Linalg

LU	Skript	S. 186
Cholesky	"	S. 186
QR		S. 190
SVD		S. 196

Ausgleichsrechnung (linear)

```
def solve_normal (A,b):
    A_H = A.conjugate().transposé()
    return solve( dot(A_H, A), dot(A_H, b))
```

```
def solve_qr (A,b)
    Q,R = scipy.linalg.qr(A, mode="economic")
    return solve_triangular(R, dot(Q.transpose(), b))
```
scipy.linalg

Exact: `scipy.linalg.lstsq(A,b)[0]`

```
def solve_svd (A, b, eps=0):
    U, S, Vh = scipy.linalg.svd(A)
    r = 1 + where(S/S[0] > eps)[0].max()
    return dot(Vh[:r,:].T, dot(U[:,:r].T, b)/S[:r])
```

```
def solve_svd (A, b):
    U,S,Vh = scipy.linalg.svd(A)
    r = S.shape[0] ; Ux = U[:,:r]
    return dot(Vh[:,:r].T, dot(np.diag(1/S), dot(Ux.T, b)))
```

anz. messungen

x = linspace(t_0, t_end, m)

y = array mit messresultat

```
def monom_basis (deg):
    basis = []
    for n in range(deg+1):
        def monom(x, deg=n)
            return power(x,deg)
        basis.append(monom)
    return basis
```

```
def build_system (x, y, basis):
    n = len(basis); m = x.shape[0]
    A = zeros((m,n))
    for j in range(n):
        A[:,j] = basis[j](x)
    return A, y
```

(nicht linear) S. 212

```
def Gauss_newton (F, DF, x0, tol=10e^-6, maxit=100)
    for i in range(maxit):
        s = linalg.lstsq(DF(x0), F(x0))[0]
        x0 = x0 - s
        if norm(s) < tol * norm(x0)
            return x0, True, i+1
    return x0, False, maxit
```

$$F = \begin{pmatrix} f(x,t_1) - y_1 \\ f(x,t_m) - y_m \end{pmatrix}$$

DF = Jacobi Matrix von F

$x = \operatorname{argmin} \|F(x)\|_2^2$

Residuen vektor

scipy.optimize.leastsq(F(x), x0)

Exact: `scipy.optimize.curve_fit(f, [t_1,...,t_m], [y_1,...,y_m])[0]`

Splitting

```
def compile_splitting_step(phi_a, phi_b, a, b):
    def splitting_step(y0, dt):
        y = y0
        for i in range(a.shape[0]):
            y = phi_a(y, a[i]*dt)
            y = phi_b(y, b[i]*dt)
        return y
    return splitting_step
```

```
def integrate_splitting(step, y0, t0, tend, N):
    t, dt = linspace(t0, tend, N, retstep=True)
    y = zeros((t.shape[0], y0.shape[0]))
    y[0] = y0
    for i in range(N-1):
        y[i+1] = step(y[i], dt)
    return t, y
```

Runge Kutta

```
class RKSolver:
    def __init__(self, A, b, c):
        s = A.shape[0]
        if array_equal(tril(A, k=-1), A):
            self.__k = self._explicit_k
        else:
            self.__k = self._implicit_k
        self.__dict__.update(locals())

    def _explicit_k(self, k0, rhs, y0, t0, dt):
        d = y0.shape[0]
        k = zeros((self.s, d))
        for i in range(self.s):
            k[i] = rhs(y0 + dt*dot(self.A[i], k), t0 + dt*self.c[i])
        return k

    def _k_equation(self, k, rhs, y0, t0, dt):
        gk = zeros_like(k)
        for i in range(self.s):
            gk[i] = rhs(y0 + dt*dot(self.A[i], k), t0 + dt*self.c[i])
        return k - gk

    def _implicit_k(self, k0, rhs, y0, t0, dt):
        return -fsolve2(self.__k_equation, k0, args=(rhs, y0, t0, dt))

    def integrate(self, rhs, y0, t0, tend, N):
        d = y0.shape[0]; t, dt = linspace(t0, tend, N, retstep=True)
        y = zeros((N, d)); y[0] = y0; k = zeros((self.s, d))
        for n in range(N-1):
            k = self.__k(k, rhs, y[n], t[n], dt)
            y[n+1] = y[n] + dt*dot(self.b, k)
        return t, y
```

ausserhalb class

```
def _fsolve2(f, x0, args):
    shape = x0.shape
    flat_x0 = x0.reshape((-1))
    def flat_f(x, *args):
        return f(x.reshape(shape), *args).reshape(-1)
    x = fsolve(flat_f, flat_x0, args)
    return x.reshape(shape)
```

Bsp

```
rk1 = RKSolver(A, b, c)
t, y = rk1.integrate(rhs, y0, t0, tend, N)
```

ODE mit Matrix A $\quad \dot{v} = A \cdot v$

```
def integrate_krylov(A, v, dt, k, method):
    V,H = method(A,v,k)
    eH = scipy.linalg.expm(dt*H)
    r = dot(V[:,:-1], eH)
    return r[:,0]
```

```
def integrate_scipy(A, v, dt):
    return dot(scipy.linalg.expm(dt*A), v)
```

```
def rkstep(f, y0, t0, h, s):
    k = zeros((y0.size, s))
    for i in range(s):
        k[:,i] = f(t0 + c[i]·h, y0 + h·dot(k, A[i]))
    y = y0 + h·dot(k, b)
    return y
```

```
def integrate(f, y0, t0, tend, N, A, b, c):
    T, h = linspace(t0, tend, N, retstep=True)
    Y = zeros((y0.size, N))
    Y[:,0] = y0
    s = len(b)
    for i in range(N-1):
        Y[:,i+1] = rkstep(f, Y[:,i], T[i], h, s)
    return T, Y
```

```
def integrate_diagonalize(A, v, dt):
    ew, ev = scipy.linalg.eig(A)
    D = diag(exp(dt*ew))
    y = scipy.linalg.solve(ev, v)
    y = dot(D, y)
    y = dot(ev, y)
    return y
```

Rosenbrock-Euler

```
def expEV(N, t0, tend, y0, f, Df):
    ts, h = linspace(t0, tend, N, retstep=True)
    y0 = atleast_1d(y0)
    y = zeros((N, y0.shape[0]))
    y[0,:] = y0
    for i, ti in enumerate(ts[1:]):
        DF = atleast_2d(Df(y[i,:]))
        y[i+1,:] = y[i,:] + dot(expm(h*DF) - eye(*DF.shape),
                                solve(DF, f(y[i,:])))
    return ts, y
```

```
def Gr(a):
    m,n = a.shape
    q = zeros((m,n))
    r = zeros((n,n))
    for j in range(n)
        vj = a[:,j].copy()
        for i in range(j)
            r[i,j] = np.dot(q[:,i], a[:,j])
            vj = vj - r[i,j] * q[:,i]
        r[j,j] = norm(vj)
        q[:,j] = vj / r[j,j]
    return q, r
```

```
def Grmod(a):
    m,n = a.shape
    q = zeros((m,n))
    r = zeros((n,n))
    v = a.copy()
    for i in range(n):
        r[i,i] = norm(v[:,i])
        q[:,i] = v[:,i] / r[i,i]
        for j in range(i+1, n):
            r[i,j] = dot(q[:,i], v[:,j])
            v[:,j] = v[:,j] - r[i,j] * q[:,i]
    return q, r
```

Phasenraum $\Rightarrow$ q gegen p plotten

$$H = \frac{p^2}{2} + V(q)$$

$$\dot{q} = \frac{\partial H}{\partial p} = p$$

$$\dot{p} = -\frac{\partial H}{\partial q} = -\nabla V(q)$$

$$Y = \begin{pmatrix} q \\ p \end{pmatrix}$$

$$\dot{Y} = \begin{pmatrix} \dot{q} \\ \dot{p} \end{pmatrix} = \begin{pmatrix} p \\ -\nabla V(q) \end{pmatrix} = \begin{pmatrix} p \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ -\nabla V(q) \end{pmatrix}$$

Falls Reibung:

$$\ddot{q} + \mu \cdot q + \nabla V(q) = 0$$

$$\dot{Y} = \begin{pmatrix} p \\ -\mu \dot{q} - \nabla V(q) \end{pmatrix} = \begin{pmatrix} p \\ -\mu p - \nabla V(q) \end{pmatrix}$$

Adaptives Verfahren

```
def integrate(f, y0, t0, tend, h0, hmax, atol):
    t = np.empty((1,))
    y = np.empty((1, y0.size))
    y[0] = y0
    t[0] = t0
    h = h0
    while (t[-1] != tend):
        y2 = method2(y0, f, h)
        y3 = method3(y0, f, h)
        if (norm(y2-y3) < atol):
            y = np.vstack([y, y3])
            t = np.append(t, t[-1]+h)
            h = min([2.*h, tend-t[-1], hmax])
        else:
            h = 0.5*h
    return t, y
```

2D - Integration

```
F = lambda y: method(lambda x: f(x,y), a, b, Nx)
I = method(F, c, d, Ny)
```

in method f(x) mit array(list(map(f,x)))

ersetzen z.B. sum(np.array(list(map(f, x[::2])))) Skal! sum(f(x[::2]))

Diagonalisieren

```
D, V = eig(A)
A = V @ diag(D) @ V^-1
```

```
def semirk_skp(y0, t0, h, f):
    S = A.shape[0]
    k = empty((y0.size, S))
    for i in range(S):
        initial_guess = y0
        k[:,i] = fsolve(F, initial_guess)
    y1 = y0 + h*sum(b*k, axis=-1)
```

* F = lambda x: x - f(t + c[i]*h, y0 + h*sum(A[i,:]*k[:,:], axis=-1) + A[:,:]*x)

nebsdian Störmer-Verlet

```
def leap_frog(rhs, xy0, v0, N, tend):
    t, dt = linspace(0, tend, N+1, retstep=True)
    xy = zeros((N+1, xy0.size))
    v = zeros((N+1, v0.size))
    xy[0,:] = xy0
    v[0,:] = v0
    for i in range(N):
        v12 = v[i,:] + 0.5*dt*rhs(xy[i,:])
        xy[i+1,:] = xy[i,:] + dt*v12
        v[i+1,:] = v12 + 0.5*dt*rhs(xy[i+1,:])
    return t, xy
```

```
def stoermer_verlet(rhs, xy0, v0, N, tend):
    t, dt = linspace(0, tend, N+1, retstep=True)
    xy = zeros((N+1, xy0.size))
    xy[0,:] = xy0
    xy[1,:] = xy[0,:] + dt*v0 + 0.5*dt**2*rhs(xy[0,:])
    for i in range(1, N):
        xy[i+1,:] = -xy[i-1,:] + 2*xy[i,:] + dt**2*rhs(xy[i,:])
    return t, xy
```

```
def lennard_acceleration(xy):
    r = linalg.norm(xy)
    dvdt = 24*xy*(2/r**14 - 1/r**8)
    return dvdt
```

```
def integrate(method, rhs, y0, T, N):        # array([x, v])
    y = empty((N+1,) + y0.shape)
    t0, dt = 0.0, T/N
    y[0, ...] = y0
    for i in range(0, N):
        y[i+1, ...] = method(rhs, y[i, ...], t0 + i*dt, dt)
    t = arange(N+1)*dt
    return t, y
```

Newton ohne lstsq.

```
def newton(f, df, x0, l=1, rtol=1e-14, maxit=1000)
    x = x0
    for i in range(maxit)
        delta = l * solve(df(x), f(x))
        x -= delta
        if norm(delta) < rtol
            return x, True, i+1
    return x, False, maxit
```

```
def velocity_verlet_step(rhs, xv0, t0, dt):        # (t, x)
    xv0 = xv0.reshape((2, -1))
    xvl = np.empty_like(xv0)
    x0, xl = xv0[0, :], xvl[0, :]
    v0, vl = xv0[1, :], xvl[1, :]
    xl[:] = x0 + dt*v0 + 0.5*dt**2 * rhs(t0, x0)
    vl[:] = v0 + 0.5*dt*(rhs(t0, x0) + rhs(t0+dt, xl))
    return xvl.reshape(-1)

def velocity_verlet(rhs, y0, T, N):
    return integrate(velocity_verlet_step, rhs, y0, T, N)
```

$$R(p) = \begin{pmatrix} F(t_1, p) - y_1 \\ \vdots \\ F(t_n, p) - y_n \end{pmatrix}$$

Df ist nach $p[0], p[1], \ldots p[k]$

scipy.optimize.leastsq(R(x), x0)

Quadratur

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

SR: Skript S. 12

TR: Skript S.13

Golub/Welsch: Skript S. 17

Gaussquad: Skript S 19

Adapt. quad: Skript S. 23

composite_legendre, zusammengesetzte Gauss Legendre

```
def composite_legendre_error(f, exact)
    n= ....
    rule = lambda f,a,b,N: comp.leg.(f,a,b,N,n)
    error, n_chunks = quadError(rule,f, exact)
    return errors, n * n_chunks
```

```
def MPR(f,a,b,N)
    x,h = linspace(a,b, N+1, retstep=True)
    c= x + 0.5 h
    return sum(h*f(c[:-1]))
```

order

```
N_array= 2** arange(5,12)
error = zeros_like(N_array, dtype=float)
I_exact = integrate.quad(f,a,b)[0]
for j,N in enumerate(N_array):
    I= method(f,a,b,N)
    error[j]= abs(I - I_exact)
order = polyfit(log(N_array), log(error), 1)[0]
plt.loglog(N_array, error)
```

```
def gauss_legendre(f,a,b,n)        # anzahl Quadralpunkte
    x_ref, wref = golub_welsch(n)
    x= a+ (x_ref +1.0) · (b-a) ·0,5
    w= 0.5 · (b-a) · w_ref
    return np.sum(w·f(x))
```

```
def composite_legendre(f,a,b,N,n)   # anz. Teilintervalle
    I=0
    dx= (b-a)/N
    for i in range(N):
        I += gauss_legendre(f,a+ i*dx, a+ (i+1)·dx, n)
    return I
```

ODE

f(y, t)

```
Exact: def scipy(f, y0, t0, tend, N):
    T= linspace(t0, tend, N)
    return T, odeint(f, y0, T)
```

(odeint: sc.integrate)

Energie

```
def eE/iE/iMP(f, y0, t0, tend, N):
    t, dt = linspace(t0, tend, N, retstep=True)
    y = zeros((t.shape[0], y0.shape[0]))
    y[0] = y0
    for k in range(t.shape[0]-1):
(ee)    y[k+1] = y[k] + dt * f(y[k], t[k])
(ie)    y[k+1] = fsolve(lambda yk1: y[k] + dt * f(yk1, t[k+1]) - yk1, y[k])
(iMP)   y[k+1] = fsolve(lambda yk1: y[k] + dt * f(0.5*(y[k]+yk1), 0.5*(t[k]+t[k+1])) - yk1, y[k])
    return t, y
```

```
def Stoermer_verlet(f, y0, v0, t0, tend, N):
    t, dt = linspace(t0, tend, N, retstep=True)
    y = zeros(N)
    y[0] = y0
    y[1] = y0 + dt*v0 + dt**2 * 0,5 * f(y0, t0)
    for k in range(1, N-1):
        y[k+1] = -y[k-1] + 2*y[k] + dt**2 * f(y[k], t[k])
    return t, y
```

order

```
N_array = 2** arange(5,12)
errors = zeros_like(N_array, dtype=float)
for j,N in enumerate(N_array):
    t, y = method(..., N)
    errors[j]= linalg.norm(y[-1] - y_exact[-1])     # Schlussresultat matte
    errors[j]= norm(y - y_exact)                    # allgemein
    errors[j]= norm(y - y_exact, axis=1).max()      # max. norm
order = polyfit(log(N_array), log(errors), 1)[0]
```

Energie

```
def velocity_verlet(f, y0, v0, t0, T, N):
    t, dt = ....
    y= zeros(N) ; y[0]= y0
    v= zeros(N) ; v[0]= v0
    for k in range(N-1):
        y[k+1]= y[k] + dt*v[k] + dt**2 * 0.5 * f(y[k], t[k])
        v[k+1]= v[k] + dt*0,5 * f(y[k], t[k]) + f(y[k+1], t[k+1])
    return t, y, v
```

```
t,y = ode45(f, [t0, tend], y0)
```

f(t,z) — array

Eigenwerte

eigenwerte / vektoren — eigh falls hermite symmetrisch

Exact: `w, v = scipy.linalg.eig(A)`

QR-Algorithmus S. 223

S. 230

```
def inverse_potenz_method(A, rtol=10e-6)
    LUP = scipy.linalg.lu_factor(A)
    x = random.rand(A.shape[0])
    x /= np.linalg.norm(x)
    ew_before = 0
    while (True):
        lu_solve(LUP, x, overwrite_b=True)
        ew = 1 / norm(x)
        x* = ew
        if (abs(ew - ew_before) < rtol * ew
            break
        ew_before = ew
    return np.dot(x.transpose(), np.dot(A,x))
```

```
def potenz_method(A, rtol=10e-6):
    x = np.random.rand(A.shape[0])
    x /= np.linalg.norm(x)
    ew = 1 ; ew_before = 0
    while (abs(ew - ew_before) > rtol * ew:
        ew_before = ew
        x = dot(A,x)
        ew = linalg.norm(x)
        x /= ew
    return ew
```

```
def shifted_inverse(A, c, rtol=10e-6):
    return inverse_potenz_method(A - c*np.eye(A.shape[0])) + c
```

S. 237

```
def arnoldi(A, v0, k):
    r, c = A.shape
    V = zeros((r, k+1), dtype=complexfloating)
    H = zeros((k+1, k), dtype=complexfloating)
    V[:,0] = v0.reshape(-1) / norm(v0)
    for i in range(1, k+1):
        vi = dot(A, V[:, i-1])
        for j in range(i):
            H[j, i-1] = dot(conjugate(V[:,j]), vi)
            vi -= H[j, i-1] * V[:,j]
        H[i, i-1] = norm(vi)
        V[:, i] = vi / H[i, i-1]
    return V, H[:-1, :]
```

S. 239.

```
def lanczos(A, v0, k):
    r, c = A.shape
    V = zeros((r, k+1), dtype=complexfloating)
    H = zeros((k, k), dtype=complexfloating)
    alpha = zeros((k,), dtype=complexfloating)
    beta = zeros((k+1,), dtype=complexfloating)
    V[:, 0] = v0.reshape(-1) / norm(v0)
    for i in range(1, k+1):
        vi = dot(A, V[:, i-1])
        if i > 1:
            vi -= beta[i-1] * V[:, i-2]
        alpha[i-1] = dot(conjugate(V[:, i-1]), vi)
        vi -= alpha[i-1] * V[:, i-1]
        beta[i] = norm(vi)
        V[:, i] = vi / beta[i]
    beta = beta[1:-1]
    H = diag(alpha) + diag(beta, 1) + diag(beta, -1)
    return V, H
```

```
ew = np.linalg.eigvals(H[0:k, :])
ew, ev = eig(H[:-1, :])
```

```
ew = np.linalg.eigvals(H[0:k, :])
I = argsort(ew)
ev = dot(V[:, :-1], ev[:, I])
```

Nullstellen

10)

Bisektion S. 140

Sekanten S. 143

Exact: scipy.optimize.fsolve(f, x_0)
↳ $\mathbb{R}^n \to \mathbb{R}^n$

```
def Newton_1D(f, df, x0, L=1.0, rtol=1e-12, maxit=1000):
    x = x0
    for i in range(maxit):
        delta = L * solve(df(x), f(x))   # numpy.linalg
        x -= delta
        if norm(delta) < rtol:
            return x, True, i+1
    return x, False, maxit
```

```
def nicer_fsolve(F, guess)
    shape = guess.shape
    guess = guess.reshape((-1,))
    result = fsolve(lambda x: F(x.reshape(shape)).reshape((-1)), guess)
    return result.reshape(shape)
```

mehrdim

Newton S. 145 at least 2d löschen

dampNewton S. 150

Broyden S. 154 ($J = Df(x_0)$)

Konv. Ordnung $p \approx \dfrac{\log(e^{k+1}) - \log e^k}{\log e^k - \log e^{k-1}}$

e^k = Fehler bei k

Code:

```
p = log(e[2:]/e[1:-1])/log(e[1:-1]/e[:-2])
paverage = sum(p)/size(p)        + 1e-11
plt.semilogy(k, e)               # k: Anzahl, e: Fehler
fit = polyfit(k, log(e), 1)      # e: Fehler
L = exp(fit[0])
```

Householder

$$H(v) = I - 2\frac{v \cdot v^H}{v^H \cdot v}$$

$$V = \begin{cases} \frac{1}{2}(a + \|a\|_2 e_1), & a_1 > 0 \\ \frac{1}{2}(a - \|a\|_2 e_1), & a_1 < 0 \end{cases}$$

```
def Householder(A)
    L = len(A)
    for i in range(0, L-1):
        I = np.eye(L-1)
        ai = A[:, i]
        abs = norm(ai[i, L])
        sign = np.sign(A[i,i])
        vi = array([ai[i:L]]).T + sign * abs * array([I[:,0]]).T
        Qi = I - 2*(vi@vi.T)/(vi.T@vi)
        H = np.eye(L)
        H[i:L, i:L] = Qi
        R = I@A
    return R, A, H
```

Givens Rotation

$Q \cdot A = R$

$$Q = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}$$

Lösungsschema

$$\begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \leftarrow \text{dieser Eintrag wird zu 0}$$

G_1

$$c = \frac{a_{11}}{\sqrt{a_{21}^2 + a_{11}^2}} \qquad s = \frac{a_{21}}{\sqrt{a_{21}^2 + a_{11}^2}}$$

1) $G_1 \cdot A = B$

2) Bearbeite B mit G_2

3) $C = G_2 \cdot B$

4) Bearbeite C mit G_3

5) $Q = G_3 \cdot G_2 \cdot G_1$