lektion12

January 14, 2026

Inhalt

1 Lektion 12

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
init_printing()
%matplotlib inline
```

1.1 Gradient, Hessematrix (Wiederholung)

[2]:
```
x, y, z = symbols('x y z')
var = (x, y, z)
```

[3]:
```
f = Function('f')
f(*var)
```

[3]:
$$f(x, y, z)$$

[4]:
```
G = Matrix([f(*var)]).jacobian(var)
G
```

[4]:
$$\left[\frac{\partial}{\partial x} f(x,y,z) \quad \frac{\partial}{\partial y} f(x,y,z) \quad \frac{\partial}{\partial z} f(x,y,z)\right]$$

[5]:
```
H = hessian(f(*var), var)
H
```

[5]:
$$\begin{bmatrix} \frac{\partial^2}{\partial x^2} f(x,y,z) & \frac{\partial^2}{\partial y \partial x} f(x,y,z) & \frac{\partial^2}{\partial z \partial x} f(x,y,z) \\ \frac{\partial^2}{\partial y \partial x} f(x,y,z) & \frac{\partial^2}{\partial y^2} f(x,y,z) & \frac{\partial^2}{\partial z \partial y} f(x,y,z) \\ \frac{\partial^2}{\partial z \partial x} f(x,y,z) & \frac{\partial^2}{\partial z \partial y} f(x,y,z) & \frac{\partial^2}{\partial z^2} f(x,y,z) \end{bmatrix}$$

[6]:
```
Matrix([f(*var)]).hessian(var)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 1
----> 1 Matrix([f(*var)]).hessian(var)

AttributeError: 'MutableDenseMatrix' object has no attribute 'hessian'
```

[7]:
```
H = Matrix([f(*var)]).jacobian(var).jacobian(var)
H
```

[7]:
$$\begin{bmatrix} \frac{\partial^2}{\partial x^2} f(x,y,z) & \frac{\partial^2}{\partial y \partial x} f(x,y,z) & \frac{\partial^2}{\partial z \partial x} f(x,y,z) \\ \frac{\partial^2}{\partial y \partial x} f(x,y,z) & \frac{\partial^2}{\partial y^2} f(x,y,z) & \frac{\partial^2}{\partial z \partial y} f(x,y,z) \\ \frac{\partial^2}{\partial z \partial x} f(x,y,z) & \frac{\partial^2}{\partial z \partial y} f(x,y,z) & \frac{\partial^2}{\partial z^2} f(x,y,z) \end{bmatrix}$$

1.2 Beispiel

[8]:
```
var = (x, y)
f = 2 - x**2 / 2 - y**2
f
```

[8]:
$$-\frac{x^2}{2} - y^2 + 2$$

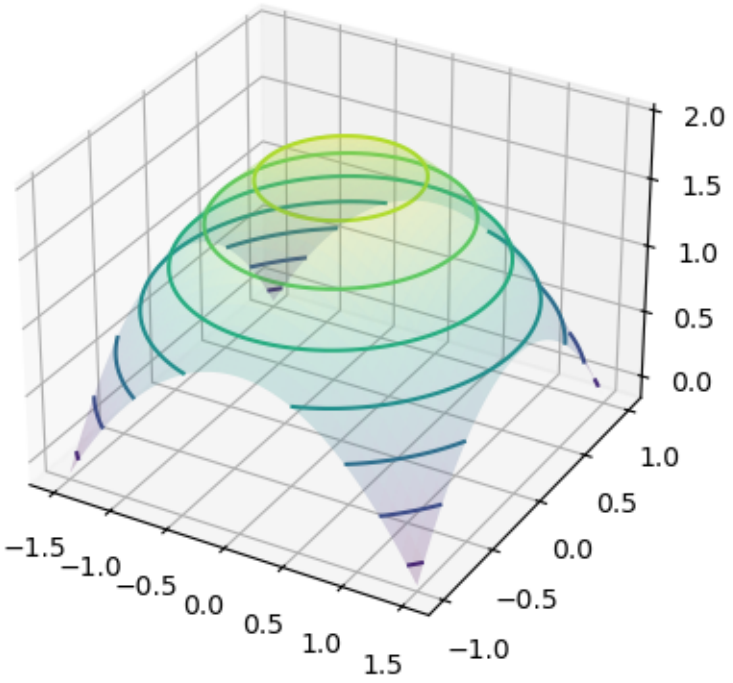
[9]:
```
fn = lambdify(var, f)
```

[10]:
```
xn = np.linspace(-1.5, 1.5, 91)
yn = np.linspace(-1, 1, 61)
X, Y = np.meshgrid(xn, yn)
F = fn(X, Y)
```

[11]:
```
fig1 = plt.figure()
ax1 = fig1.add_subplot(projection='3d')
```

```
ax1.plot_surface(X, Y, F, alpha=.2, cmap=plt.cm.viridis)
ax1.contour(X, Y, F)
```

[11]: <mpl_toolkits.mplot3d.art3d.QuadContourSet3D at 0x7fa825d4fe60>



1.2.1 Extrema (Wiederholung)

[12]:
```
ext = solve(Matrix([f]).jacobian(var))
ext
```

[12]: $\{x : 0,\ y : 0\}$

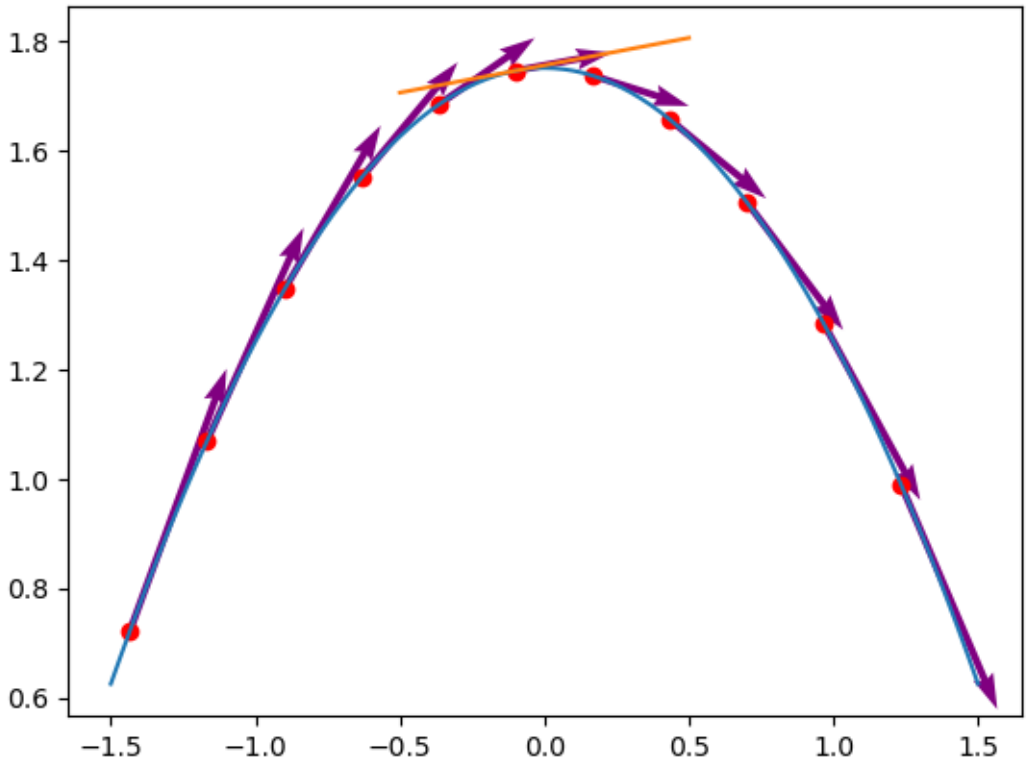
[13]:
```
H = hessian(f, var)
H
```

[13]: $\begin{bmatrix} -1 & 0 \\ 0 & -2 \end{bmatrix}$

1.2.2 Tangenten

[15]:
```
fig1d = plt.figure()
ax1d = fig1d.gca()

pnt = {x: -.1, y: -.5}


def tangente1dx(f, pnt):
    """ Parametrisierung der Tangeten an f in pnt in x-Richtung
    """
    t = np.linspace(-.5, .5)
    return t, f.subs(pnt) + f.diff(x).subs(pnt) * (t - pnt[x])


ax1d.plot(xn, fn(xn, pnt[y]))
ax1d.plot(*tangente1dx(f, pnt))

t = symbols('t')
fdxn = lambdify((x, y), f.diff(x))
xn_c = xn[2:-2:8]

ax1d.quiver(xn_c,
            fn(xn_c, pnt[y]),
            1 + 0 * xn_c,
            fdxn(xn_c, pnt[y]),
            angles='xy', scale_units='xy', scale=3, color='purple')
ax1d.scatter(xn_c, fn(xn_c, pnt[y]), c='red')
```

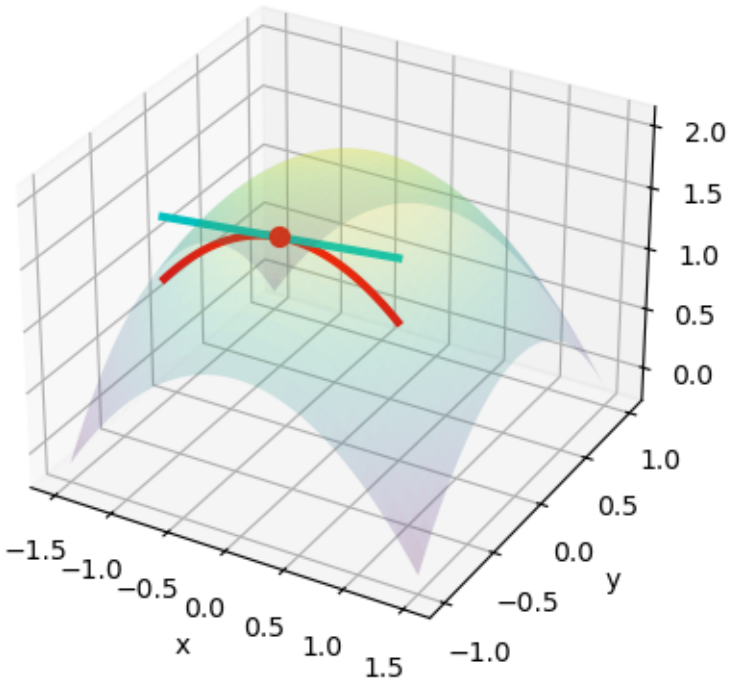
[15]:
```
<matplotlib.collections.PathCollection at 0x7fa824bd2840>
```



[16]:
```
fig1 = plt.figure()
ax1 = fig1.add_subplot(111, projection='3d')
ax1.plot_surface(X, Y, F, cmap=plt.cm.viridis, alpha=0.2)
ax1.set_xlabel('x')
ax1.set_ylabel('y')

vec = [1, 0]


def cut(pnt, f, vec, scale=1):
    '''
    Schnittkurve durch Punkt pnt in Richtung vec
    '''
    tn = np.linspace(-1, 1) / scale
    xn = pnt[x] + vec[0] * tn
    yn = pnt[y] + vec[1] * tn
    zn = lambdify((x, y), f)(xn, yn)
    return xn, yn, zn


ax1.scatter(pnt[x], pnt[y], f.subs(pnt), c='r', s=50)
ax1.plot(*cut(pnt, f, vec), 'r', lw=3)
```

```
def tangente2(pnt, f, vec, var, scale=1):
    '''
    Tangente an Punkt (pnt, f(pnt)) in Richtung vec
    als parametrische Kurve
    '''
    t_ = symbols('t')
    fc = (f + t_ *
          (Matrix([f]).jacobian(var).dot(Matrix(2, 1, vec)))).subs(pnt)
    fcn = lambdify(t_, fc)

    tn = np.linspace(-1, 1) / scale
    xn = pnt[x] + vec[0] * tn
    yn = pnt[y] + vec[1] * tn
    zn = fcn(tn)

    return xn, yn, zn


ax1.plot(*tangente2(pnt, f, vec, var), 'c', lw=3)
```

[16]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fa81ca69160>]

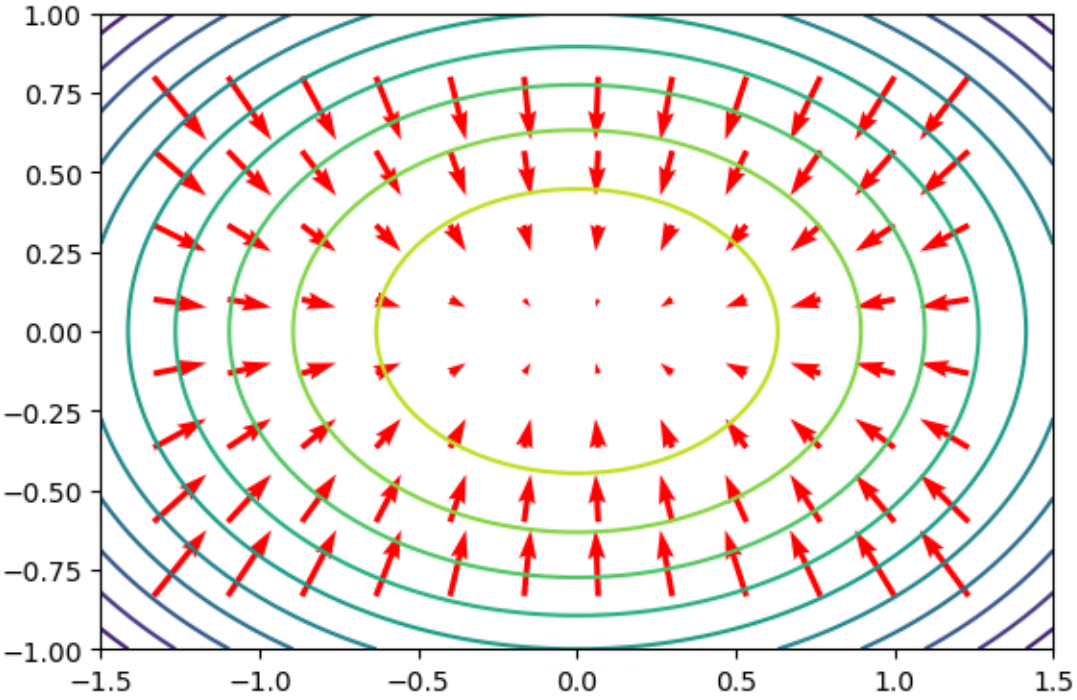


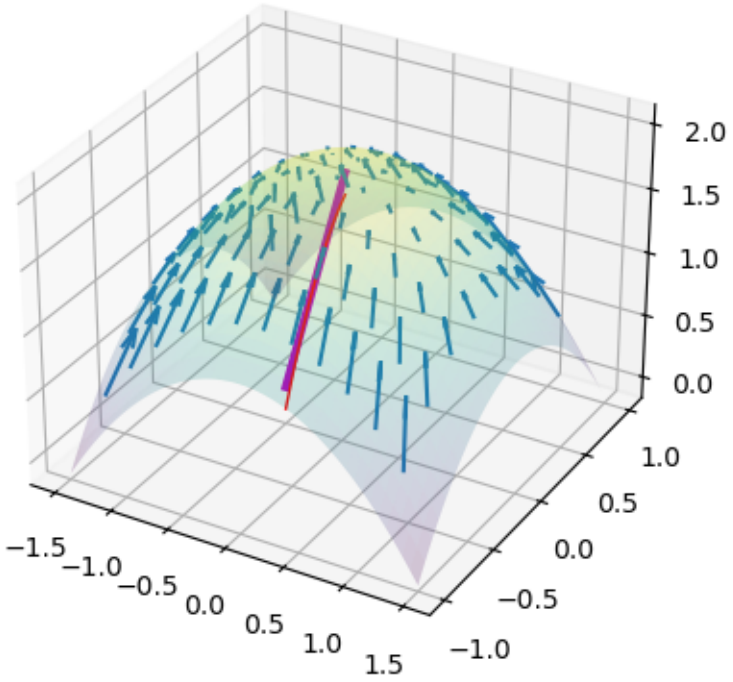
1.2.3 Gradient als Vektorfeld (quiver) Höhenlinien (contour)

[18]:
```
X, Y = np.meshgrid(xn, yn)
X_c = X[5:-2:7,5:-2:7]
Y_c = Y[5:-2:7,5:-2:7]
grad = Matrix([f]).jacobian((x, y))
gradn = lambdify((x, y), grad)
GR = gradn(X_c, Y_c)[0]
```

[19]:
```
fig = plt.figure()
ax = fig.gca()
ax.contour(X, Y, fn(X, Y), 10)
ax.set_aspect('equal', 'box')

ax.quiver(X_c, Y_c, GR[0], GR[1], angles='xy', scale_units='xy', scale=8,␣
 ↪color='red');

t_, dx, dy = symbols('t dx dy')
fc = f.subs({x: x + t_ * dx, y: y + t_ * dy})
dfc = diff(fc, t_)
dfcn = lambdify((x, y, dx, dy, t_), dfc)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, fn(X, Y), cmap=plt.cm.viridis, alpha=.2)

zG = dfcn(X_c, Y_c, GR[0, :, :], GR[1, :, :], 0)

ax.quiver(X_c, Y_c, fn(X_c, Y_c), GR[0, :, :] / 10,
          GR[1, :, :] / 10, zG / 10)



pnt = {x: X_c[1, 7], y: Y_c[1, 7]}
ax.plot(*tangente2(pnt, f, GR[:, 1, 7], var=var, scale=3), 'm', lw=3)
ax.plot(*cut(pnt, f, GR[:, 1, 7], scale=3), 'r', lw=1)
```

[19]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7fa81acd0350>]
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


2.0
1.5
1.0
0.5
0.0
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

1.3 Differentialgleichungen

vgl. Analysis II

1.3.1 erstes Beispiel

[20]:
```
import matplotlib.pyplot as plt
import numpy as np
from sympy import *
init_printing()
%matplotlib notebook
```

Gesucht ist eine differenzierbare Funktion $y : \mathbb{R} \to \mathbb{R}$ mit

$$\frac{d}{dt}y(t) = y(t).$$

[21]:
```
y = Function('y')
t, tau = symbols('t tau', real = True)
dgl = Eq(y(t).diff(t), y(t))
dgl
```

[21]:
$$\frac{d}{dt}y(t) = y(t)$$

[22]:
```
sol = dsolve(dgl, y(t))
sol
```

[22]:
$$y(t) = C_1 e^t$$

[23]:
```
sol.subs(symbols('C1'), 1)
```

[23]:
$$y(t) = e^t$$

[24]:
```
c1_s=symbols('C1')
```

Wir ergänzen die Differentialgleichunng mit einer Anfangsbedingung zur einem Anfangswertproblem.

Gesucht ist $y : \mathbb{R} \to \mathbb{R}$ mit

$$\begin{aligned} \frac{d}{dt}y(t) &= y(t) \\ y(t_0) &= y_0 \end{aligned}$$

[25]:
```
t0, y0 = symbols('t_0 y_0')
eq = sol.subs({t: t0}).subs({y(t0): y0})
eq
```

[25]:
$$y_0 = C_1 e^{t_0}$$

[26]:
```
sol = sol.subs(c1_s, solve(eq, c1_s)[0])
sol
```

[26]:
$$y(t) = y_0 e^{t} e^{-t_0}$$

[27]:
```
checkodesol(dgl, sol)
```

[27]:
```
(True, 0)
```

Das geht auch einfacher

[28]:
```
aw = {y(t0): y0} # Anfangswerte (initial conditions)
```

[29]:
```
dsolve(dgl, y(t), ics=aw)
```

[29]:
$$y(t) = y_0 e^{t} e^{-t_0}$$

1.3.2 Inhomogene lineare Differentialgleichung erster Ordnung

$$u'(t) = u(t) + \sin(t)$$

[30]:
```
u = Function('u')
t = symbols('t', real=True)
dgl = Eq(u(t).diff(t) - u(t) - sin(t), 0)
sol = dsolve(dgl, u(t))
sol
```

[30]:
$$u(t) = C_1 e^{t} - \frac{\sin(t)}{2} - \frac{\cos(t)}{2}$$

[31]:
```
c1 = solve(sol.subs(t, t0), c1_s)[0]
c1
```

[31]:
$$\left(u(t_0) + \frac{\sqrt{2}\sin\left(t_0 + \frac{\pi}{4}\right)}{2}\right) e^{-t_0}$$

[ ]:
```
w = sol.subs(c1_s, c1)
w = w.rhs.subs(t0, 0)
w
```

1.3.3 Variation der Konstanten Formel

Die Lösung von

$$\frac{d}{dt}u(t) = au(t) + g(t, u(t)), \quad u(0) = u_0$$

ist

$$u(t) = \mathrm{e}^{at}u_0 + \int_0^t \mathrm{e}^{a(t-\tau)} g(\tau, u(\tau))\, d\tau$$

Für das Beispiel oben ist $a = 1$ und $g(t, u) = \sin(t)$. Das ergibt:

[ ]:
```
v = u(0) * exp(t) + integrate(exp(t - tau) * (sin(tau)), (tau, 0, t))
v
```

[ ]:
```
simplify(v - w)
```

1.3.4 Lösung mit einem Reihenansatz

$$y'(t) = y(t), \quad y(0) = 1$$

mit einem Reihenansatz

[ ]:
```
y0, t, C = symbols('y0 t C')
a = symbols('a:8')
y = Function('y')
```

[ ]:
```
dgl = Eq(y(t).diff(t), y(t))
dgl
```

[ ]:
```
ys = sum([a[i] * t**i for i in range(8)])
ys = ys.subs(a[0], 1)  # y(0) = 1
ys
```

[ ]:
```
gl = dgl.subs(y(t), ys).doit()
gl
```

[ ]:
```
gl.coeff(t) # das klappt so nicht
```

[ ]:
```
gls = gl.as_poly(t).all_coeffs()
gls
```

[ ]:
```
ac = solve(gls[1:])
ac
```

[ ]:
```
ac[a[0]] = 1  # wir raten das Bildungsgesetz der a's
acc = [ac[j] for j in a]
acc
```

[ ]:
```
[acc[j] / acc[j + 1] for j in range(7)]
```

[ ]:
```
[acc[j] - 1 / factorial(j) for j in range(8)]
```

[ ]:
```
n = symbols('n')
yr = Sum(t**n / factorial(n), (n, 0, oo))
yr
```

[ ]:
```
yr.doit()
```

1.3.5 Logistische Gleichung

einfaches Modell zur Beschreibung von Wachstum in einem Habitat mit beschränkten Resourcen

$$y'(t) = (1 - y(t))y(t)$$

[ ]:
```
y = Function('y')
t = symbols('t', real = True)
dgl = Eq(y(t).diff(t), (1-y(t))*y(t))
dgl
```

[ ]:
```
sol = dsolve(dgl, y(t))
sol
```

[ ]:
```
C1 = solve(sol, c1_s)
C1
```

[ ]:
```
C1 = solve(sol, c1_s).pop().subs(t, t0)
C1
```

[ ]:
```
sol.subs(c1_s, C1)
```