lektion14_i

January 28, 2026

Table of Contents

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
init_printing()
#%matplotlib qt
%matplotlib inline
plt.rcParams['figure.figsize'] = (5, 3)
```

1 Lektion 14

1.1 Extrema unter Nebenbedingungen

Problemstellung

Wir wollen für hinreichend oft differenzierbare reellwertige Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ und $g_j : \mathbb{R}^n \to \mathbb{R}$, $j = 1, \dots, m$, das Minimierungsproblem

$$\min_x f(x) \qquad s.d. \quad g_j(x) = 0 \text{ für } j = 1, \dots, m \qquad (*) \tag{1}$$

lösen.

Definition

x^* ist eine lokale Lösung von $(*)$, falls $g_j(x^*) = 0$ für $j = 1, \dots, m$ und, falls es $\varepsilon > 0$ gibt, mit

$$f(x^*) \leq f(x) \quad \text{für alle} \quad x \text{ mit } \|x - x^*\| < \varepsilon \text{ und } g_j(x) = 0, \quad j = 1, \dots, m.$$

1.1.1 Notwendige Bedingungen erster Ordnung (vgl. Analysis 2)

Definition (Lagrangefunktion) Für $\mu = (\mu_1, \dots, \mu_m) \in \mathbb{R}^m$ ist die Lagrangefunktion zu $(*)$:

$$\mathcal{L}(x, \mu) = f(x) - \sum_{j=1}^{m} \mu_j g_j(x)$$

(Unser Lagrangemultiplikatoren heißen μ weil es lambda in Python schon gibt.)

In der Analysis 2 wird folgender Satz bewiesen:

Satz (Notwendige Bedingungen erster Ordnung)

Ist x^* eine lokale Lösung von $(*)$ und ist die Menge $\{\nabla_x g_j(x^*), j = 1, \dots, m\}$ linear unabhängig, dann gibt es $\mu_j^* \in \mathbb{R}$, $j = 1 \dots, m$ so, dass

$$\nabla_x \mathcal{L}(x^*, \mu^*) = 0, \tag{2}$$
$$g_j(x^*) = 0, \ \forall\ j = 1, \dots, m. \tag{3}$$

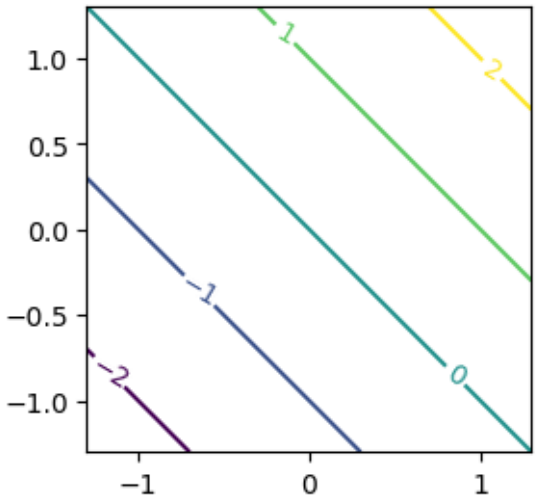
äquivalente kompakte Formulierung

$$\nabla \mathcal{L}(x^*, \mu^*) = 0$$

1.1.2 Beispiel (n=2, m=1)

[2]:
```
x = symbols('x:2', real=True)
mu = symbols('mu:1', real=True) # Für den Fall m=1 ginge das einfacher.

f = x[0] + x[1]
g = Matrix(1, 1, [x[0]**4 + x[1]**4 - 1])

fn = lambdify(x, f)
gn = lambdify(x, g)
```

[3]:
```
x0, x1 = np.linspace(-1.3, 1.3, 100), np.linspace(-1.3, 1.3, 100)
X0, X1 = np.meshgrid(x0, x1)
fig, ax = plt.subplots()
pf = ax.contour(X0, X1, fn(X0, X1), np.linspace(-2, 2, 5))
plt.clabel(pf)
ax.set_aspect('equal')
```



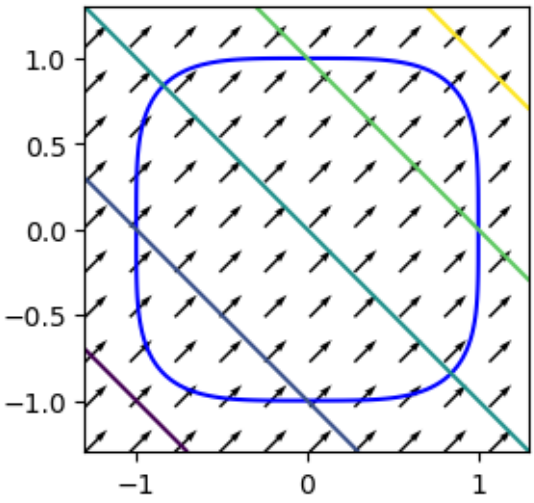
[4]:
```
pg = ax.contour(X0, X1, gn(X0, X1)[0,0,:,:], [0], colors ='blue')
```

[5]:
```
grad_g = g.jacobian(x).T
grad_f = Matrix([f]).jacobian(x).T
grad_g, grad_f
```

[5]:
$$\left(\begin{bmatrix}4x_0^3\\4x_1^3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right)$$

[6]:
```
grad_gn = lambdify(x, grad_g)
grad_fn = lambdify(x, grad_f)
```

[7]:
```
fig, ax = plt.subplots()
pg = ax.contour(X0, X1, gn(X0, X1)[0, 0, :, :], 0, colors='blue')
ax.contour(X0, X1, fn(X0, X1), np.arange(-2, 3))
GF = grad_fn(X0[::10, ::10], X1[::10, ::10])
ax.quiver(X0[::10, ::10], X1[::10, ::10], GF[0], GF[1], angles='xy', scale=20)
ax.set_aspect('equal')
```



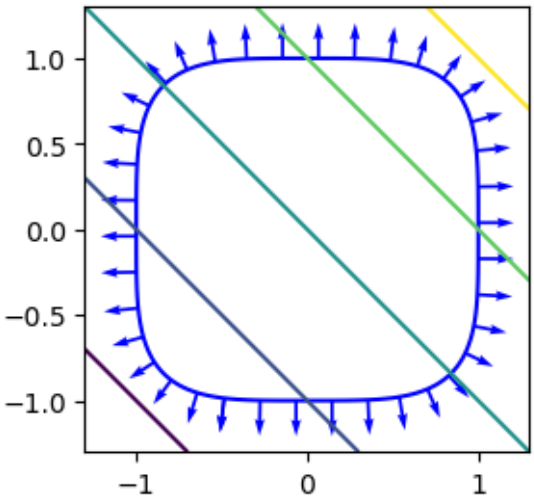
[8]:
```
# 305 Punkte auf der g(x)=0 Linie
pg.allsegs[1][0].shape
```

[8]:
(305, 2)

[9]:
```
# Punkte auf der g(x)=0 Linie
SG = np.array(pg.allsegs[1][0])
np.allclose(SG[:,0]**4 + SG[:,1]**4, 1, atol=1e-3, rtol=1e-3)
```

[9]:
True

[10]:
```
fig = plt.figure()
ax = fig.gca()
pg = ax.contour(X0, X1, gn(X0, X1)[0, 0, :, :], 0, colors='blue')
ax.contour(X0, X1, fn(X0, X1), [-2, -1, 0, 1, 2])
GG = grad_gn(SG[::8, 0], SG[::8, 1])
ax.quiver(SG[::8, 0], SG[::8, 1], GG[0],  GG[1],
          angles='xy', scale=50, color='blue')
ax.set_aspect('equal')
```



[11]:
```
# Gradient der Lagrangefunktion bzgl. x
GxL = grad_f - mu[0] * grad_g
# hier steht eigentlich das Skalarprodukt von mu und grad_g
GxL
```

[11]:
$$\begin{bmatrix} -4\mu_0 x_0^3 + 1 \\ -4\mu_0 x_1^3 + 1 \end{bmatrix}$$

[12]:
```
# Notwendige Bedingungen erster Ordnung
eqn = [Eq(_, 0) for _ in GxL]
eqn.append(Eq(g[0], 0))
eqn
```

[12]:
$$\left[-4\mu_0 x_0^3 + 1 = 0,\ -4\mu_0 x_1^3 + 1 = 0,\ x_0^4 + x_1^4 - 1 = 0\right]$$

[13]:
```
M = nonlinsolve(eqn, (*x, *mu))
Mr = [l for l in M
     if all(l_.is_real for l_ in l)]  #hier suchen wir die reellen Lösungen
Mr
```

[13]:
$$\left[\left(-\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{4}\right), \left(\frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{4}\right)\right]$$

[14]:
```
# Alternative zum raussuchen der reellen Lösungen
Mr = []
for sol in M:
    if (im(sol[0]) == 0) & (im(sol[1]) == 0) & (im(sol[2]) == 0):
        Mr.append(sol)
Mr
```

[14]:

$$\left[\left(-\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{4}\right), \left(\frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{4}\right)\right]$$

[15]:
```
M = solve(eqn, (*x, *mu)) # solve respektiert dass x und mu reelle Symbole sind
M
```

[15]:
$$\left[\left(-\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{2}, -\frac{2^{\frac{3}{4}}}{4}\right), \left(\frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{2}, \frac{2^{\frac{3}{4}}}{4}\right)\right]$$

[16]:
```
# Notwendige Bedingungen erster Ordnung kompakte Formulierung
L = f - (mu[0] * g[:, 0])[0]
GL = Matrix([L]).jacobian((*x, *mu))

eqn = [Eq(_, 0) for _ in GL]
eqn
```

[16]:
$$[-4\mu_0 x_0^3 + 1 = 0,\ -4\mu_0 x_1^3 + 1 = 0,\ -x_0^4 - x_1^4 + 1 = 0]$$

[17]:
```
M = solve(eqn)
M
```

[17]:
$$\left[\left\{\mu_0 : -\frac{2^{\frac{3}{4}}}{4},\ x_0 : -\frac{2^{\frac{3}{4}}}{2},\ x_1 : -\frac{2^{\frac{3}{4}}}{2}\right\}, \left\{\mu_0 : \frac{2^{\frac{3}{4}}}{4},\ x_0 : \frac{2^{\frac{3}{4}}}{2},\ x_1 : \frac{2^{\frac{3}{4}}}{2}\right\}\right]$$

[18]:
```
[g[0].subs(_) for _ in M] # Probe
```

[18]:
$$[0,\ 0]$$

In der Optimierung werden folgende Sätze bewiesen:

1.1.3 Notwendige Bedingungen zweiter Ordnung

Satz (Zweite Ordnung notwendige Bedingungen)

Ist x^* eine lokale Lösung von $(*)$ und ist die Menge $\{\nabla_x g_j(x^*), j = 1, \dots, m\}$ linear unabhängig, dann gilt

$$s^T \nabla_{xx} \mathcal{L}(x^*, \mu^*) s \geq 0 \quad \forall s \in \{s \in \mathbb{R}^n : s^T \nabla_x g_j(x^*) = 0,\ \forall j = 1, \dots, m\}.$$

1.1.4 Hinreichende Bedingungen zweiter Ordnung

Satz (Zweite Ordnung hinreichende Bedingungen)

Gibt es für x^* mit $g_j(x^*) = 0$ für $j = 1, \dots, m$ Lagrangemultiplikatoren $\mu_j^* \in \mathbb{R}$, $j = 1, \dots, m$ so, dass $\nabla_x \mathcal{L}(x^*, \mu^*) = 0$ und ist

$$s^T \nabla_{xx} \mathcal{L}(x^*, \mu^*) s > 0, \quad \forall s \in \{s : s^T \nabla_x g_j(x^*) = 0\ \forall j = 1, \dots, m, s \neq 0\},$$

so ist x^* eine lokale Lösung von $(*)$.

[19]:
```
H = hessian(L, x)
```

[20]:
```
H.subs(M[0])  # diese Matrix ist positiv definit (das sieht man)
```

[20]:
$$\begin{bmatrix} 3\sqrt[4]{2} & 0 \\ 0 & 3\sqrt[4]{2} \end{bmatrix}$$

Also haben wir hier ein Minimum (Achtung: Für das Minimum würde es schon reichen, dass nur ein Teil der Hessematrix positiv definit ist, siehe obiger Satz)

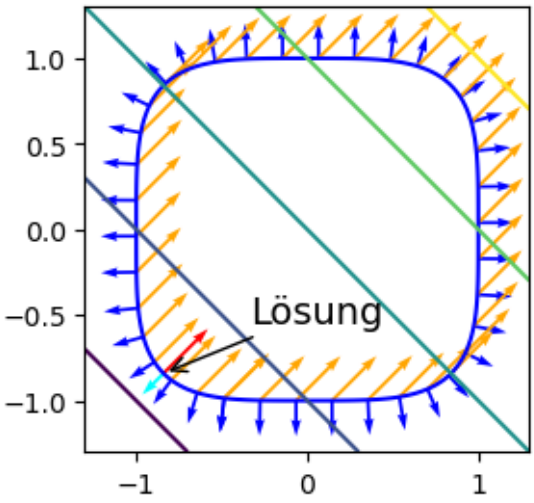
Wir berechnen eine Basis von $span\{s : s^T \nabla_x g_j(x^*) = 0 \ \forall j = 1, \dots, m\}$

[21]:
```
bkgg = g.jacobian(x).subs(M[0]).nullspace()
P = bkgg[0] # und bilden eine Matrix aus diesen Basisvektoren
P.T*H.subs(M[0])*P # laut obigem Satz muss nur diese Matrix posistiv definit␣
 ↪sein
```

[21]:
$$\begin{bmatrix} 6\sqrt[4]{2} \end{bmatrix}$$

[22]:
```
xx0 = float(M[0][x[0]])
xx1 = float(M[0][x[1]])
```

[23]:
```
fig = plt.figure()
ax = fig.gca()
pg = ax.contour(X0, X1, gn(X0, X1)[0, 0], 0, colors='blue')

ax.contour(X0, X1, fn(X0, X1), [-2, -1, 0, 1, 2])

GG = grad_gn(SG[::8, 0], SG[::8, 1])
ax.quiver(SG[::8, 0], SG[::8, 1], GG[0], GG[1],\
          angles='xy', scale=50, color='blue')

GF = grad_fn(SG[::8, 0], SG[::8, 1])
ax.quiver(SG[::8, 0], SG[::8, 1], GF[0], GF[1],\
          angles='xy', scale=10, color='orange')

ax.quiver(xx0, xx1, grad_gn(xx0, xx1)[0], grad_gn(xx0, xx1)[1],\
          angles='xy', scale=50, color='cyan')
ax.quiver(xx0, xx1, grad_fn(xx0, xx1)[0], grad_fn(xx0, xx1)[1],\
          angles='xy', scale=10, color='red')

ax.set_aspect('equal')
ax.annotate('Lösung', (xx0, xx1), (xx0+.5, xx1+.3),\
            arrowprops={'arrowstyle':'->'}, fontsize=14);
```



1.1.5 Beispiel (n=3, m=2)

[24]:
```
x = symbols('x:3', real=True)
mu = symbols('mu:2', real=True)

f = x[0]*x[1]*x[2]
g = Matrix(2, 1, [x[0]**2 + x[2]**2 - 1, \
                  2*x[0]**2 + x[1]**2 - 2])

fn = lambdify(x, f)
gn = lambdify(x, g)
f, g
```

[24]:
$$\left(x_0 x_1 x_2,\ \begin{bmatrix} x_0^2 + x_2^2 - 1 \\ 2x_0^2 + x_1^2 - 2 \end{bmatrix}\right)$$

[25]:
```
L = Matrix([f]) - Matrix(1, 2, [*mu]) *  g
GL = Matrix([L]).jacobian((*x, *mu))

eqn = [Eq(_, 0) for _ in GL]
eqn
```

[25]:
$$[-2\mu_0 x_0 - 4\mu_1 x_0 + x_1 x_2 = 0,\ -2\mu_1 x_1 + x_0 x_2 = 0,\ -2\mu_0 x_2 + x_0 x_1 = 0,\ -x_0^2 - x_2^2 + 1 = 0,\ -2x_0^2 - x_1^2 + 2 = 0]$$

[26]:
```
sols = solve(eqn)
sols
```

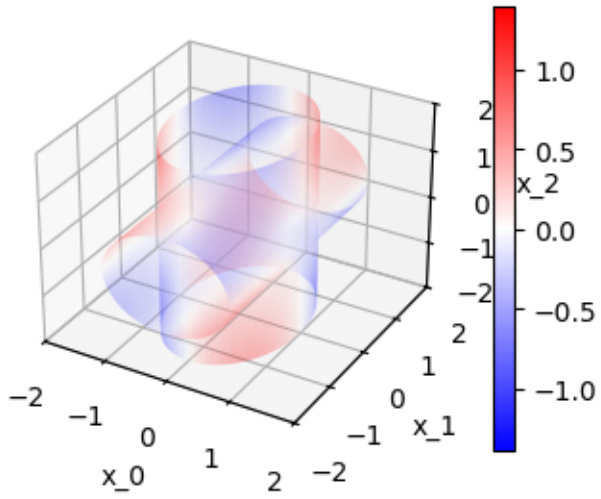
[26]:
$$\left[\{\mu_0 : -2\mu_1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ x_0 : 1,\ x_1 : 0,\ x_2 : 0\},\ \{\right.$$

[27]:
```
for i in range(4):
    sols[i] |= {mu[1]:1}
sols
```

[27]:
$$\left[\{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0\right.$$

[28]:
```
from skimage import measure

xn = np.linspace(-2, 2, 51)
dx = xn[1] - xn[0]
X = np.meshgrid(xn, xn, xn)
G0 = gn(*X)[0, 0]
G1 = gn(*X)[1, 0]
p0, t0, n, v = measure.marching_cubes(G0, level=0, spacing=(dx, dx, dx))
p1, t1, n, v = measure.marching_cubes(G1, level=0, spacing=(dx, dx, dx))
p0 -= 2
p1 -= 2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
pg0 = ax.plot_trisurf(p0[:, 1], p0[:, 0], p0[:, 2],
                      triangles=t0, color='orange', alpha=0.3)
pg1 = ax.plot_trisurf(p1[:, 1], p1[:, 0], p1[:, 2],
                      triangles=t1, color='purple',
                      alpha=0.3)
ax.set_xlabel('x_0')
ax.set_ylabel('x_1')
ax.set_zlabel('x_2')
ax.set_xlim((-2, 2))
ax.set_ylim((-2, 2))
ax.set_zlim((-2, 2))


# Source - https://stackoverflow.com/a/63325673
# modified

from matplotlib.colors import Normalize
import matplotlib.cm as cm

def clrs(pg, f):
    """
    pg: a Poly3DCollection, as returned e.g. by ax.plot_trisurf
    f: a single-valued function of 3 arrays: x, y, z
    """
    # reconstruct the triangles from internal data
    x, y, z, _ = pg._vec
```

```
    slices = pg._segslices
    triangles = np.array([np.array((x[s], y[s], z[s])).T for s in slices])
    # compute the barycentres for each triangle
    xb, yb, zb = triangles.mean(axis=1).T

    # compute the function in the barycentres
    return f(xb, yb, zb)


vg0 = clrs(pg0, fn)
vg1 = clrs(pg1, fn)
vmin = min(min(vg0), min(vg1))
vmax = max(max(vg0), max(vg1))

# usual stuff
norm = Normalize(vmin=vmin, vmax=vmax)
colors_g0 = plt.get_cmap('bwr')(norm(vg0))
colors_g1 = plt.get_cmap('bwr')(norm(vg1))
# set the face colors of the Poly3DCollection
pg0.set_fc(colors_g0)
pg1.set_fc(colors_g1)
fig.colorbar(cm.ScalarMappable(norm=norm, cmap='bwr'), ax=ax);
```



[29]:
```
grad_g = g.jacobian(x)
grad_f = Matrix([f]).jacobian(x)
grad_g, grad_f
```

[29]:
$$\left(\begin{bmatrix} 2x_0 & 0 & 2x_2 \\ 4x_0 & 2x_1 & 0 \end{bmatrix}, \begin{bmatrix} x_1x_2 & x_0x_2 & x_0x_1 \end{bmatrix}\right)$$

[30]:
```
grad_gn = lambdify(x, grad_g)
grad_fn = lambdify(x, grad_f)

# Gradient der Lagrangefunktion bzgl. x
GxL = grad_f +  Matrix(1, 2, [*mu]) * grad_g

# Notwendige Bedingungen erster Ordnung
eqn = [Eq(_, 0) for _ in GxL]
eqn.append(Eq(g[0], 0))
eqn.append(Eq(g[1], 0))
eqn
```

[30]:
$$[2\mu_0 x_0 + 4\mu_1 x_0 + x_1 x_2 = 0,\ 2\mu_1 x_1 + x_0 x_2 = 0,\ 2\mu_0 x_2 + x_0 x_1 = 0,\ x_0^2 + x_2^2 - 1 = 0,\ 2x_0^2 + x_1^2 - 2 = 0]$$

[31]:
```
#sols = nonlinsolve(eqn, (*x, *mu))
#solsr = [sol for sol in sols
#     if all(l_.is_real for l_ in sol)]  # hier suchen wir die reellen Lösungen
#solsr
```

[32]:
```
# Alternativ:
# Notwendige Bedingungen erster Ordnung kompakte Formulierung
L = Matrix([f]) + Matrix(1, 2, [*mu]) *  g
GL = Matrix([L]).jacobian((*x, *mu))

eqn = [Eq(_, 0) for _ in GL]
eqn
```

[32]:
$$[2\mu_0 x_0 + 4\mu_1 x_0 + x_1 x_2 = 0,\ 2\mu_1 x_1 + x_0 x_2 = 0,\ 2\mu_0 x_2 + x_0 x_1 = 0,\ x_0^2 + x_2^2 - 1 = 0,\ 2x_0^2 + x_1^2 - 2 = 0]$$

[33]:
```
sols = solve(eqn)
sols
for i in range(4):
    sols[i] |= {mu[1]:1}
sols
```

[33]:
$$\left[\{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0 : -1,\ x_1 : 0,\ x_2 : 0\},\ \{\mu_0 : -2\mu_1,\ \mu_1 : 1,\ x_0\right.$$

[34]:
```
H = hessian(L, x)
```

[35]:
```
[H.subs(sols[i]).n() for i in range(len(sols))]
```

[35]:
$$\left[\begin{bmatrix}0 & 0 & 0\\ 0 & 2.0 & -1.0\\ 0 & -1.0 & -4.0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 2.0 & -1.0\\ 0 & -1.0 & -4.0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 2.0 & 1.0\\ 0 & 1.0 & -4.0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 2.0 & 1.0\\ 0 & 1.0 & -4.0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0\\ 0 & 0.707106781186548 & -1\\ 0 & -1.0 & -1.414213\end{bmatrix}\right.$$

Wir berechnen eine Basis von $span\{s : s^T \nabla_x g_j(x^*) = 0\ \forall j = 1, \dots, m\}$

[36]:
```
for sol in sols:
    bkgg = g.jacobian(x).subs(sol).nullspace()
    P = Matrix([v.T for v in bkgg]).T  # und bilden eine Matrix aus diesen␣
 ↪Basisvektoren
    display([ev.n() for ev in (P.T*H.subs(sol)*P).eigenvals().keys()]) # laut␣
 ↪obigem Satz muss nur diese Matrix posistiv definit sein
```

$[-4.16227766016838, 2.16227766016838]$

$[-4.16227766016838, 2.16227766016838]$

$[-4.16227766016838, 2.16227766016838]$

$[-4.16227766016838, 2.16227766016838]$

$[-1.8112913643046, 1.10418458311805]$

$[-1.10418458311805, 1.8112913643046]$

$[-9.79795897113271]$

$[-9.79795897113271]$

$[-9.79795897113271]$

$[-9.79795897113271]$

$[9.79795897113271]$

$[9.79795897113271]$

$[9.79795897113271]$

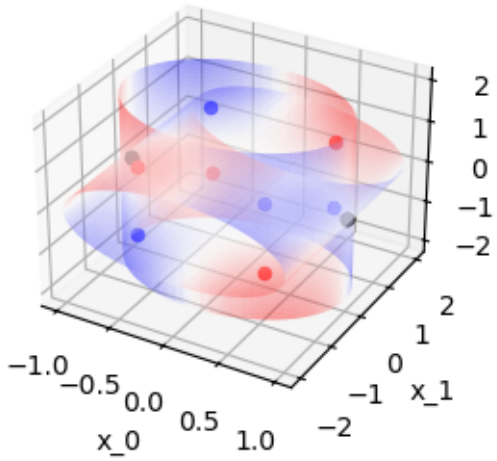
$[9.79795897113271]$

[37]:
```
TestM = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])
NM = TestM.nullspace()
NM, Matrix([v.T for v in NM]).T
```

[37]:
$$\left(\left[\begin{bmatrix}-15\\6\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}1\\-\frac{1}{2}\\0\\0\\1\end{bmatrix}\right], \begin{bmatrix}-15 & 0 & 1\\6 & 0 & -\frac{1}{2}\\1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\right)$$

[38]:
```
xn = np.linspace(-2, 2, 51)
dx = xn[1] - xn[0]
X = np.meshgrid(xn, xn, xn)
G0 = gn(*X)[0, 0]
G1 = gn(*X)[1, 0]
p0, t0, n, v = measure.marching_cubes(G0, level=0, spacing=(dx, dx, dx))
p1, t1, n, v = measure.marching_cubes(G1, level=0, spacing=(dx, dx, dx))
p0 += xn[0]
p1 += xn[0]
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
pg0 = ax.plot_trisurf(p0[:, 1], p0[:, 0], p0[:, 2],
                      triangles=t0, color='orange', alpha=0.4)
pg1 = ax.plot_trisurf(p1[:, 1], p1[:, 0], p1[:, 2],
                      triangles=t1, color='purple', alpha=0.4)

vg0 = clrs(pg0, fn)
vg1 = clrs(pg1, fn)
vmin = min(min(vg0), min(vg1))
vmax = max(max(vg0), max(vg1))

# usual stuff
norm = Normalize(vmin=vmin, vmax=vmax)
colors_g0 = plt.get_cmap('bwr')(norm(vg0))
colors_g1 = plt.get_cmap('bwr')(norm(vg1))
# set the face colors of the Poly3DCollection
pg0.set_fc(colors_g0)
pg1.set_fc(colors_g1)

ax.set_xlabel('x_0')
ax.set_ylabel('x_1')
ax.set_zlabel('x_2')

for sol in sols:
    xx = [float(sol[x_]) for x_ in x]
    bkgg = g.jacobian(x).subs(sol).nullspace()
    P = Matrix([v.T for v in bkgg]).T
    if all(
        [ev.n() > 0
         for ev in (P.T * H.subs(sol) * P).eigenvals().keys()]):
        ax.scatter(*xx, c='b')
    elif all(
        [ev.n() < 0
         for ev in (P.T * H.subs(sol) * P).eigenvals().keys()]):
        ax.scatter(*xx, c='r')
    else:
        ax.scatter(*xx, c='k')
```


2
1
0
−1
−2
−1.0
−0.5
0.0
0.5
1.0
x_0
2
1
0
−1
−2
x_1

[ ]: