# 1 Quadratur

**Quadratur**
Das Integral wird durch eine gewichtete Summe von der Funktion $f$ an verschiedenen Stellen $c_i^n$ angenommenen Werte approximiert:

$$\int_a^b f(x)dx \approx Q_n(f;a,b) := \sum_i^n \omega_i^n f(c_i^n)$$

Hierbei sind die $\omega_i^n$ die **Gewichte** und die $c_i^n \in [a,b]$ die **Knoten** der Quadraturformel.

**Fehler**
Der Fehler einer Quadraturformel $Q_n(f)$ ist

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

**Definition**
Eine Quadraturformel besitzt **Ordnung** $n+1$, wenn sie Polynome vom Grad $n$ exakt integriert.

**Mittelpunkt-Regel**

$$Q^M(f;a,b) = (b-a) f\left(\frac{a+b}{2}\right)$$

Sie besitzt Ordnung 2.

**Trapezregel**
Quadraturformel der Ordnung 2.

$$Q^T(f;a,b) = \frac{b-a}{2}(f(a)+f(b))$$

Für den Fehler gilt:

$$E(n) = \left| -\frac{1}{12}(b-a)^3 f^{(2)}(\xi) \right|$$

mit einem $\xi \in [a,b]$.

**Simpson-Regel**
Quadraturformel der Ordnung 4. Bedarf drei Stützstellen:

$$Q^s(f;a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$

Der Fehler ist:

$$E(n) = \left| -\frac{1}{90}\left(\frac{b-a}{2}\right)^5 f^{(4)}(\xi) \right|$$

mit einem $\xi \in [a,b]$

**Summierte Mittelpunkt-Regel**

$$I(f;a,b) \approx \sum_{i=0}^{N-1} h f\left(\frac{x_i + x_{i+1}}{2}\right)$$

Mit $h = \frac{b-a}{2}$, $x_0 = a$, $x_i = x_0 + ih$, $x_N = b$.

**Summierte Trapezregel**

$$I(f;a,b) \approx \sum_{i=0}^{N-1} \frac{h}{2}(f(x_i) + f(x_{i+1}))$$
$$= \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right)$$

Mit $h = \frac{b-a}{2}$, $x_0 = a$, $x_i = x_0 + ih$, $x_N = b$.

**Summierte Simpson-Regel**

$$I(f;a,b) \approx \sum_{i=0}^{N-1} \frac{h}{6}\left(f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1})\right)$$
$$= \frac{h}{6}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + 4\sum_{i=1}^{N} f\left(\frac{x_{i-1} + x_i}{2}\right) + f(b)\right)$$

Mit $h = \frac{b-a}{2}$, $x_0 = a$, $x_i = x_0 + ih$, $x_N = b$.

**Gauss-Legendre Quadratur**

- $n = 2$:

$$\int_{-1}^{1} f(x)dx \approx Q_2^G = 2 \cdot \left(\frac{1}{2} f\left(\frac{-1}{\sqrt{3}}\right) + \frac{1}{2} f\left(\frac{1}{\sqrt{3}}\right)\right)$$

- $n = 3$:

$$\int_{-1}^{1} f(x)dx \approx Q_3^G = 2 \cdot \left(\frac{5}{18} f\left(\frac{-\sqrt{15}}{5}\right) + \frac{8}{18} f(0) + \frac{5}{18} f\left(\frac{\sqrt{15}}{5}\right)\right)$$

- Allgemeine $n$:

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

Die Stützstellen $c_i$ sind die EW der Matrix:

$$\begin{pmatrix} 0 & b_1 & & & \\ b_1 & 0 & b_2 & & \\ & b_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & b_{n-1} & 0 \end{pmatrix} \quad \text{mit} \quad b_j = \frac{j}{\sqrt{4j^2-1}}$$

```python
from numpy import arange, diag, sqrt
from numpy.linalg import eigh

def gaussquad(n):
    r"""
        Compute nodes and weights for Gauss-Legendre quadrature.

        n:  Number of node-weight pairs
        """
    i = arange(n)   #i = array([0,1,...,n-1])
    b = (i+1) / sqrt(4*(i+1)**2 - 1)
    # now we generate the matrix; it is symmetric
    J = diag(b, -1) + diag(b, 1)
    # in order to find the eigenvalues we can use eigh since J is symmetric
    x, ev = eigh(J)
    # finally, we apply the formula for the weights
    w = 2 * ev[0,:]**2
    return x, w
```

Skript Seite 16

**Clenshaw-Curtis Formel**
Wir substituieren:

$$\int_{-1}^{1} f(x)dx = \int_0^{\pi} f(\cos(\theta))\sin(\theta)d\theta = \sum_{\text{k gerade}} \frac{2a_k}{1-k^2}$$

```python
def cc1(func, a, b, N):
    """
        Clenshaw-Curtis quadrature rule
        by constructing the points and the weights
        """
    bma = b-a
    c = np.zeros((2,2*(N-1) ))
    c[0,0] = 2.0
    c[1,1] = 1.0
    c[1,-1]= 1.0
    for i in np.arange(2,N,2):
        val = 2.0/(1-i**2)  #compare with C.Curtis formula
        c[0,i] = val
        c[0,2*(N-1)-i] = val

    f = np.real(np.fft.ifft(c))
    w = f[0,:N]; w[0] *= 0.5; w[-1] *= 0.5 # weights
    x = 0.5*((b+a)+(N-1)*bma*f[1,:N]) # points
    return np.dot(w,func(x))*bma
```

Skript Seite 18

```
def cc2(func,a,b,N):
    """
        Clenshaw-Curtis quadrature rule
        by FFT with the function values
        """
    # first:  change of variable with adaptation of the interval
    bma = 0.5*(b-a)
    x = np.cos(np.pi * np.linspace(0,N,N+1)/N) # Chebyshev-abszissa
    x *= bma
    x += 0.5*(a+b)
    fx = func(x) # F(theta) = f(cos(theta)) = f(x)
    vx = np.hstack((fx,fx[-2:0:-1])) # extend by reflection to 2π-periodic
    # note:  hstack generates a new array consistig of two former arrays put
together.
    # the index [-2:0:-1] allows to take all elements of an array, except the
first one, in reverse order
    # now we perform the FFT to find the coefficients.
    g = np.real(np.fft.fft(vx))/(2.0*N) # we have to divide by the length, which
is 2N after reflexion
    A = np.zeros(N+1) #A is an array with the coeffs of cosines
    A[0] = g[0]; A[N] = g[N]
    A[1:N] = g[1:N] + np.flipud(g[N+1:]) # put together terms to get coeffs of
cosines
    w = 0.*x
    w[::2] = 2./(1.-np.r_[:N+1:2]**2) #directly apply the C.Curtis quadrature
formula
    # now we can return the final value of the integral; the dot product is a
clever way of writing; otherwise we could have opted for a for-cycle
    return np.dot(w,A)*bma
```

Skript Seite 17

## Adaptive Quadratur

```
from numpy import *
from scipy import integrate

def adaptquad(f,M,rtol,abstol):
    """
    adaptive quadrature using trapezoid and simpson rules
    Arguments:
    f       handle to function f
    M       initial mesh
    rtol    relative tolerance for termination
    abstol  absolute tolerance for termination, necessary in case the exact
integral value = 0, which renders a relative tolerance meaningless.
    """
    h = diff(M)                             #  compute lengths of mesh intervals
    mp = 0.5*( M[:-1]+M[1:] )               #  compute midpoint positions
    fx = f(M); fm = f(mp)                   #  evaluate function at positions and
midpoints
    trp_loc = h*( fx[:-1]+2*fm+fx[1:] )/4   #  local trapezoid rule
    simp_loc= h*( fx[:-1]+4*fm+fx[1:] )/6   #  local simpson rule
    I = sum(simp_loc)                       #  use simpson rule value as
intermediate approximation for integral value
    est_loc = abs(simp_loc - trp_loc)       #  difference of values obtained from
local composite trapezoidal rule and local simpson rule is used as an estimate
for the local quadrature error.
    err_tot  = sum(est_loc)                 #  estimate for global error (sum
moduli of local error contributions)
    #  if estimated total error not below relative or absolute threshold, refine
mesh
    if err_tot > rtol*abs(I) and err_tot > abstol:
        refcells = nonzero( est_loc > 0.9*sum(est_loc)/size(est_loc) )[0]
        I = adaptquad(f,sort(append(M,mp[refcells])),rtol,abstol)   #  add
midpoints of intervalls with large error contributions, recurse.
    return I

if __name__ == '__main__':
    f = lambda x: exp(6*sin(2*pi*x))
    #f = lambda x:  1.0/(1e-4+x*x)
    M = arange(11.)/10 # 0, 0.1, ...  0.9, 1
    rtol = 1e-6; abstol = 1e-10
    I = adaptquad(f,M,rtol,abstol)
    exact,e = integrate.quad(f,M[0],M[-1])
    print('adaptquad:',I, '"exact":',exact)
    print('error:',abs(I-exact))
```

Skript Seite 22

## Mehrdimensionale Integration

Mit Satz von Fubini: eins nach dem anderen integrieren:

$$\int_{a_1}^{b_1} \dots \int_{a_d}^{b_d} f(x_1,\dots,x_d)dx_1\dots dx_d$$
$$\approx \sum_{i_1=0}^{n_1} \dots \sum_{i_d=1}^{n_d} w_{i_1}^1 \dots w_{i_d}^{n_d} f(c_{i_1}^1,\dots,c_{i_d}^d)$$

## Python Code:

- Eindimensionale Integration: scipy.integrate.quad(f,a,b)[0]
- Mehrdimensionale Integration: scipy.integrate.nquad(f,array([[a1,b1],...,[ad,bd]]))[0]

## Drei Term Rekursion der Legendre Polynome:

$$P_{k+1}(x) = \left(x - \frac{\langle x\cdot P_k, P_k\rangle}{\langle P_k, P_k\rangle}\right)\cdot P_k(x) - \left(\frac{\langle P_k, P_k\rangle}{\langle P_{k-1}, P_{k-1}\rangle}\right)P_{k-1}(x)$$

Mit $P_0(x) = 1$ und $P_{-1}(x) = 0$

## Definition

Wir definieren die **Lagrange-Polynome** für die Stützstellen $x_1,\dots,x_n$ als

$$l_i(x) = \prod_{\substack{j=0\\ i\neq j}} \frac{x - x_j}{x_i - x_j}$$

Für diese gelten:

1. $l_i(x_j) = \delta_{ij}$
2. grad $l_i = n$
3. $\sum_{i=0}^n l_i(x) = 1 \ \forall x \in \mathbb{R}$
4. $\sum_{i=0}^n l_i^{(m)}(x) = 0$ für $m \geq 1$
5. $l_1,\dots,l_n$ bilden eine Basis im Raum der Polynome von Grad $\leq n$.

## Drei Term Rekursion der Lagrange Polynome

$$xP_{k-1}(x) = \frac{c_k}{a_k}P_{k-2}(x) - \frac{b_k}{a_k}P_{k-1}(x) + \frac{1}{a_k}P_k(x)$$

Dies können wir in Matrixform bringen:

$$A = \begin{pmatrix} -\frac{b_1}{a_1} & \frac{1}{a_1} & & & 0 \\ \frac{c_2}{a_2} & -\frac{b_2}{a_2} & \frac{1}{a_2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{c_{n-1}}{a_{n-1}} & \frac{b_{n-1}}{a_{n-1}} & \frac{1}{a_{n-1}} \\ 0 & & & \frac{c_n}{a_n} & \frac{b_n}{a_n} \end{pmatrix}$$

Dies ist die gleiche Matrix wie bei der Gauss-Legendre Quadratur. Sei $P_A$ das charakteristische Polynom von $A$. Dann sind die NST von $P_A$ die Knotenpunkte der Gauss-Quadraturformel und die zu den jeweiligen Knotenpunkten gehörigen Gewichte sind definiert als:

$$w_j = \frac{1}{||v_j||}$$

mit $v_j$ dem Eigenvektor zum Eigenwert $c_j$. Da diese aber nicht eindeutig sind, muss normiert werden. Wir finden mit folgendem Verfahren die richtigen Eigenvektoren:

1. Schreibe $u_j = c\cdot v_j = c\cdot[P_0(t_j),\dots,P_{n-1}(t_j)]^T$ für alle Eigenvektoren.
2. Es muss gelten: $1 = \langle P_0, P_0\rangle = \int_{-1}^1 P_0(t_1)P_0(t_1)dx \implies P_0(t_1) = \frac{1}{\sqrt{2}}$
3. Wir betrachten den ersten Eintrag vom Eigenvektor: $(v_j)_1 = c\cdot P_0(t_1) = c\cdot\frac{1}{\sqrt{2}} \implies c = \sqrt{2}(v_j)_1$
4. Man normiere alle Eigenvektoren: $v_j = \frac{1}{c}u_j = \frac{1}{\sqrt{2}(v_j)_1}v_j$
5. Zum Schluss: $w_j = \frac{2(v_j)_1^2}{||v_j||^2}$

# 2 Gewöhnliche DGL

**Definition**
Eine gewöhnliche DGL heisst **autonom**, wenn $f$ unabhängig von $t$ ist.

**Umwandlung in DGL erster Ordnung**
Gegeben: $\vec{y}_n = \vec{f}\left(t, \vec{y}, \ldots, \vec{y}^{(n-1)}\right)$
Vorgehen:

1. Schreibe einen Vektor mit Einträgen $\vec{y}(t), \ldots, \vec{y}^{(n-1)}(t)$:

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

2. Leite diesen Vektor ab und setze die DGL ein:

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

**Autonomisierung einer DGL**
Hat man eine DGL erster Ordnung $\dot{\vec{y}} = \vec{f}(t, \vec{y})$ mit $\vec{y} \in \mathbb{R}^n$, so kann man di Zeitabhängigkeit in die DGL einpacken. Das Vorgehen ist genau analog zur Umwandlung in eine DGL erster Ordnung. Dazu wird $\vec{z} \in \mathbb{R}^{n+1}$ eingeführt:

$$\vec{z}(t) := \begin{pmatrix} \vec{y}(t) \\ t \end{pmatrix} = \begin{pmatrix} \vec{z} \\ z_{n+1} \end{pmatrix}$$

Somit erhalten wir die autonome DGL:

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

**Konvergenzordnung**
Sei $p > 0$ die Konvergenzordnung des Verfahrens sodass $\forall N \in \mathbb{N} : E(N) \leq \frac{c}{N^p}$ bzw $E(h) \leq ch^p$ mit $h := \frac{t_{\text{end}} - t_0}{N}$ 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 $N$: $e(N) := \|\vec{y}_{\text{exact}} - \vec{y}_N\|$
- Berechne:
$p = -\text{polyfit}(\log(N), \log(e), 1)[0] = \text{polyfit}(\log(h), \log(e), 1)[0]$

**Polygonzugverfahren**
Gegeben: $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, $\vec{y}(t_0) = \vec{y}_0$
Gesucht: Lösung $y(t)$ des AWP 1. Ordnung

**Explizites Eulerverfahren**
Approximation durch Tangende zum Anfangszeitpunkt $t_k$.
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(t_k, \vec{y}_k)$
Lokaler Fehler: $O(h^2)$

```
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,:])
```

**Implizites Eulerverfahren**
Approximation durch Tangente am nächsten Zeitpunkt $t_{k+1}$.
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(t_{k+1}, \vec{y}_{k+1})$
$\Rightarrow \vec{y}_{k+1}$ NST 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.
Lokaler Fehler: $O(h^2)$

```
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,:])
```

**Implizite Mittelpunktregel**
Approximation durch Tangente zum Zeitpunkt $(t_k + t_{k+1})/2$
$\vec{y}_{k+1} := \vec{y}_k + h_k \vec{f}(\frac{1}{2}(t_k + t_{k+1}), \frac{1}{2}(\vec{y}_k + \vec{y}_{k+1}))$
$\Rightarrow \vec{y}_{k+1}$ NST 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

Lokaler Fehler: $O(h^3)$

```
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,:]))
```

**Implizite Trapezregel**
$\vec{y}_{k+1} = \vec{y}_k + \frac{h}{2}\left(\vec{f}(t_k, \vec{y}_k) + \vec{f}(t_{k+1}, \vec{y}_{k+1})\right)$
$\Rightarrow y_{k+1}$ NST von: $F(\vec{X}) = \vec{X} - \vec{y}_k - \frac{h}{2}\left(\vec{f}(t_k, \vec{y}_k) + \vec{f}(t_{k+1}, \vec{X})\right)$
Guter Startwert für Nullstellensuche: Schritt aus eine
Lokaler Fehler: $O(h^3)$

**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

**Zwei-Schritt-Verfahren**
Idee: Approximation durch eine Parabel und äquidistante $t_k$
Idee: $f(t_k, y_k) = \ddot{y}_k \approx \frac{\dot{y}_{k+1} - \dot{y}_k}{h} \approx \frac{\frac{y_{k+1} - y_k}{h} - \frac{y_k - y_{k-1}}{h}}{h} \approx \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2}$
$\vec{y}_{k+1} := -y_{k-1} + 2\vec{y}_k + h^2 \vec{f}(t_k, \vec{y}_k)$
mit zweitem Startwert: $\vec{y}_1 = \vec{y}_0 + h\vec{v}_0 + \frac{h^2}{2}\vec{f}(t_0, \vec{y}_0)$

```
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,:])
```

**Ein-Schritt-Verfahren**
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 eE für $\dot{\vec{y}}(t) = \vec{v}(t)$ und $\dot{\vec{v}}(t) = \vec{f}(t, \vec{y})$
Es gibt 2 Methoden: Leap-Frog-Methode und Velocity-Verlet

**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**

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

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

```
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 Vel.Verl. die Geschw. bei $t_k$.

Die Einschritt-Verfahren sind numerisch stabiler.
In allen Störmer-Verlet-Verfahren ist die Energie erhalten.
Code im Skript auf Seite 80

**Konvergenzordnung**
Wie bei Quadratur.
Exacte Lösung: ode45
from ODE45 import ODE45
$t, y = \text{ode45}(f, (t_0, t_{\text{end}}), y_0)$

**Fehler**

- Lokaler Fehler: $\|y(t_{n+1}) - y_{n+1}\|$ abschätzen durch fixe Konstanten.
- Fehlerfortpflanzung: $\|y^\star_{n+1} - y_{n+1}\|$ abschätzen mit $y(t_n)$ und $y_n$. Methode: Approximiere $y^\star_{n+1}$ mit $y_n$ und $y_{n+1}$ mit $y(t_n)$
- Fehlerakkumulierung: Über die Fehlerfortpflanzung bis $n$ summieren. Die Potenz nicht vergessen. Tipp: Geometrische Reihe.

**Theorem**
$\|\vec{y}_n - \vec{y}(t_n)\| \le M \cdot h$ für alle $n$, wobei
$M = \frac{1}{L}\left(e^{L(T-t_0)} - 1\right)\frac{1}{2}\max\left\|\ddot{\vec{y}}(t)\right\|$ für $t \in [t_0, T]$.
Kann man mittels den vorherigen drei Fehlertypen beweisen.

# 3 Strukturerhaltung

**Definition**
Eine Funktion $I : D \to \mathbb{R}$ heisst **ersts Integral/Invariante** der DGL, wenn $I(y(t)) \equiv$ const. für jede Lösung $y = y(t)$ der DGL. Eine notwendige und hinreichende Bedingung für differenzierbares erstes Integral der DGL ist

$$\text{grad}I(y)f(t,y) = 0 \qquad \text{für alle } (t,y) \in [t_0, T] \times D$$

**Definition**
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 DGL:

$$\begin{cases} \dot{q}_j = \frac{\partial H}{\partial p_j}(q,p) \\ \dot{p}_j(t) = -\frac{\partial H}{\partial q_j}(q,p) \end{cases} \qquad \text{für } j = 1, 2, \ldots, d$$

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 DGL bleiben gleich, doch dann ist $H$ nicht mehr erhalten.

**Splitting-Verfahren**
Geg.: AWP 1. Ordnung, autonom: $\dot{\vec{y}} = f(y)$ und $\vec{y}(t_0) = y_0$ das nur schwer oder gar nicht analytisch lösbar ist.
Wenn DGL nicht autonom ist, muss man autonomisieren.

**Evolutionsoperatoren**
$\Phi^{t_0,t} : D \subset \mathbb{R}^n \to D$ 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 $\dot{\vec{y}} = \vec{f}(t, \vec{y})$, $\vec{x}(t_0) = \vec{y}_0$ ist.
Für autonome DGL gilt: $\Phi^{t_0,t} = \Phi^{0,t-t_0} := \Phi^{t-t_0} := \Phi^h$
Numerische Verfahren liefern den diskreten Evolutionsoperator $\Psi^h \approx \Phi^h$

Geg.: Autonome, separierte DGL $\dot{\vec{y}} = \vec{f}_a(\vec{y}) + \vec{f}_b(\vec{y})$, $\vec{y}(t_0) = \vec{y}_0$
Mit bekannten Evolutionsoperatoren $\Phi^h_a$ zu $\dot{\vec{y}} = \vec{f}_a(\vec{y})$ und $\Phi^h_b$ 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**: $\Psi^h_1 = \Phi^h_a \circ \Phi^h_b$ oder $\Psi^h_1 = \Phi^h_b \circ \Phi^b_a$
**Strange-Splitting**: $\Phi^h_2 = \Phi^{h/2}_a \circ \Phi^h_b \circ \Phi^{h/2}_a$
**Allgemein**: $\Psi^h_s = \prod_{i=1}^s \Phi^{b_i h}_b \cdot \Phi^{a_i h}_a$ mit $\sum_{i=1}^s b_i = \sum_{i=1}^s a_i = 1$

**Bemerkung**
Das Splittingverfahren kann als Verallgemeinerung des Störmer-Verlet Verfahren betrachtet werden. Sind $\Phi_a$ und $\Phi_b$ nicht bekannt, können diskrete Evolutionsoperatoren $\Psi_a$ und $\Phi_b$ verwendet werde.

**Processing**

$$\hat{\Psi} = \Pi^h \circ \Psi^h \circ \left(\Pi^h\right)^{-1}$$

Dabei ist $\Pi^h$ der **post-processor** und $\left(\Pi^h\right)^{-1}$ der **pre-processor**. Der Vorteil dieser Schreibweise:

$$\left(\hat{\Psi}^h\right)^n = \Pi^h \circ \left(\Psi^h\right)^n \circ \left(\Pi^h\right)^{-1}$$

Vorteile falls: $\hat{\Psi}^h$ genauer als $\Psi^h$, $\Pi^h$, $\left(\Pi^h\right)^{-1}$ günstig, keine/wenige Ausgaben der Lösung vor Endschritt gewünscht.

# 4 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 DGLproblem in ein Integrationsproblem um: $\vec{y}(t_1) = \vec{y}(t_0) + \int_{t_0}^{t_1} \vec{f}(t, \vec{t})dt$ und verwende eine Quadraturformel zur Approximation des Integrals:
$\vec{y}(t_1) \approx \vec{y}_0 + h\sum_{i=1}^s b_i \vec{f}(t_0 + c_i h, \vec{y}(t_0 + c_i h))$
mit Gewichten $b_i$ und Stützstellen $c_i \in [0,1]$ und $h = t_1 - t_0$

**Explizite Mittelpunktsregel**
Wir wählen die Mittelpunktsregel als QF. Der noch unbekannte Funktionswert in der Mitte $y(t_0+\frac{h}{2})$ 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 RK 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.

```
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)
```

$\vec{k}_1 := \vec{f}(t_0, \vec{y}(t_0))$ und $\vec{k}_2 := \vec{f}(t_0 + h, \vec{y}(t_0) + h\vec{f}(t_0, \vec{y}_0))$, dann ist $\vec{y}_1 = \vec{y}_0 + \frac{h}{2}(\vec{k}_1 + \vec{k}_2)$

**Polygonzugverfahren**: Die drei Polygonzugverfahren sind jeweils $s = 1$-stufige RK Verfahren.

**s-stufiges RK-Verfahren**
Gegeben sind $b_i$ und $a_{ij}$ in $\mathbb{R}$ mit $i, j = 1, \dots, s$ und $\sum_{i=1}^{s} b_i = 1$. Dann ist $c_i := \sum_{j=1}^{s} a_{ij}$ und definiere:

**Vorgehen**: (Allgemeines RK-Verfahren)
Berechne: $\vec{k}_i := \vec{f}(t_0 + c_i h, \vec{y}_0 + h\sum_{j=1}^{s} a_{ij}\vec{k}_j)$ mit $i = 1, \dots, s$
Daraus: $\vec{y}_{j+1} = \vec{y}_j + h\sum : i = 1^s b_i \vec{k}_i$

**Definition**
Das **Butcher-Tableau** zur Darstellung eines RK-Verfahrens:

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

Ein RK-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 inpliziten Gleichungen gelöst werden.
- **diagonal implizit**, falls $A$ eine nicht-strikte untere Dreiecksmatrix ist. Es kann nacheinander eine Gleichung für das neue $k_i$ aufgestellt werden.
- **implizit**, sonst. Es muss eine grosse Gleichung mit allen $k_i$'s als Unbekannten auf eine Schlange gelöst werden. Dies benötigt den grössten und kompliziertesten Rechenaufwand.

**Beispiele**:

eE: $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ iE: $\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$ iM: $\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$

eT: $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ eM: $\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$

**Gauss-Kollokation**
Motivation: Nutze Analog zu Gauss-Legendre QF die maximale Ordnung eines $s$-stufigen RK-Verfahrens aus:

- Explizites Verfahren: Ordnung $p \le s$
- Implizites Verfahren: Ordnung $p \le 2s$

Idee: Approximiere $y(t)$ zwischen $t_0$ und $t_0 + h$ durch ein Polynom von Grad $s$ (= Kollokationspolynom $u(t)$), welches die DGL an den Punkten $t_0 + c_i h \in [t_0, t_0 + h]$ erfüllt:

$$\begin{cases} u(t_0) = y_0 \\ \dot{u}(t + c_i h) = f(t_0 + c_i h, u(t_0 + c_i h)) \quad \forall i \in \{1, \dots, s\} \end{cases}$$

Lösen mit: RK-Verfahren mit $a_{ij} := \int_0^{c_i} l_j(t)dt$, $b_i := \int_0^1 l_i(t)dt$ mit $l_i(t)$ den Lagrange-Polynomen $\forall i \in \{1, \dots, n\}$ und $c_i$ den Knoten der Gauss-Legendre-QF (hier: Knoten $\hat{=}$ NST des $s$-ten verschobenen Legendre Polynomes $P_s(x) := \frac{ds}{dx^s}(x^s(x-1)^s)$)

Konkret: Kollokationsverfahren sind implizite RK-Verfahren mit speziellen Einträgen im Butcher-Tableau.

**Beispiel** (Gauss-Koll. zu $s = 2$)

$$\begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{2} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Knoten $c_{1,2} = \frac{1}{2} \pm \frac{\sqrt{3}}{6}$

**Adaptivität/ Partitioniertes RK**
Bisher: Schrittweite $h$= konst.
Idee: Verkleinere $h$, wo $f$ schwierig, vergrössere $h$ wo $f$ einfach.
Vorgehen: Verwende zwei RK-Verfahren unterschiedlicher Ordnung (Code: $\Psi_{\text{high}}$, $\Psi_{\text{low}}$) und berechne den lokalen Fehler. $\Rightarrow$

- lok. Fehler gross: $h_{\text{neu}} := \frac{h}{2}$
- lok. Fehler klein: $h_{\text{neu}} := c \cdot h$ $(c > 1)$ zB. 1.1

Wichtig!: rhs mus formell Zeitabhängig sein!

# 5 Steife DGL

**Definition**
Eine DGL ist steif, falls shc das Verhalten der numerischen Lösungen eines expliziten Verfahrens ab einem bestimmten $N$ bzw. $h$ komplett verändert.

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

- **Analytisch**: Direkt die Definition eines expliziten Verfahrens (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. (zB. 10 bis $10^6$)

**Stabilitätsbegriffe**

**Testgleichung**
Die eindimensionale DGL $\dot{y} = \lambda y =: f(y)$ mit $\lambda \in \mathbb{R}$ oder $\lambda \in \mathbb{C}$ 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 $z := \lambda \cdot h$.

**Stabilitätsgebiet**
$S_\Psi := \{z \in D \mid |S(z)| < 1\} \subset \mathbb{C}$

**Bemerkung**
Für ein $s$-stufiges RK-Ein-Schritt-Verfahren mit Butcher-Tableau $\begin{array}{c|c} \vec{c} & A \\ \hline & (\vec{b})^T \end{array}$ gilt: $S(z) = \frac{\det(E_n - zA + z\vec{1}\cdot(\vec{b})^T)}{\det(E_n - zA)}$ wobei $\vec{1} = (1, \dots, 1)^T \in \mathbb{R}^n$

**Vorgehen** (Allgemeine Berechnung)
Bei RK-Verfahren: $S(z)$ mit Formel, sonst:
- Schreibe Def. des Verfahrens $y_{k+1} = F(\text{Verfahren}, y_k)$ auf
- Setze für $f(y)$ überall $\lambda y$ ein

Forme so lange um, bis man die Formel $y_{k+1} = S(\lambda h)y_k = S(z)y_k$ erreicht hat
Berechnung von $S_\Psi$ direkt mit Definition $|S(z)| < 1$

**Definition**
Ein Verfahren heisst **A-stabil**, falls die (ganze) linke komplexe Ebene im Stabilitätsgebiet des Verfahrens ist. $\mathbb{C}^- \subset S_\Psi$

**Definition**
Ein Verfahren heisst **L-stabil**, falls sie A-stabil ist und $S(-\infty) = \lim_{z \to -\infty} S(z) = 0$

**Radau Verfahren**

**ROW2** (Ordnung 2)
In jedem Schritt: Berechne $k_1, k_2$ welche durch diese lineare Gleichungen definiert sind:

$$(E_n - ahJ)\vec{k}_1 = \vec{f}(\vec{y}_i)$$
$$(E_n - ahJ)\vec{k}_2 = \vec{f}(\vec{y}_i + \frac{h}{2}\vec{k}_1) - ahJ\vec{k}_1$$
$$\vec{y}_{i+1} = \vec{y}_i + h\vec{k}_2$$

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

**ROW3** (Ordnung 3)
Analog mit drei linearen GLS:

$$(E_n - ahJ)\vec{k}_1 = \vec{f}(\vec{y}_i)$$
$$(E_n - ahJ)\vec{k}_2 = \vec{f}(\vec{y}_i + \frac{h}{2}\vec{k}_1) - ahJ\vec{k}_1$$
$$(E_n - ahJ)\vec{k}_3 = \vec{f}(\vec{y}_i + h\vec{k}_2) - d_{31}hJ\vec{k}_1 - d_{32}hJ\vec{k}_2$$
$$\vec{y}_{i+1} = \vec{y}_i + \frac{h}{6}\left(\vec{k}_1 + 4\vec{k}_2 + \vec{k}_3\right)$$

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 $J = Df(\vec{y}_i)$

# 6 Nullstellensuche

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

**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, wie zB.
  - Max. Anzahl Schritte
  - Absolute Toleranz: $\|x_{k+1} - x_k\| <$ abstol
  - Relative Toleranz: $\frac{\|x_{k+1} - x_k\|}{\|x_{k+1}\|} <$ reltol

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

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

**Definition** (Konvergenzordnung $p$)
Sei $c > 0$ sodass $e_{k+1} \le ce_k^p$. Dann folgt die Berechnung:

$$\left.\begin{aligned} e_{k+1} &\approx ce_k^p \\ e_k &\approx ce_{k-1}^p \end{aligned}\right\} \Rightarrow p \approx \frac{\log\left(\frac{e_{k+1}}{e_k}\right)}{\log\left(\frac{e_k}{e_{k-1}}\right)} =: p_k$$

Achtung! $p$ ist ein Array mit einträgen $[p_1, \dots, p_{N-2}]$. Wähle für $p$ einfach den letzten "vernünftigen" Eintrag.

Wichtig! Dies ist nicht die selbe Konvergenzordnung wie bei DGL oder Quadratur.

**Definition**
$x^{(k+1)} = \Phi(x^\star)$ heisst **linear konvergent** nach $x^\star$, falls es ein $L < 1$ gibt, sodass:

$$\left\|x^{(k+1)} - x^\star\right\| \le L\left\|x^{(k)} - x^\star\right\| \quad \forall k \in \mathbb{N}$$

**Bemerkung**

$$\left\|x^{(k+1)} - x^\star\right\| \le L\left\|x^{(k)} - x^\star\right\| \le L^{k+1}\left\|x^{(0)} - x^\star\right\|$$

**Definition**
**Konvergenzordnung** $p$ des Iterativen Verfahrens heisst, es gibt ein $C > 0$, sodass

$$\left\|x^{(k+1)} - x^\star\right\| \le C \cdot \left\|x^{(k)} - x^\star\right\|^p$$

**Bemerkung**
Bei linearer Konvergenz und bekanntem $L$ kann man folgende Abschätzung machen:

$$\left\|x^{(k+1)} - x^\star\right\| \le \frac{1}{1-L}\left\|x^{(k+1)} - x^{(k)}\right\|$$

## Fixpunktiteration

**Vorgehen**
Wähle ein $\phi(x)$. Iteratives Verfahren mit Iterationsvorschrift: $x_{k+1} = \phi(x)$

**Konvergenz**

**Definition**
Eine Funktion $\phi$ heisst **Kontraktion**, falls es ein $L > 1$ gibt, so dass $\|\phi(x) - \phi(y)\| \leq L\|x - y\|$ für alle $x, y$.

**Bemerkung**
Wenn $x^\star$ ein Fixpunkt der Kontraktion $\phi$ ist, dann ist

$$\left\|x^{(k+1)} - x^\star\right\| = \left\|\phi(x^{(k)}) - \phi(x^\star)\right\| \leq L\left\|x^{(k)} - x^\star\right\|$$

Das heisst, dass das iterative Verfahren $x^{k+1} = \phi(x^k)$ mindestens linear konvergiert.

**Satz**
Sei $D \subset \mathbb{K}^n$ ($\mathbb{K} = \mathbb{R}, \mathbb{C}$), mit $D$ abgeschlossen, und $\phi : D \to D$ einer Kontraktion. Dann existiert ein eindeutiger Fixpunkt $x^\star$, also $\phi(x^\star) = x^\star$. Dieser ist der Grenzwert der Folge $x^{(k+1)} = \phi(x^{(k)})$.

**Satz** (Hinreichende Bedingung für lokale lineare Konvergenz einer Fixpunktiteration)
Es Sei $U$ konvex und $\phi : U \subset \mathbb{R}^n \to \mathbb{R}^n$ stetig differenzierbar mit $L := \sup_{x \in U} \|D\phi(x)\| < 1$ wobei ($D\phi(x)$ die Jacobi-Matrix von *phi* ist). Wenn $\phi(x^\star) = x^\star$ für $x^\star \in U$, dann konvergiert die Fixpunktiteration $x^{(k+1)} = \phi(x^{(k)})$ gegen $x^\star$ lokal mindestens linear.

**Satz**
Sei $\phi : \mathbb{R}^n \to \mathbb{R}^n$ mit $\phi(x^\star) = x^\star$ und $\phi$ stetig differenzierbar in $x^\star$. Ist $\|D\phi(x^\star)\| < 1$, dann konvergiert die Fixpunktiteration $x^{(k+1)} = \phi(x^{(k)})$ lokal (mindestens) linear, mit $L = \|D\phi(x^\star)\|$.

**Satz**
Sei $U \subset \mathbb{R}$ ein Intervall und $\phi : U \to \mathbb{R}$ $(m+1)$-mal differenzierbar mit $\phi(x^\star) = x^\star \in U$. Sei weiterhin $\phi^{(l)}(x^\star) = 0$ für $l = 1, \ldots, m$ ($m \geq 1$). Dann konvergiert die Fixpunktiteration $x^{(k+1)} = \phi(x^{(k)})$ gegen $x^\star$ lokal der Ordnung $p \geq m + 1$.

**Lemma**
Konvergiert die Kontraktion $\phi$ linear mit dem Faktor $L < 1$, dann gilt die folgende Abschätzung:

$$\left\|x^\star - x^{(k)}\right\| \leq \frac{L^{k-l}}{1-L}\left\|x^{(l+1)} - x^{(l)}\right\|$$

## Bisektionsverfahren (Intervallhalbierungsverfahren)

Idee: Zwischenwertsatz: $F : [a, b] \to \mathbb{R}$ stetig mit $F(a)F(b) < 0 \Rightarrow \exists x^\star \in [a, b]$ sodass $F(x^\star) = 0$
Vorgehen: Iteratives Verfahren mit der Iterationsvorschrift:
- Intervall halbieren $m = \frac{a+b}{2}$
- Suche Intervall mit unterschiedlichen Vorzeichen: $F(a)F(m) < 0$ oder $F(m)F(b) < 0$
- Weiter mit Intervall "$< 0$"

Konvergiert immer. Lineare Konvergenz. Keine verallgemeinerung in mehreren Dimensionen.

## Newtonverfahren

Idee Approximation von $F$ durch Tangente bei $x_k$ (Taylor):
$F(x) \approx \tilde{F}(x) := F(x_k) + F'(x) \cdot (x - x_k) \stackrel{!}{=} 0$
Vorgehen: (Newtonverfahren)
Iteratives Verfahren mit Iterationsvorschrift: $x_{k+1} = x_k - \frac{F(x_k)}{F'(x_k)}$

Mehrdimensional
Gleiche Idee: $x_{k+1} = x_k - DF(x_k)^{-1}F(x_k) := x_k - s_k$ Aber $DF(x_k) := (\frac{\partial F_i}{\partial x_j})_{ij} \in \mathrm{Mat}(n \times n; \mathbb{R})$ "Berechne nie das Inverse einer Matrix!"
Vorgehen (Mehrdimensionales Newtonverfahren)
Iteratives Verfahren mit Iterationsvorschrift:
- $\vec{s}_k$ = Lösung des linearen GLS $DF(\vec{x}_k)\vec{s}_k = F(\vec{x}_k)$
- $\vec{x}_{k+1} = \vec{x}_k - \vec{s}_k$

Newtonverfahren = Fixpunktiteration mit $\phi(x) = x - \frac{F(x)}{F'(x)} \Rightarrow$ lokal mindestens quadratische Konvergenz.

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. Startwert benötigt. Mehrdimensional nicht möchlich.

## Gedämpftes Newtonverfahren

Idee: Dämfung von $s_k$ mit einem Dämpfungsparameter $\lambda_k \in (0, 1]$
Es wird das maximale $\lambda_k$ gewählt, so dass

$$\left\|\frac{F\left(x^{(k)} - \lambda^{(k)}\frac{F(x^{(k)})}{DF(x^{(k)})}\right)}{DF(x^k)}\right\|_2 \leq \left(1 - \frac{\lambda^{(k)}}{2}\right)\left\|\frac{F(x^{(k)})}{DF(x^{(k)})}\right\|_2$$

Praxis: $\lambda_k = 1 \to \frac{1}{2} \to \frac{1}{4} \to \ldots$ bis die obere 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: $\frac{F(x_k)}{F'(x_k)}$
- $\lambda_k = \max\left\{\frac{1}{2^n} \mid n \in \mathbb{N}_0 \text{ und erfüllt obige Bedingung}\right\}$ (in jedem Iterstionsschritt neu berechnen!)
- $\vec{x}_{k+1} = \vec{x}_k - \lambda_k\vec{s}_k$

## Quasi-Newtonverfahren

Broyden Verfahren
Setze $J_0 = DF(x_0) \in M(n \times n, \mathbb{R})$ und löse für $k = 1, 2, \ldots$

$$\begin{cases} J_k \cdot s_k = F(x_k) \Leftrightarrow s_k = \mathrm{solve}(J_k, F(x_k)) \\ x_{k+1} = x_k - s_k \\ J_{k+1} = J_k + \frac{1}{\|s_k\|^2}F(x_{k+1})(-s_k)^T \end{cases}$$

Sherman-Morrison-Formel

$$J_{k+1}^{-1} = J_k^{-1} + \frac{J_k^{-1}F(x_{k+1})s_k^T J_k^{-1}}{\|s_k\|^2 - s_k^T J_k^{-1}F(x_{k+1})}$$

Daraus entsteht folgende Iteration:

1. Schritt:

$$\begin{cases} J_0 = DF(x_0) \\ s_0 = \text{solve}(J_0, F(x_0)) \\ x_1 = x_0 - s_0 \end{cases}$$

2. Schritt:

$$\begin{cases} J_1^{-1} = J_0^{-1} + \frac{J_0^{-1}F(x_1)s_0^T J_0^{-1}}{\|s_0\|^2 - s_0^T J_0^{-1}F(x_1)} \\ s_1 = J_1^{-1}F(x_1) \\ x_2 = x_1 - s_1 \\ \vdots \end{cases}$$

Alternativ

$$s_{k+1} = J_{k+1}^{-1}F(x_{k+1}) = J_k^{-1}F(x_{k+1}) + \frac{J_k^{-1}F(x_{k+1})s_k^T J_k^{-1}F(x_{k+1})}{\|s_k\|^2 - s_k^T J_k^{-1}F(x_{k+1})}$$

Wir definieren nun $w_k := J_k^{-1}F(k+1)$ (Lsg. des GLS $J_k w_k = F(x_{k+1})$) und $z_k := s_k^T w_k$. Dann folgt:

$$S_{k+1} = \left(1 + \frac{z_k}{\|s_k\|^2 - z_k}\right) w_k$$

# 7 Numerische Lineare Algebra

**Bemerkung**

| Invertierbar | Nicht invertierbar |
|---|---|
| $A$ ist regulär | $A$ ist singulär |
| Zeilen sind linear unabhängig | Zeile sind linear abhängig |
| Spalten sind linear unabhängig | Spalten sind linear abhängig |
| $\det A \neq 0$ | $\det A = 0$ |
| $Ax = 0$ hat eine Lösung $x = 0$ | $Ax = 0$ hat $\infty$ viele Lösungen |
| $Ax = b$ hat eine Lösung $x = A^{-1}b$ | $Ax = b$ hat keine oder $\infty$ Lösungen |
| $A$ hat vollen Rang | $A$ hat Rang $r < n$ |
| $A$ hat $n$-nicht-Null-Pivoten | $A$ hat $r < n$ Pivoten |
| span $\{A_{:,1}, \dots, A_{:,n}\}$ hat dim $n$ | span $\{A_{:,1}, \dots, A_{:,n}\}$ hat dim $r < n$ |
| span $\{A_{1,:}, \dots, A_{n,:}\}$ hat dim $n$ | span $\{A_{1,:}, \dots, A_{n,:}\}$ hat dim $r < n$ |
| Alle Eigenwerte von $A$ sind $\neq 0$ | 0 ist EW von $A$ |
| $0 \notin \sigma(A)$ = Spektrum von $A$ | $0 \in \sigma(A)$ |
| $A^H A$ ist symmetrisch positiv definit | $A^H A$ ist semidefinit |
| $A$ hat $n$ (positive) Singulärwerte | $A$ hat $r < n$ (positive) Singulärwerte |

**Bemerkung**
Orthogonale/Unitäre Transformationen erhalten die euklidische Norm:

$$\|Qx\|_2^2 = (Qx)^H(Q_x) = x^H Q^H Q x = x^H E_n x = \|x\|_2^2$$

**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 GLS
$Ax = b \Leftrightarrow LUx = Pb \Leftrightarrow Lz = Pb$ (Vorwärtssubstitution) und $Ux = z$ (Rückwärtssubstitution)
Code: P,L,U = scipy.linalg.lu(A)

**Cholesky-Zerlegung**

Ist $A$ symetrisch ($a = A^T$) und positiv definit, dann existiert eine Zerlegung $A = LL^T = U^T U$, wobei $L$ und $U$ untere Dreiecksmatrizen sind mit strikt positiven Diagonaleinträgen.

Anwendung: $A = LL^T \Rightarrow Ax = LL^T x = b \Leftrightarrow Ly = b \Rightarrow$ finde $y \Rightarrow L^T x = y \Rightarrow$ finde $x$.

Code: L = numpy.linalg.cholesky(A)

**QR-Zerlegung**

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

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

Code: Qhat, Rhat = numpy.linalg.qr(A) oder
Q,R = numpy.linalg.qr(A,mode='complete')

Methoden:

**Gram-Schmidt-Verfahren**

Orthogonalisiere die Spalten von $A \Rightarrow Q$. Finde danach $R$, sodass gilt: $QR = A$.

QR via **Rotation**:

$$G_{ij}(\varphi) = \begin{bmatrix} 1 & & & & & & & & & & \\ & \dots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & \cos(\varphi) & & & & \sin(\varphi) & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & -\sin(\varphi) & & & & \cos(\varphi) & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{bmatrix}$$

mit $\cos(\varphi)$ an der $ii$-ten und $jj$-ten Stelle. Und $\sin(\varphi)$ an der $ij$-ten Stelle, sowie $-\sin(\varphi)$ an der $ji$-ten Stelle. Dabei gilt:

$$r = \sqrt{x_i^2 + x_j^2} \quad \cos(\varphi) = \frac{x_i}{r} \quad \sin(\varphi) = \frac{x_j}{r}$$

Dann folgt:

$$G_{ij}(\varphi) \cdot \begin{bmatrix} x_i \\ \dots \\ x_{i-1} \\ x_i \\ x_{i+1} \\ \vdots \\ x_{j-1} \\ x_j \\ x_{j+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_i \\ \dots \\ x_{i-1} \\ r \\ x_{i+1} \\ \vdots \\ x_{j-1} \\ 0 \\ x_{j+1} \\ \vdots \\ x_n \end{bmatrix}$$

Mit diesem Verfahren erhält man einen Algorithmus zur Bestimmung von $Q$.

**Housholder-Spiegelung**
Sei $a$ der Startvektor := erste Spalte von $A$. Dann folgt der folgende Algorithmus:

$$v := \begin{cases} \frac{1}{2}(a + \|a\|_2 e_1) & \text{falls } a_1 > 0 \quad \text{(der erste Eintrag von } a\text{)} \\ \frac{1}{2}(a - \|a\|_2 e_1) & \text{falls } a_1 < 0 \end{cases}$$

$$u := \frac{v}{\|v\|_2}$$

$$Q^T := E_m - 2uu^T$$

Dies wiederholt man für alle Spalten von $A$. Dann erhält man: $Q = \left(Q_m^T \dots Q_1^T\right)^T = \left(\prod_{i=0}^{m-1} Q_{m-i}^T\right)^T$
Am Schluss muss man noch $R$ herausfinden: $R = \left(\prod_{i=}^{m} Q_i^T\right) \cdot A$

**Singulärwertzerlegung**
Sei $A \in \mathbb{C}^{m\times m}$ beliebig. Dann gibt es unitäre Matrizen $V \in \mathbb{C}^{m\times m}$ und $U \in \mathbb{C}^{n\times n}$ und die $m \times n$ Diagonalmatrix in $\sigma = \text{diag}(\sigma_1, \dots, \sigma_p)$ mit $p = \min\{m, n\}$ und $\sigma_1 \geq \dots \sigma_p \geq 0$, sodass

$$A = U\Sigma V^H$$

$m = n \Rightarrow \Sigma$ ist invertierbar:
$\Rightarrow Ax = b \Leftrightarrow U\Sigma V^T x = b \Leftrightarrow x = \left(U\Sigma V^T\right)^{-1} b \Leftrightarrow x = V\Sigma^{-1}U^T b$
Code:
U,s,Vt = scipy.linalg.svd(A)
Sigma.inv = np.diag(1/s)
x = np.dot(Vt.T , np.dot(Sigma.inv , np.dot(U.T , y)))

$m \neq n \Rightarrow \text{rang}(\Sigma) = r < p = \min\{m, n\}$: (reduzierte SVD)

1. Zerlege $A = U\Sigma V^T =$

$$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} \Sigma_r & \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix}$$

mit $U_1 \in M(m \times r; K)$, $\Sigma_r \in M(r \times r; K)$, $V_1^T \in M(r \times n; K)$, $A \in M(m \times n; K)$, dann ist $\Sigma_r^{-1}$ wohldefiniert und ist gegeben durch:

$$\Sigma_r^{-1} = \begin{pmatrix} \frac{1}{\sigma_1} & & \\ & \cdots & \\ & & \frac{1}{\sigma_r} \end{pmatrix}$$

2. Schreibe um: $Ax = b \Leftrightarrow U_1\Sigma_r V_1^T x = b \Leftrightarrow x = V_1\Sigma_r^{-1}U_1^T b$. Man nenne $V_1\Sigma_r^{-1}U_1^T$ auch die **Pseudoinverse** von $A$.

**Kondition**

**Definition**
Die **Konditionszahl** einer Matrix $A$ ist $\text{cond}(A) := \left\|A^{-1}\right\| \cdot \|A\|$

**Definition**
Die **Matrixnorm** ist gegeben als:

$$\|A\| := \sup_{\|x\|\neq 0} \frac{\|Ax\|}{\|x\|} = \sup_{\|x\|=1} \|Ax\|$$

**Theorem**
Wenn die Singulärwerte von $A$ erfüllen:

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \Sigma_r > \sigma_{r+1} = \sigma_p = 0$$

dann gilt: $\text{rang}A = r$ und:

$$\text{Ker}A = \text{span}\{v_{r+1}, \dots, v_n\}$$
$$\text{Im}A = \text{span}\{u_1, \dots, u_r\}$$

**Bemerkung**
Gegeben $A \in \mathbb{C}^{m\times n}$ mit $m > n$, finde $x \in \mathbb{C}^n$ mit $\|x\| = 1$, sodass $\|Ax\|_2$ minimal wird. Die SVD hilft, denn unitäre Matrizen erhalten die 2-Norm:

$$\min_{\|x\|=1} \|Ax\|_2^2 = \min_{\|x\|=1} \left\|U\Sigma V_x^H\right\|_2^2 = \min_{\|V_x^H\|_2=1} = \min_{\|y\|_2=1} \|\Sigma y\|_2^2$$
$$= \min_{\|y\|_2=1} (\sigma_1^2 y_1^2 + \dots + \sigma_n^2 y_n^2) \geq \sigma_n^2$$

**Theorem**
Sei $A \in \mathbb{C}^n$. Dann gilt: $\|A\|_2 = \sigma_1(A)$. Falls $A$ invertierbar ist, dann gilt:

$$\text{cond}_2(A) = \frac{\sigma_1}{\sigma_m}$$

**Definition**
Die **Frobeniusnorm** der $m \times n$-Matrix $A$ ist

$$\|A\|_F^2 := \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2$$

**Theorem**
Für die $m \times n$-Matrix $A$ mit Rang $r$ gelten die Singulärwertzerlegung $A = U\Sigma V^H$ mit $m \geq n$ und $U = [\vec{u}_1 \ \dots \ u_m]$ und $V = [v_1 \ \dots \ v_n]$. Für $k \in \{1, \dots, r\}$ sei die $m \times k$-Matrix $U_k = [\vec{u}_1 \ \dots \ \vec{u}_k]$, die $n \times k$-Matrix $V_k = [\vec{v}_1, \dots, \vec{v}_k]$ und die $k \times k$-Matrix $\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k)$. Für $\|\cdot\| = \|\cdot\|_F$ und $\|\cdot\| = \|\cdot\|_2$ gilt dann:

$$\left\|A - U_k\Sigma_k V_k^H\right\| \leq \|A - B\|$$

für alle $m \times n$-Matrizen $B$ von Rang $k$

# 8 Ausgleichsrechnung

**Bemerkung**
$\text{cond}(A^T A) = \text{cond}(A^2)$

**Bemerkung**
Norm minimieren: finde $x$ sodass $A^T Ax = A^T b$

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

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

Bei der Ausgleichsrechnung: $m > n$. Das erste GLS ist somit überbestimmt und es gibt keine Lösung. Stattdessen erhalten wir das Minimierungsproblem:
**Least Squares Problem**: $\vec{x}^\star = \text{argmin}_{\vec{x}} \sum_{i=1}^m |f_{\vec{x}}(t_i) - y_i|^2 = \text{argmin}_{\vec{x}} \left\|\vec{R}\right\|_2^2$

**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:

$$\underbrace{\begin{pmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & \ddots & \vdots \\ b_1(t_m) & \dots & b_n(t_n) \end{pmatrix}}_{:=A \in M(m\times n;K)} \underbrace{\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}}_{:=\vec{x}} = \underbrace{\begin{pmatrix} y_1 \\ \vdots \\ x_m \end{pmatrix}}_{:=\vec{b}} \Leftrightarrow A\vec{x} = \vec{b} \Leftrightarrow \left\|A\vec{x} - \vec{b}\right\| = 0$$

Da $m > n$ ist GLS $A\vec{x} = \vec{b}$ wie im allgemeinen Fall überbestimmt. Wir lösen also Stattdessen das für den linearen Fall umschriebene **Least Squares Problem**:
$x^\star = \text{argmin}_{\vec{x}} \left\|A\vec{x} - \vec{b}\right\|_2^2$
Code: numpy.linalg.lstsq(A,b)[0]

Normalengleichung
Idee: Bestimme Minimum durch: $\text{grad}(\left\|A\vec{x} - \vec{b}\right\|_2^2) \stackrel{!}{=} 0$
$\vec{x}^\star = \text{argmin} \left\|A\vec{x} - \vec{b}\right\|_2^2 \Leftrightarrow A^T A\vec{x}^\star = A^T\vec{b}$
Code: ATA = np.dot(A.T,A)
ATb = np.dot(A.T,b)
xstar = numpy.linalg.solve(ATA,ATb)

QR-Zerlegung
Idee: $A = QR = \hat{Q}\hat{R}$
$\left\|A\vec{x} - \vec{b}\right\|_2^2 = \left\|QR\vec{x} - \vec{b}\right\|_2^2 = \left\|R\vec{x} - Q^T\vec{b}\right\|_2^2$
$= \left\|\hat{R}\vec{x} - \hat{Q}^T\vec{b}\right\|_2^2 + \left\|\begin{pmatrix} q_{n+1} \\ \vdots \\ q_m \end{pmatrix}\vec{b}\right\|_2^2$
Minimiere den von $\vec{x}$ abhängigen Teil exakt:
$\vec{x}^\star = \text{argmin} \left\|A\vec{x} - \vec{b}\right\|_2^2 \Leftrightarrow \hat{R}\vec{x}^\star = \hat{Q}^T\vec{b}$
Code: Qhat,Rhat = numpy.linalg.qr(A)
xstar = numpy.linalg.solve(Rhat , dot(Qhat.T , b))

Singulärwertzerlegung
Idee: $A = U\Sigma V^T = \begin{pmatrix} U_1 & | & U_2 \end{pmatrix}\begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix}$
Dabei ist $r = \text{rang}A$ = Anzahl Singulärwerte $\neq 0$, $u_1 \in \mathbb{R}^{m\times r}$, $V_1 \in \mathbb{R}^{n\times r}$. Analoge Überlungen wie bei QR liefert:
$\vec{x}^\star = \text{argmin} \left\|A\vec{x} - \vec{b}\right\|_2^2 \Leftrightarrow \vec{x}^\star = V_1\Sigma_r^{-1}U_1^T\vec{b}$
Code: psinvA = numpy.linalg.pinv(A)
xstar = dot(psinvA,b)

Allgemeines Vorgehen

1. Daten implementieren: t = np.array($[t_1, \dots, t_m]$), y = np.array($[y_1, \dots, y_m]$)
2. Basisfunktionen und $A$ bestimmmen:
   b = lambda s: np.array($[b_1(s), \dots, b_n(s)]$),
   A = np.array($[b(s)$ for $s$ in $t]$)
3. Ausgleichsrechnung lösen: lstsq, red. QR oder SVD
4. Lösung plotten:
   tnew = np.linspace(t[0],t[-1],1000)
   f = lambda t : np.dot(x,b(t))
   plt.plot(t,y)
   plt.plot(tnew,f(tnew))

**Nichtlineare-Ausgleichsrechnung**
$f_{\vec{x}}$ ist nicht linear in $\vec{x}$. Ziel: Minimierung der Quadrate: $\vec{x}^\star = \text{argmin}_{\vec{x}} \sum_{i=1}^m |f_{\vec{x}}(t_i) - y_i|^2$
Definiere:

$$F(\vec{x}) := \begin{pmatrix} f_{\vec{x}}(t_1) - y_1 \\ \vdots \\ f_{\vec{x}}(t_m) - y_m \end{pmatrix}$$

$$\phi(\vec{x}) := \frac{1}{2}\|F(\vec{x})\|_2^2 \to \vec{x}^\star = \text{argmin}_{\vec{x}} \sum_{i=1}^m |f_{\vec{x}}(t_i) - y_i|^2$$

$$\Leftrightarrow \vec{x}^\star = \text{argmin}\phi(\vec{x})$$

Code: xstar = scipy.optimize.leastsq($F, x_0$)[0]
mit $x_0$ einem guten Startwert, oftmals reicht ein Einervektor: x0 = np.ones(n)

Newton Verfahren
Idee: $\phi(\vec{x})$ minimal $\Leftrightarrow \text{grad}(\phi(\vec{x})) = 0 \Rightarrow$ bestimme NST mit Newtonverfahren. Wir brauchen dazu auch die Ableitung von $\text{grad}(\phi(\vec{x}))$, also die Hessematrix.
Vorgehen:
- $\text{grad}(\phi(\vec{x})) := (DF(\vec{x}))^T F(\vec{x})$
- $Hess(\phi(\vec{x})) := (DF(\vec{x}))^T DF(\vec{x}) + \sum_{j=1}^m F_j(\vec{x}) Hess(F_j(\vec{x}))$
- $x^\star = \text{Newton}(\text{grad}(\phi), Hess(\phi), \vec{x}_0, \text{tol}, \text{maxit})$

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

# 9 Eigenwertprobleme

Code: w,V = scipy.linalg.eig(A)
eigh für hermitische Matrizen
w Vektor mit eigenwerten
V Matrix mit V[:,i] (=i-te Spalte) EV zum EW w[i]

**Definition**
Der **Reyleigh-Quotient** von $x$ ist definiert als

$$\rho_A = \frac{x^H Ax}{x^H x}$$

**Bemerkung**
Wenn $x$ ein Eigenvektor ist, dann ist $Ax = \lambda x$ und $\rho_A(x) = \lambda$. Wenn $x$ allgemein ist, dann ist

$$\rho_A = \text{argmin}_\alpha \|Ax - \alpha x\|_2$$

**Bemerkung**
Für $A \in \mathbb{R}^{n\times n}$, $A^T = A$ und $\lambda_1, \dots, \lambda_n \in \mathbb{R}$, $q_1, \dots, q_n \in \mathbb{R}^n$ sind orthogonale Eigenvektoren. Wir berechnen die Ableitung von $\rho_A : \mathbb{R}^n \to \mathbb{R}^n$:

$$\frac{\partial \rho_A}{\partial x_j} = \frac{1}{x^T x}(Ax - \rho_A(x)x)$$

sodass $\nabla\rho_A(x) = \frac{2}{\|x\|^2}(Ax - \rho_A(x)x)$. Somit sind die Eigenwerte von $A$ die Stationärpunkte von $\rho_A$. Der Satz von Taylor für $\rho_A$ gibt dann

$$\rho_A(x) - \rho_A(q_j) = O(\|x - q_j\|^2) \qquad \text{für } x \to q_j$$

was bedeutet, dass der Rayleigh-Quotient quadratisch akkurat ist.

**Direkte Potenzmethode**
Ziel: Finde dan Betragsgrössten EW von $A$ und ein EV dazu.
Annahme: $A$ ist diagonalisierbar: $S^{-1}AS = \text{diag}(\lambda_1, \dots, \lambda_n)$ mit
$|\lambda_1| > |\lambda_1| \cdots \geq |\lambda_n|$ (> und ≥ sind extra so)

$$\frac{A^k x}{\|A^k x\|} \to S_1 \quad \text{für } k \to \infty$$

Idee: notiere $x_k = A^k x$ und berechne $\rho_A(x_k)$:

$$\rho_A(x_k) = \frac{x_k^H A x_k}{x_k^H x_k} = \frac{1}{x_k^H x_k}\left(x_k^H \sum_{j=1}^{n} a_j \lambda_j^{k+1} s_j\right) = \lambda_1 + \mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^k\right)$$

Code:
Wähle $x_0$ zufällig, $\|x_0\| = 1$
für $k = 1, 2, \dots$
$w = Ax_{k-1}$
$x_k = \frac{w}{\|w\|}$
$\lambda_k = x_k^H A x_k$

**Theorem**
Die Potenzmethode liefert eine Iteration die linear gegen $\lambda_1$ konvergiert mit der Rate $\left|\frac{\lambda_2}{\lambda_1}\right|$

**Bemerkung**
Falls $A$ normal ⇒ EV orthogonal ⇒ Fehler $\mathcal{O}\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}\right)$ ⇒ quadratische Konvergenz

**Inverse Potenzmethode**
Ziel: Finde den kleinsten EW von $A$
Annahme: $A$ ist invertierbar
$Ax = \lambda x \Rightarrow x = \lambda A^{-1}x \Rightarrow \frac{1}{\lambda}x = A^{-1}x$
⇒ betragskleinste EW $\lambda_n \Rightarrow \frac{1}{\lambda_n}x = A^{-1}x$
$\frac{1}{\lambda_n}$ ist der betragsgrösste EW von $A^{-1}$

**Bemerkung**
Wir berechnen nicht $A^{-1}$ sondern nur einmal eine LU-Zerlegung von $A$ (strukturerhaltend). Dann löse GLS mit Matrizen L,U um $A^{-1}x$ zu implementieren.

**Shifted Inverse Iteration**
Ziel: Finde den EW von $A$ am nächsten bei $\alpha$
$|\alpha - \lambda| = \min\{|\alpha - \mu| \text{ mit } \mu \text{ EW von } A\}$
$Ax - \lambda x \Leftrightarrow Ax - \alpha Ix = (\lambda - \alpha)x \Leftrightarrow (A - \alpha I)x = (\lambda - \alpha)x \Leftrightarrow \frac{1}{\lambda - \alpha}x = (A - \alpha I)^{-1}x$
⇒ Potenzmethode für $(A - \alpha I)^{-1} \Rightarrow \frac{1}{\lambda - \alpha} \Rightarrow \lambda$

**Bemerkung**
Die Potenzmethode ist schneller wenn $\alpha \approx \lambda_j$
Idee: wähle $\alpha$ adaptiv, zB. $\alpha = \rho_A(x_{k-1})$ im $k$-ten Schritt.
⇒ beschleunigte Konvergenz: **Rayleigh-Quotienten-Iteration**

**Bemerkung**
Wir brauchen immer einen guten Startwert.
Konvergenzordnung 3!

**Theorem** (Courant-Fischer)

$$\lambda_k = \min_{\dim U = k} \max_{x \in U} \rho_A(x) = \max_{\dim U = n-k+1} \min_{x \in U} \rho_A(x)$$

**Krylov-Verfahren**
gut für kleine Matrizen.

**Definition**
Der $l$-te Krylov-Raum ist definiert als:

$\mathcal{K}_l(A, z) = \text{span}\left\{z, Az, \dots, A^{l-1}z\right\} = \{p(A)z; p = \text{ Polynom vom Gra}$

Finde ONB von Krylov-Raum mittels Gram-Schmidt-Verfahren.

**Arnoldi-Verfahren** (Code: Seite 284)

1) Wähle $l < n$ selber
2) Sei $\{v_1, \dots, v_{l-1}\}$ eine ONB von $K_{l-1}(A, z)$, dann:
   $\tilde{v}_l = Av_{l-1} - \sum_{j=1}^{l-1} \langle v_j, Av_{l-1}\rangle$
   $v_l = \frac{\tilde{v}_l}{\|\tilde{v}_l\|}$
3) Eigentlich: (**Vorgehen**)
   $z$ beliebig
   $v_1 = \frac{z}{\|z\|}$
   für $l = 1, 2, ..., k-1$
   $\tilde{v} = Av_l$
   $h_{jl} = v_j^H \tilde{v}$
   $\tilde{v} = \tilde{v} - h_{jl}v_j$
   $h_{l+1,l} = \|\tilde{v}\|$
   $v_{l+1} = \frac{\tilde{v}}{h_{l+1,l}}$

**Bemerkung**
Falls $h_{l+1,l} = 0 \Rightarrow$ Abbruch der Iteration. $Av_l \in \mathcal{K}_l(A, z)$

Daraus definieren wir: $V_l := (v_1 \mid \dots \mid v_l)$ mit $\{v_1, \dots, v_l\}$ ONB von $\mathcal{K}_l$ und Hessenbergmatrizen:

$$H_l := \begin{pmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,l} \\ h_{2,1} & \ddots & & \vdots \\ & \ddots & \ddots & \vdots \\ 0 & & h_{l,l-1} & h_{l,l} \end{pmatrix} \in \mathbb{R}^{l\times l}$$

$$\tilde{H} := \begin{pmatrix} & & H_l & \\ & & & \\ 0 & \dots & 0 & h_{l+1,l} \end{pmatrix} \quad \text{mit } \tilde{h}_{ij} = \begin{cases} h_{ij} = v_i^H A v_j \text{ für } i \leq j \\ \|\tilde{v}_j\|_2 \text{ für } i = j+1 \\ 0 \text{ sonst} \end{cases}$$

**Bemerkung**

1) $V_l^H V_l = i_l$, da $v_1, \dots, v_l$ orthogonal sind.
2) $H_l = V_l^H A V_l$
3) Arnolli-Verfahren bricht ab, falls $h_{l+1,l} = 0$ und dann: $AV_l = V_l H_l$ und $\mathcal{K}_{l+1} = 0$
4) Falls $A$ Hermit-symmetrisch ist ($A^H = A$), dann gilt: $H_l^H = H_l$. Somit ist $H_l$ eine tridiagonale Matrix.

Code: Lanczos-Iteration: Seite 286
Arnoldi EWapproximation: Seite 288

**Theorem** (Falls $V_l$ quadratisch)
Falls $h_{l+1,l} = 0$ und $h_{j+1,j} \neq 0$, dann:

1) Jeder Eigenwert von $H_l$ ist auh EW von $A$
2) Falls $A$ regulär ist, dann gibt es $y \in \mathbb{C}^l$, sodass $Ax = b$ mit $x = V_l y$, wobei $V_l$ normal für $\mathcal{K}_l(A, b)$ konstruirt wurde.

**Theorem** (Falls $V_l$ nicht quadratisch)

1) EW von $H_l$ finden, Code: $ew, ev = eig(H_l)$
2) Achtung! ev $\neq$ EV von $A$. Man kann sie aber berechnen
3) Sei $u$ ein EV von $H_l$, dann ist der zugehörige EV von $A$: $V_l u$

**Theorem**
Seien $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ und $\mu_1^{(l)} \geq \dots \geq \mu_l^{(l)}$ die EW der Hermit-symmetrischen Matrix $A \in \mathbb{C}^{n \times n}$, bzw von $H_l = V_l^H A V_l$ für $l = 1, 2, \dots$. Dann gelten für $1 \leq j \leq l$ die Ungleichungsketten:

$$\lambda_{n-j+1} \leq \mu_{l+1-j+1}^{(l+1)} \leq \mu_{l-j+1}^{(l)} \qquad \text{und} \qquad \mu_j^{(l)} \leq \mu_j^{l+1} \leq \lambda_j$$

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

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 $x_0, \dots, x_N$ von $[a, b] \Rightarrow \Psi(x_0) = \Psi(x_N) = 0$
Noch zu bestimmen: $\Psi(x_1), \dots, \Psi(x_{N-1})$ und $\lambda$. Dazu:

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

Die gesuchten Lösungen $\Psi(x_1), \dots, \Psi(x_{N-1})$ sind somit die EV der Matrix $A$ und die unbestimmten Werte $\lambda$ dia dazugehörigen EW.
Bem: Dieses $A$ ist dünnbesetzt $\rightarrow$ Krylov für grosses $N$ nützlich.