $\lambda < 0$ bedeutet dies, dass die numerische Lösung unabhängig von der Schrittweite h konvergiert.

Vorgehen (Allgemeine Berechnung)
Bei Runge-Kutta-Verfahren: $S(z)$ mit Formel, sonst:
- Schreibe die Definition des Verfahrens: $y_{k+1} =$ F(Verfahren, y_k) auf.
- Setze für f(y) überall λy ein.
- Forme so lange um, bis man die Form $y_{k+1} = S(\lambda h)y_k = S(z)y_k$ erreicht hat.

Berechnung von S_Ψ direkt mit Definition: $|S(z)| < 1$.

8.3 ROW (Rosenbrock-Wanner) Verfahren

Geg: $\dot{\vec{y}} = \vec{f}(\vec{y})$, $\vec{y}(t_0) = \vec{y}_0$
Ges: Lösung $\vec{y}(t)$ des autonomen AWP 1.Ordnung

Speziell für steife Differentialgleichungen entwickelte Verfahren basierend auf diagonalimpliziten Runge-Kutta-Verfahren. Anstatt die impliziten Gleichungen ganz zu lösen wird nur ein Newton-Schritt durchgeführt, wodurch nur noch lineare Gleichungen vorkommen.

ROW2 (ROW-Verfahren der Ordnung 2)

In jedem Schritt: Berechne k_1, k_2, welche durch diese lineare Gleichungen definiert sind:

$$\begin{aligned} (\mathbb{I} - ahJ)\vec{k}_1 &= \vec{f}(\vec{y}_i) \\ (\mathbb{I} - ahJ)\vec{k}_2 &= \vec{f}(\vec{y}_i + \tfrac{h}{2}\vec{k}_1) - ahJ\vec{k}_1 \\ \vec{y}_{i+1} &= \vec{y}_i + h\vec{k}_2 \end{aligned}$$

mit a $= \frac{1}{2+\sqrt{2}}$ und $J = Df(\vec{y}_i)$ die Jacobi-Matrix von f an der letzten Stelle.

ROW3 (ROW-Verfahren der Ordnung 3)

Analog mit drei linearen Gleichungssystemen:

$$\begin{aligned}(\mathbb{I}-ahJ)\vec{k}_1 &= \vec{f}(\vec{y}_i)\\ (\mathbb{I}-ahJ)\vec{k}_2 &= \vec{f}(\vec{y}_i+\tfrac{h}{2}\vec{k}_1)-ahJ\vec{k}_1\\ (\mathbb{I}-ahJ)\vec{k}_3 &= \vec{f}(\vec{y}_i+h\vec{k}_2)-d_{31}hJ\vec{k}_1-d_{32}hJ\vec{k}_2\\ \vec{y}_{i+1} &= \vec{y}_i+\tfrac{h}{6}(\vec{k}_1+4\vec{k}_2+\vec{k}_3)\end{aligned}$$

mit a = $\frac{1}{2+\sqrt{2}}$, $d_{31}=-\frac{4+\sqrt{2}}{2+\sqrt{2}}$, $d_{32}=\frac{6+\sqrt{2}}{2+\sqrt{2}}$ und $J=Df(\vec{y}_i)$

ode23s

Adaptives Verfahren basierend auf den ROW2 und ROW3-Verfahren. Eine gute Alternative zu ode45 für steife Differentialgleichungen.

8.4 Lineare Anfangswertprobleme

Geg: $\dot{\vec{y}}=A\vec{y}$, $\vec{y}(t_0)=\vec{y}_0$
Ges: Lösung des autonomen, homogenen und linearen AWP

$\Rightarrow \vec{y}(t)=e^{At}\vec{y}_0=\exp(At)\vec{y}_0$. Matrixexponentiale berechnet man in Python mit expm:

```
yt = dot(scipy.linalg.expm(A*t),y0) # y zur Zeit t
y = lambda t: dot(scipy.linalg.expm(A*t),y0) # y(t) als Funktion von t
```

Für grössere Matrizen $A\in\mathbb{R}^{d\times d}$: Krylovverfahren mit A und dem Startvektor $\vec{y}_0$ (siehe Abschnit 4.1). Dann: $V_m^H A V_m = H_m$ mit $H_m\in\mathbb{R}^{m\times m}, V_m\in\mathbb{R}^{d\times m}$ und $m\ll d$.

Approximierte Lösung: $\vec{u}_m(t)=\| \vec{y}_0 \| V_m \exp(H_m t)\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}$, mit $\begin{pmatrix}1\\0\\\vdots\\0\end{pmatrix}\in\mathbb{R}^m$ (Länge m)

8.5 Exponentielles Rosenbrock-Euler-Verfahren

Geg: $\dot{\vec{y}}=\vec{f}(\vec{y})$, $\vec{y}(t_0)=\vec{y}_0$
Ges: Lösung $\vec{y}(t)$ des autonomen AWP 1.Ordnung

Idee: Linearisierung durch Taylor → Matrixexponential für den Teil mit Jacobi-Matrix
Sei $\varphi(z):=(e^z-\mathbb{I})z^{-1}$ für Matrizen z. Dann:

$$\begin{aligned}\vec{y}_{k+1} &:= \vec{y}_k+h\varphi(hJ(\vec{y}_k))\vec{f}(\vec{y}_k)\\ &= \vec{y}_k+(e^{hJ(\vec{y}_k)}-\mathbb{I})J(\vec{y}_k)^{-1}\vec{f}(\vec{y}_k)\end{aligned}$$

Vorgehen (Exponentielles Rosenbrock-Euler-Verfahren)
- $\vec{x}$ = Lösung des LGS: $J(\vec{y}_k)\vec{x}=\vec{f}(\vec{y}_k)$
- $\vec{y}_{k+1}=\vec{y}_k$ + (exp(h*J($\vec{y}_k$)) - $\mathbb{I}$)$\vec{x}$

Stabilitätsfunktion und Stabilitätsgebiet:
$y_{k+1}=y_k+h\varphi(hJ(y_k))f(y_k)=y_k+h\varphi(h\lambda)\lambda y_k=y_k+h\frac{e^{h\lambda}-1}{h\lambda}\lambda y_k=e^{h\lambda}y_k=:e^z y_k$
Also: $S(z)=e^z$ und $S_\Psi=\{z\in\mathbb{C}:|e^z|<1\}\subset\mathbb{C}$
Mit $z=x+iy$: $|e^z|=|e^{x+iy}|=|e^x|\overset{!}{<}1\Leftrightarrow x<0\Rightarrow S_\Psi=\{z\in\mathbb{C}:Re(z)<0\}$ die linke komplexe Halbebene, wie bei iM. Dieses Verfahren ist also auch sehr gut für steife DGL.

8.6 Zusammenfassung

Verfahren	Ordnung des Verfahrens	für DGL der Ordnung	nur autonome DGL?	braucht Ableitung?	Bemerkungen
eE	$O(h)$	1	nein	nein	
iE	$O(h)$	1	nein	nein	
iM	$O(h^2)$	1	nein	nein	
eM	$O(h^2)$	1	nein	nein	
eTR	$O(h^2)$	1	nein	nein	
RK	-	1	nein	nein	hängt ab von A, b, c
SV	$O(h^2)$	2	nein	nein	keine $\dot{\vec{y}}$ Abh.
LieTr	$O(h)$	1	ja	nein	
Strang	$O(h^2)$	1	ja	nein	
Split	-	1	ja	nein	hängt ab von a_i, b_i
ROW2	$O(h^2)$	1	ja	ja	
ROW3	$O(h^3)$	1	ja	ja	
ExpEul	$O(h^2)$	1	ja	ja	

Bem: Bei iM und den SV Verfahren ist die Energie erhalten.
Viele Professionelle Codes (z.B. ode45, ode23s) verwenden eine adaptive Kombination dieser Verfahren.