notebook_3_f

June 17, 2025

1 Notebook 3 - Finite Differenzen und schnelle Löser

Namen: Friedward Wenzler, Yueheng Li —> Erreichte Punktzahl: 9,5/10

Erweitern Sie Ihre conda Umgebung `python-science` um die Paket `pytest` und `ipytest`. Aktivieren Sie dafür die virtuelle Umgebung `python-science` und laden Sie die Pakete mit:

```
conda install -c conda-forge pytest ipytest
```

Starten Sie anschließend Ihre jupyter-Instanz neu. Schließen Sie dazu alle offenen Fenster, beenden Sie die `jupyter-ServerApp` (z.B. mit `Strg+C`) und starten Sie anschließend Jupyter-Lab erneut.

Nun sollten die folgenden imports ohne Fehlermeldung ausgeführt werden.

[199]:
```
!pip install pytest
!pip install ipytest
```

```
Requirement already satisfied: pytest in /usr/local/lib/python3.11/dist-packages
(8.3.5)
Requirement already satisfied: iniconfig in /usr/local/lib/python3.11/dist-
packages (from pytest) (2.1.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-
packages (from pytest) (24.2)
Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.11/dist-
packages (from pytest) (1.6.0)
Requirement already satisfied: ipytest in /usr/local/lib/python3.11/dist-
packages (0.14.2)
Requirement already satisfied: ipython in /usr/local/lib/python3.11/dist-
packages (from ipytest) (7.34.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-
packages (from ipytest) (24.2)
Requirement already satisfied: pytest>=5.4 in /usr/local/lib/python3.11/dist-
packages (from ipytest) (8.3.5)
Requirement already satisfied: iniconfig in /usr/local/lib/python3.11/dist-
packages (from pytest>=5.4->ipytest) (2.1.0)
Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.11/dist-
packages (from pytest>=5.4->ipytest) (1.6.0)
Requirement already satisfied: setuptools>=18.5 in
/usr/local/lib/python3.11/dist-packages (from ipython->ipytest) (75.2.0)
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.11/dist-
```

```
packages (from ipython->ipytest) (0.19.2)
Requirement already satisfied: decorator in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (4.4.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (0.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in
/usr/local/lib/python3.11/dist-packages (from ipython->ipytest) (3.0.51)
Requirement already satisfied: pygments in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (2.19.1)
Requirement already satisfied: backcall in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (0.2.0)
Requirement already satisfied: matplotlib-inline in
/usr/local/lib/python3.11/dist-packages (from ipython->ipytest) (0.1.7)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.11/dist-
packages (from ipython->ipytest) (4.9.0)
Requirement already satisfied: parso<0.9.0,>=0.8.4 in
/usr/local/lib/python3.11/dist-packages (from jedi>=0.16->ipython->ipytest)
(0.8.4)
Requirement already satisfied: ptyprocess>=0.5 in
/usr/local/lib/python3.11/dist-packages (from pexpect>4.3->ipython->ipytest)
(0.7.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.11/dist-
packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->ipython->ipytest)
(0.2.13)
```

[200]:
```python
# code zelle mit import
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import time
import scipy
import scipy.fft
import scipy.sparse
import sympy as sp
import ipytest
ipytest.autoconfig()
from matplotlib import rcParams
rcParams.update({'figure.autolayout': True})
```

Machen Sie sich mit der Dokumentation von `pytest` vertraut. Um pytest innerhalb eines Jupyter-Notebooks zu verwenden, müssen wir einen Umweg über ipytest gehen. Jede Zelle, die einen Test enthalten soll, muss zu Beginn das magic-command `%%ipytest` enthalten. Das zusätzliche Argument `-vv` führt zu einer detailierteren Ausgabe der Tests.

Jeder Test wird als Funktion definiert und startet mit `test_`. Im Verlauf des Notebooks könnten Funktionen aus `numpy.testing` nützlich werden. Machen Sie sich auch mit dieser Dokumentation

vertraut. Im Folgenden sind zwei Beispielanwendungen von pytest gegeben.

[201]:
```
%%ipytest -vv
def test_addition():
    assert 1 + 1 == 2
def test_2():
    np.testing.assert_allclose(np.array([1.0, 1.0]), np.array([1.0 - 1e-16,␣
 ↪1]), atol=1e-14)
```

```
========================================= test session starts
==========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 2 items

t_11c50f6b742b471cab6a388de5e8291e.py::test_addition PASSED
[ 50%]
t_11c50f6b742b471cab6a388de5e8291e.py::test_2 PASSED
[100%]

========================================== 2 passed in
0.02s ==========================================
```

Ein fehlgeschlagener Test würde die folgende Rückgabe liefern:

[202]:
```
%%ipytest -vv
def test_addition():
    assert 1 + 1 == 3
```

```
========================================= test session starts
==========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_addition FAILED
[100%]

============================================== FAILURES
==========================================
```

```
_____________________________________ test_addition
_____________________________________

    def
test_addition():
>       assert 1 + 1 ==
3
E       assert (1 + 1) == 3

<ipython-input-202-4154234050>:2: AssertionError
=================================== short test summary info
=====================================
FAILED t_11c50f6b742b471cab6a388de5e8291e.py::test_addition -
assert (1 + 1) == 3
====================================== 1 failed in
0.04s ======================================
```

5/5

2 Aufgabe 1 - Finite Differenzen mit ortsabhängigem Materialkoeffizienten

In dieser Aufgabe ist es unser Ziel, das Randwertproblem

$$\begin{aligned} -\partial_x(D(x)\partial_x u(x)) &= f(x), & x &\in (0,1), \\ u(x) &= 0, & x &\in \{0,1\}, \end{aligned}$$

aus Abschnitt 14.2 numerisch mit dem Verfahren der Finiten-Differenzen zu lösen. Dabei wollen wir für D glatte wie auch nicht glatte Beispiele testen.

2.1 a) Assemblierung der 1D Finite-Differenzen Matrizen

Vervollstädigen Sie die unten aufgeführte Funktion zur Assemblierung der Finite-Differenzen Matrix, welche durch die Diskretisierung (14.31) entsteht.

[203]:
```python
# Darstellung (14.31)
def assemble_Lh_nonsym(a, b, N, D = None):
    # assume D = None is D = I
    if D is None:
      D = lambda x: x
    h = (b-a)/N
    x = np.linspace(a, b, N+1)
    D_x = D(x)
    # berechne diagonale und nebendiagonale
    n = N - 1
    diag = (1 / h**2) * 2 * D_x[1:-1]
    upper_diag = - (1 / h**2) * (D_x[1: -2] + 0.25 * (D_x[2: -1] - D_x[0: -3]))
    lower_diag = - (1 / h**2) * (D_x[2: -1] - 0.25 * (D_x[3:] - D_x[1: -2]))
```

```
    return scipy.sparse.diags_array([upper_diag, diag, lower_diag],
                                    offsets=[1, 0, -1],
                                    format="csc")

assemble_Lh_nonsym(0, 1, 5).toarray()
```

[203]:
```
array([[ 10. ,  -7.5,   0. ,   0. ],
       [ -7.5,  20. , -12.5,   0. ],
       [  0. , -12.5,  30. , -17.5],
       [  0. ,   0. , -17.5,  40. ]])
```

Vervollständigen Sie auch die folgende Funktion zur Assemblierung der Finite-Differenzen Matrix, welche mittels (14.35) entsteht.

[204]:
```
# Darstellung (14.36)
def assemble_Lh_sym(a, b, N, D=None):
    # assume D = None is D = I
    if D is None:
      D = lambda x: x
    h = (b-a)/N
    x = np.linspace(a, b, N + 1)
    # verschobenes gitter
    x_minus_half = x[1: -1] - 0.5 * h
    x_plus_half = x[1: -1] + 0.5 * h
    D_x_minus_half = D(x_minus_half)
    D_x_plus_half = D(x_plus_half)
    # berechne diagonale und nebendiagonale
    diag = (1 / h**2) * (D_x_plus_half + D_x_minus_half)
    diag_upper = - (1 / h**2) * D_x_plus_half[:-1]
    diag_lower = - (1 / h**2) * D_x_minus_half[1:]
    return scipy.sparse.diags_array([diag_upper, diag, diag_lower],
                                    offsets=[1, 0, -1],
                                    format="csc")

assemble_Lh_sym(0, 1, 5).toarray()
```

[204]:
```
array([[ 10. ,  -7.5,   0. ,   0. ],
       [ -7.5,  20. , -12.5,   0. ],
       [  0. , -12.5,  30. , -17.5],
       [  0. ,   0. , -17.5,  40. ]])
```

2.2 b) Konstanter Koeffizient D

Testen Sie ihre beiden Assemblierungen für den einfachsten Fall $D(x) = d_0 > 0$, indem Sie sich eine sinnvolle nicht-triviale Funktion u als Lösung von

$$\begin{aligned} -\partial_x(D(x)\partial_x u(x)) &= f(x), & x &\in (0,1), \\ u(x) &= 0, & x &\in \{0,1\}, \end{aligned}$$

vorgeben und damit eine rechte Seite f berechnen. Erzeugen Sie damit einen Konvergenzplot.

Überlegen Sie sich außerdem einen weiteren Test. Was würde man im Fall konstanter Koeffizienzen von den beiden Assemblierungen erwarten?

Beide Assemblierungen sollen bei konstanten Koeffizienten geich sein

[205]:
```
a = 0
b = 1
N = 100
d0 = 5.
```

[206]:
```
D_const = lambda x: np.full_like(x, d0)
```

[207]:
```
%%ipytest -vv

"""
Constant D implies same matrix for both the symmetric and non symmetric␣
 ↪assemblies
"""
def test_constant_same():
  Lh_nonsym = assemble_Lh_nonsym(a, b, N, D_const).toarray()
  Lh_sym = assemble_Lh_sym(a, b, N, D_const).toarray()
  np.testing.assert_array_equal(Lh_nonsym, Lh_sym)
```

```
========================================= test session starts
==========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_constant_same PASSED
[100%]

========================================== 1 passed in
0.03s ==========================================
```

[208]:
```
"""x = sp.symbols('x')
u_sp = ...
u = sp.lambdify(..., "numpy")
f_sp = ...
f = sp.lambdify(..., "numpy")"""

import sympy as sp
from functools import partial
```

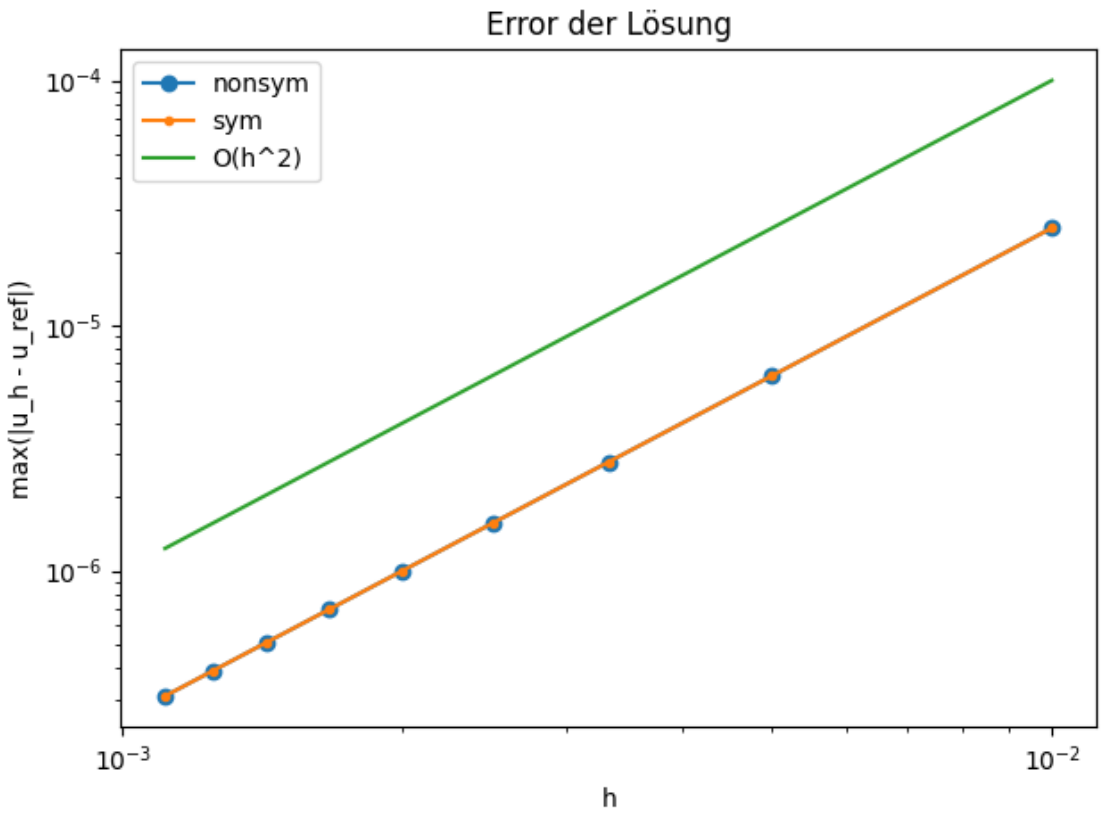
```
h_f = lambda a, b, n_min, n_max, s: [(b - a) / n for n in range(n_min, n_max,␣
 ↪s)]

# Lösung
u = lambda x: x**3 * (1 - x)

# f
x = sp.symbols('x')
u_sym = u(x)
du_dx = sp.diff(u_sym, x)
f_sym = -1 * sp.diff(d0 * du_dx, x)
f_full = sp.lambdify((x), f_sym, "numpy")
```

[209]:
```
%%ipytest -vv

def test_sym_order():
  N = 1000
  x = np.linspace(a, b, N+1)
  h = (b - a) / N
  f_x = f_full(x)[1:-1]   # only inner x
  Lh_sym = assemble_Lh_sym(a, b, N, D_const).toarray()
  u_x = u(x)[1:-1]  # only inner x
  np.testing.assert_allclose(Lh_sym @ u_x, f_x, atol=11 * h**2, rtol=0)

def test_nonsym_order():
  N = 1000
  x = np.linspace(a, b, N+1)
  h = (b - a) / N
  f_x = f_full(x)[1:-1]   # only inner x
  Lh_nonsym = assemble_Lh_nonsym(a, b, N, D_const).toarray()
  u_x = u(x)[1:-1]  # only inner x
  np.testing.assert_allclose(Lh_nonsym @ u_x, f_x, atol=11 * h**2, rtol=0)
```

```
========================================= test session starts
=========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 2 items

t_11c50f6b742b471cab6a388de5e8291e.py::test_sym_order PASSED
[ 50%]
t_11c50f6b742b471cab6a388de5e8291e.py::test_nonsym_order PASSED
[100%]
```

```
========================================== 2 passed in
0.03s ==========================================
```

[210]:
```python
# Konvergenzplot

N_min = 100
N_max = 1000
s = 100

def plot_u_assemblies_error(a, b, N_min, N_max, s, u, f, D):
  e_nonsym = np.empty((N_max - N_min) // s)
  e_sym = np.empty((N_max - N_min) // s)
  for i, n in enumerate(range(N_min, N_max, s)):
    x = np.linspace(a, b, n+1)
    f_x = f(x)[1:-1]   # only inner x
    Lh_nonsym = assemble_Lh_nonsym(a, b, n, D).toarray()
    Lh_sym = assemble_Lh_sym(a, b, n, D).toarray()
    u_nonsym = scipy.linalg.solve(Lh_nonsym, f_x)
    u_sym = scipy.linalg.solve(Lh_sym, f_x)
    u_ref = u(x)[1:-1]  # only inner x
    e_nonsym[i] = np.max(np.abs(u_nonsym - u_ref))
    e_sym[i] = np.max(np.abs(u_sym - u_ref))

  h = h_f(a, b, N_min, N_max, s)
  plt.plot(h, e_nonsym, marker="o", linestyle="-", label="nonsym")
  plt.plot(h, e_sym, marker=".", linestyle="-", label="sym")
  plt.plot(h, np.array(h)**2, label="O(h^2)")
  plt.xscale("log")
  plt.yscale("log")
  plt.xlabel("h")
  plt.ylabel("max(|u_h - u_ref|)")
  plt.title("Error der Lösung")
  plt.legend()
  plt.show()

plot_u_assemblies_error(a, b, N_min, N_max, s, u, f_full, D_const)
```



2.3 c) Nicht-konstanter glatter Koeffizient D

Testen Sie ihre beiden Assemblierungen erneut gegen eine sinnvolle nicht-triviale exakte Lösung u. Diesmal für den Fall $D(x) = 1.5 + \mathrm{e}^{\sin(4x)}$.

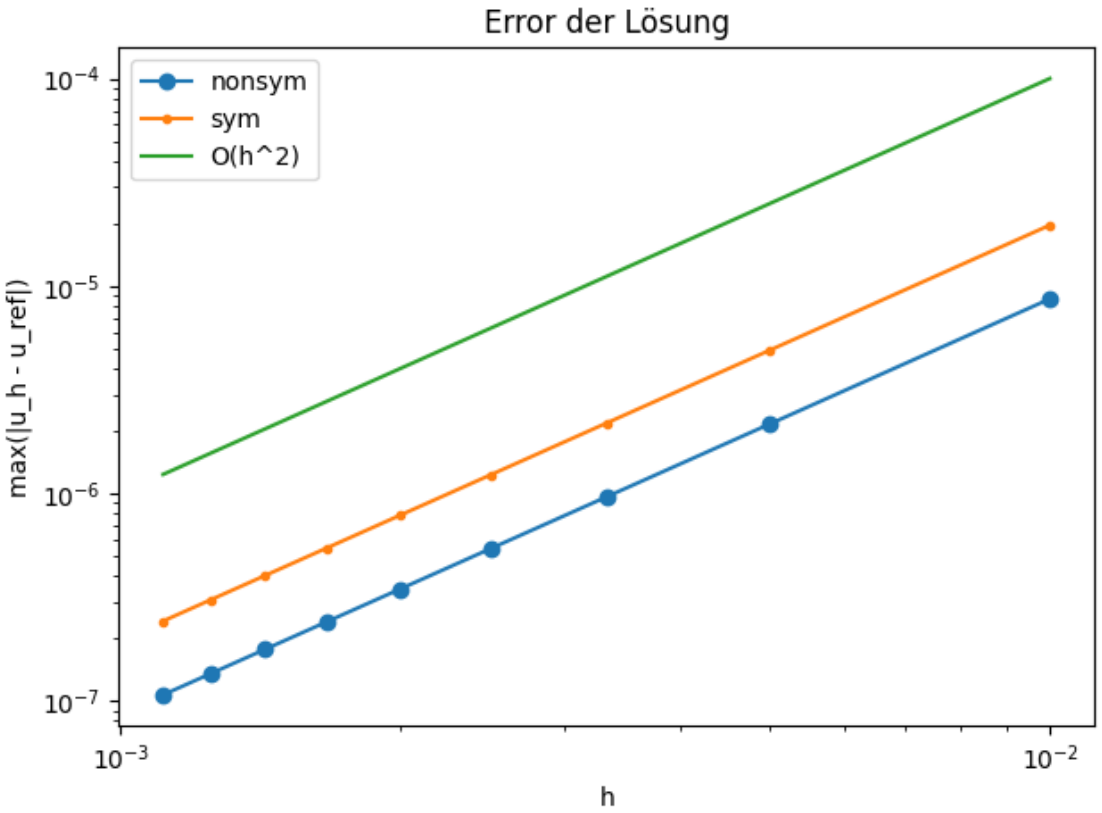
Was passiert in diesem Fall mit den beiden Assemblierungen?

[211]:
```
N = 100
a = 0
b = 1
```

[212]:
```
x = sp.symbols('x')

D_sp = 1.5 + sp.exp(sp.sin(4 * x))
u_sp = x**3 * (1 - x)

f_sp = -1 * sp.diff(D_sp * sp.diff(u_sp, x), x)

u = sp.lambdify(x, u_sp, "numpy")
D = sp.lambdify(x, D_sp, "numpy")
f = sp.lambdify(x, f_sp, "numpy")
```

[213]:
```
xh = np.linspace(a, b, N + 1)
fh = f(xh)[1:-1]
Lh_nonsym = assemble_Lh_nonsym(a, b, N, D).toarray()
Lh_sym = assemble_Lh_sym(a, b, N, D).toarray()
```

[214]:
```
# Konvergenzplot
plot_u_assemblies_error(a, b, N_min, N_max, s, u, f, D)
```



2.4 d) Springender Koeffizient D

Im folgenden Beispiel betrachten wir einen springenden Koeffizienten

$$D(x) = \begin{cases} d_1 & x < 0.5, \\ d_2 & x \geq 0.5, \end{cases} \qquad \text{mit} \quad d_1, d_2 > 0.$$

Konstruieren Sie sich für die rechte Seite $f(x) = x^k$, $k \in \mathbb{N}$, die Lösung des Randwertproblems.

Hinweis: Lösen Sie das RWP auf beiden Teilintervallen und nutzen Sie ohne Beweis, dass für die Lösung u die Interface-Bedingungen

$$u(\tfrac{1}{2}^-) = u(\tfrac{1}{2}^+) \qquad \text{und} \qquad D(\tfrac{1}{2}^-)\partial_x u(\tfrac{1}{2}^-) = D(\tfrac{1}{2}^+)\partial_x u(\tfrac{1}{2}^+)$$

gelten. Mit $\frac{1}{2}^{\pm}$ meinen wir die links- und rechtsseitigen Grenzwerte.

$-d \cdot \partial_{x^2} u(x) = x^k$

$u(x) = -\frac{1}{d \cdot (k+1) \cdot (k+2)} x^{k+2} + c_1 \cdot x + c_2$

$u_1(0) = 0$

also

$u_1(x) = -\frac{1}{d_1 \cdot (k+1) \cdot (k+2)} x^{k+2} + c_{1,1} \cdot x$

wegen

$-\frac{1}{k+1} 0.5^{k+1} + d_1 c_{1,1} = D(0.5^-) \partial_x u(0.5^-) = D(0.5^+) \partial_x u(0.5^+) = -\frac{1}{k+1} 0.5^{k+1} + d_2 c_{2,1}$

also $c_{1,1} = \frac{d_2}{d_1} c_{2,1}$

$u_2(x) = -\frac{1}{d_2 \cdot (k+1) \cdot (k+2)} x^{k+2} + c_{2,1} \cdot x + c_{2,2}$

$c_{2,2} = \frac{1}{d_1 \cdot d_2 \cdot (k+1) \cdot (k+2) \cdot 2^{k+2}} \cdot (d_1 - d_2) + 0.5 \cdot c_{2,1} \cdot (\frac{d_2}{d_1} - 1)$

$u_2(1) = -\frac{1}{d_2 \cdot (k+1) \cdot (k+2)} + c_{2,1} + c_{2,2} = 0$

also

$c_{2,1} = \frac{\frac{1}{d_2 \cdot (k+1) \cdot (k+2)} - \frac{1}{d_1 \cdot d_2 \cdot (k+1) \cdot (k+2) \cdot 2^{k+2}} \cdot (d_1 - d_2)}{1 + 0.5 \cdot (\frac{d_2}{d_1} - 1)}$

Stellen Sie die numerischen Lösungen, die Sie durch die beiden Assemblierungen erhalten, zusammen mit der exakten Lösung in einen Plot dar. Probieren Sie eine ungerade sowie eine gerade Anzahl Gitterpunkte. Erklären Sie, was Sie sehen.

[215]:
```
a = 0
b = 1
N = 101
k = 4
d_1 = 1.
d_2 = 10
D = lambda x: np.where(x >= 0.5, np.full_like(x, d_2), np.full_like(x, d_1))
f = lambda x: x**k
```

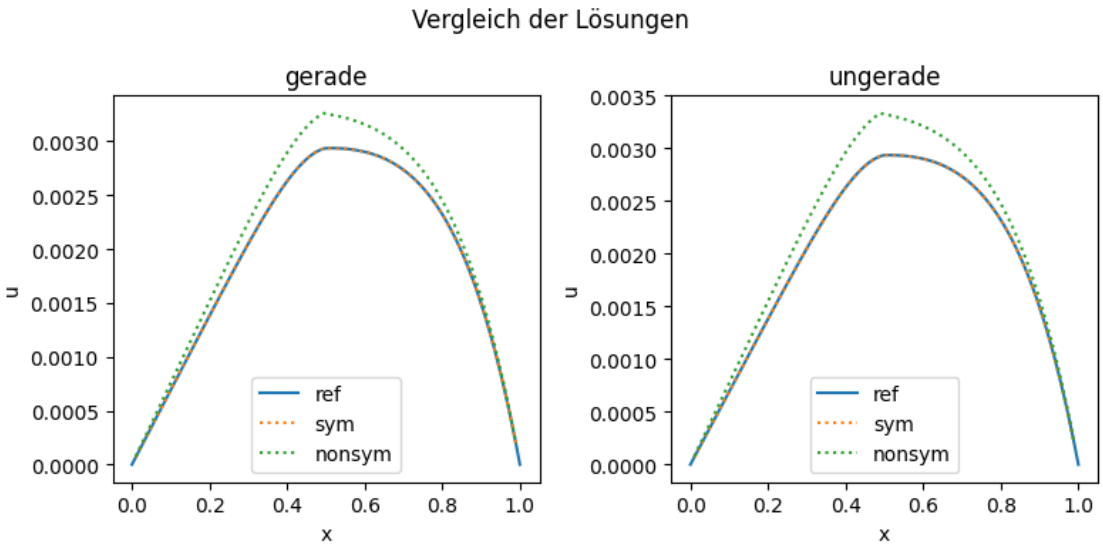
[216]:
```
def u_ex(x, d_1, d_2, k=0):
    c21 = ((1 / (d_2 * (k + 1) * (k + 2))) - (1 / (d_1 * d_2 * (k + 1) * (k +␣
 ↪2) * 2**(k + 2))) * (d_1 - d_2)) / (1 + 0.5 * ((d_2 / d_1) - 1))
    c11 = (d_2 / d_1) * c21
    c2 = (1 / (d_1 * d_2 * (k + 1) * (k + 2) * 2**(k + 2))) * (d_1 - d_2) + 0.5␣
 ↪* c21 * ((d_2 / d_1) - 1)
    coeff1 = - 1 / (d_1 * (k + 1) * (k + 2))
    coeff2 = - 1 / (d_2 * (k + 1) * (k + 2))
    u1 = lambda x: coeff1 * x**(k + 2) + c11 * x
    u2 = lambda x: coeff2 * x**(k + 2) + c21 * x + c2
    return np.where(x >= 0.5, u2(x), u1(x))
```

```
u = lambda x: u_ex(x, d_1, d_2, k=k)
```

[217]:
```
Lh_nonsym = assemble_Lh_nonsym(a, b, N, D)
Lh_sym = assemble_Lh_sym(a, b, N, D)
xh = np.linspace(a, b, N + 1)
fh = f(xh)[1:-1] # only inner x
```

[218]:
```
uh_sym = scipy.sparse.linalg.spsolve(Lh_sym, fh, use_umfpack=True)
uh_nonsym = scipy.sparse.linalg.spsolve(Lh_nonsym, fh, use_umfpack=True)
```

[219]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
plt.suptitle('Vergleich der Lösungen')

xh_even = np.linspace(a, b, N + 1)
ax1.plot(xh_even, u_ex(xh_even, d_1, d_2, k), label="ref")
ax1.plot(xh_even[1:-1], uh_sym, linestyle=":", label="sym")
ax1.plot(xh_even[1:-1], uh_nonsym, linestyle=":", label="nonsym")
ax1.set_xlabel("x")
ax1.set_ylabel("u")
ax1.set_title("gerade")
ax1.legend()

xh_odd = np.linspace(a, b, N)
fh_odd = f(xh_odd)[1:-1] # only inner x
Lh_nonsym_odd = assemble_Lh_nonsym(a, b, N - 1, D)
Lh_sym_odd = assemble_Lh_sym(a, b, N - 1, D)
uh_sym_odd = scipy.sparse.linalg.spsolve(Lh_sym_odd, fh_odd, use_umfpack=True)
uh_nonsym_odd = scipy.sparse.linalg.spsolve(Lh_nonsym_odd, fh_odd,␣
 ↪use_umfpack=True)
ax2.plot(xh_odd, u_ex(xh_odd, d_1, d_2, k), label="ref")
ax2.plot(xh_odd[1:-1], uh_sym_odd, linestyle=":", label="sym")
ax2.plot(xh_odd[1:-1], uh_nonsym_odd, linestyle=":", label="nonsym")
ax2.set_xlabel("x")
ax2.set_ylabel("u")
ax2.legend()
ax2.set_title("ungerade")
plt.show()
```

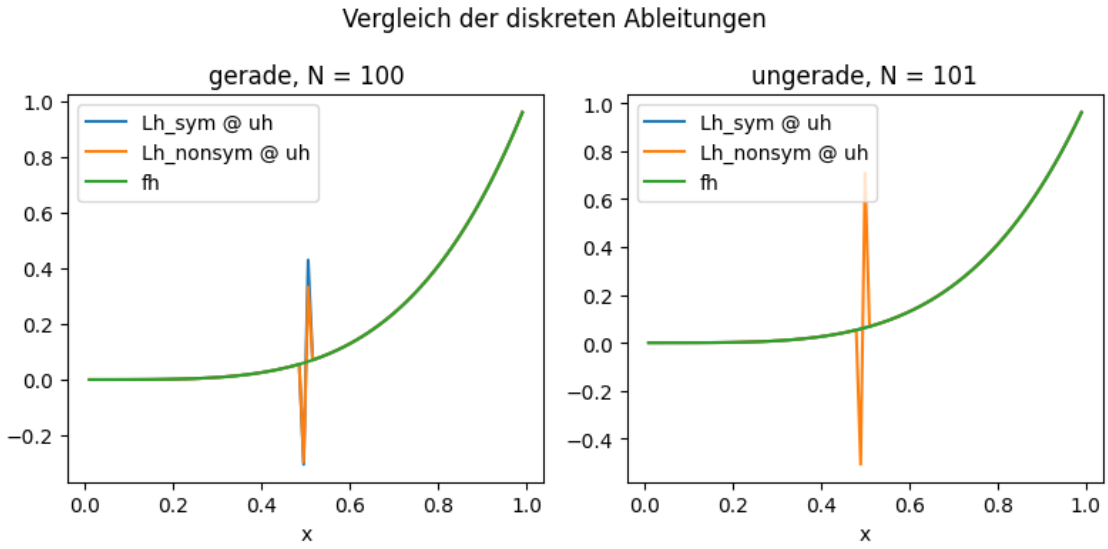


Plotten Sie die diskrete Ableitung $L_h u_h$ für beide Diskretisierungen, sowie die rechte Seite f_h. Was passiert, wenn Sie nur grade bzw. ungerade Anzahlen Gitterpunkte wählen?

[220]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))

def dis(a, b, N, D):
  Lh_nonsym = assemble_Lh_nonsym(a, b, N, D)
  Lh_sym = assemble_Lh_sym(a, b, N, D)
  xh = np.linspace(a, b, N + 1)
  fh = f(xh)[1:-1] # only inner x
  uh = u_ex(xh, d_1, d_2, k)[1:-1]
  return (Lh_nonsym, Lh_sym, xh, fh, uh)

# even
Lh_nonsym_even, Lh_sym_even, xh_even, fh_even, uh_even = dis(a, b, 99, D)
ax1.plot(xh_even[1:-1], Lh_sym_even @ uh_even, label="Lh_sym @ uh")
ax1.plot(xh_even[1:-1], Lh_nonsym_even @ uh_even, label="Lh_nonsym @ uh")
ax1.plot(xh_even[1:-1], fh_even, label="fh")
ax1.set_xlabel('x')
ax1.legend()
ax1.set_title(f"gerade, N = {100}")

# odd
Lh_nonsym_odd, Lh_sym_odd, xh_odd, fh_odd, uh_odd = dis(a, b, 100, D)
ax2.plot(xh_odd[1:-1], Lh_sym_odd @ uh_odd, label="Lh_sym @ uh")
ax2.plot(xh_odd[1:-1], Lh_nonsym_odd @ uh_odd, label="Lh_nonsym @ uh")
ax2.plot(xh_odd[1:-1], fh_odd, label="fh")
ax2.set_xlabel('x')
ax2.legend()
```

```
ax2.set_title(f"ungerade, N = {101}")
plt.suptitle('Vergleich der diskreten Ableitungen')
plt.legend()
```

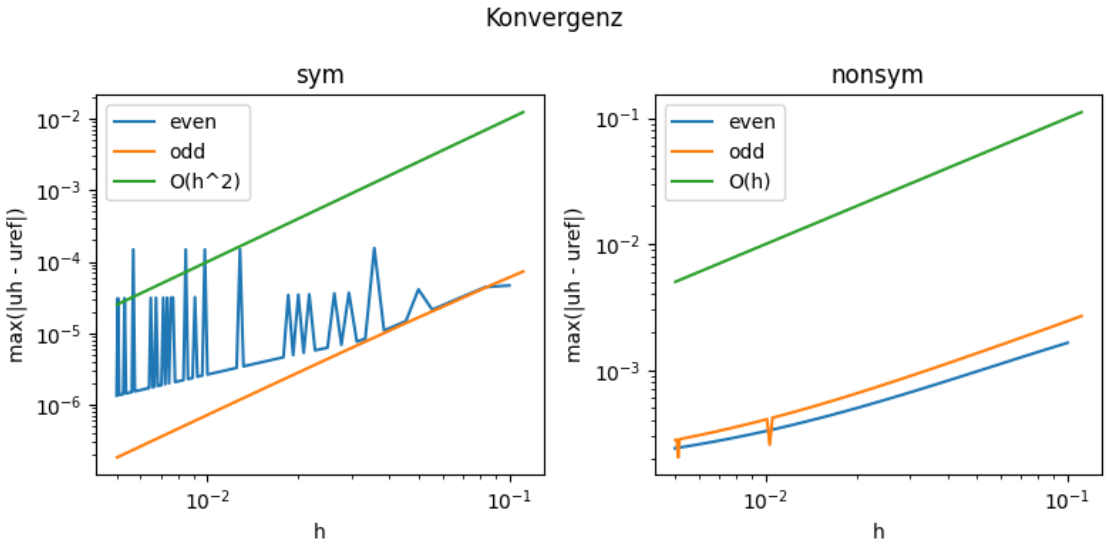
[220]: `<matplotlib.legend.Legend at 0x7e9cf4a0eb90>`



Erstellen Sie einen Konvergenzplot für Ihre beiden numerischen Lösungen. Was passiert, wenn Sie nur gerade bzw. nur ungerade viele Gitterpunkte wählen?

[221]:
```
# Konvergenzplot
N_min = 10
N_max = 201
error_sym_even = np.empty(((N_max - N_min + 1) // 2))
error_sym_odd = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_even = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_odd = np.empty(((N_max - N_min + 1) // 2))
for i, N in enumerate(range(N_min, N_max, 2)):
  Lh_nonsym_even, Lh_sym_even, _, fh_even, uh_even = dis(a, b, N + 1, D)
  Lh_nonsym_odd, Lh_sym_odd, _, fh_odd, uh_odd = dis(a, b, N, D)
  uh_sym_even = scipy.sparse.linalg.spsolve(Lh_sym_even, fh_even,␣
 ↪use_umfpack=True)
  uh_nonsym_even = scipy.sparse.linalg.spsolve(Lh_nonsym_even, fh_even,␣
 ↪use_umfpack=True)
  uh_sym_odd = scipy.sparse.linalg.spsolve(Lh_sym_odd, fh_odd, use_umfpack=True)
  uh_nonsym_odd = scipy.sparse.linalg.spsolve(Lh_nonsym_odd, fh_odd,␣
 ↪use_umfpack=True)
  error_sym_even[i] = np.max(np.abs(uh_sym_even - uh_even))
  error_nonsym_even[i] = np.max(np.abs(uh_nonsym_even - uh_even))
```

```
  error_sym_odd[i] = np.max(np.abs(uh_sym_odd - uh_odd))
  error_nonsym_odd[i] = np.max(np.abs(uh_nonsym_odd - uh_odd))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_sym_even, label="even")
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_sym_odd, label="odd")
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1)**2, label="O(h^2)")
ax1.set_xlabel("h")
ax1.set_ylabel("max(|uh - uref|)")
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_title("sym")
ax1.legend()
ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_nonsym_even, label="even")
ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_nonsym_odd, label="odd")
ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1), label="O(h)")
ax2.set_xlabel("h")
ax2.set_ylabel("max(|uh - uref|)")
ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_title("nonsym")
ax2.legend()
plt.suptitle("Konvergenz")
plt.show()
```



symmetrische Approximation, die Konvergenz der Ordnung 2 für ungerade Anzahl an Gitterpunkten wird erklärt durch Beispiel 14.9, da die Unstetigkeitsstelle bei 0.5, für ungerade Anzahl an Gitterpunkten is 0.5 in der Mitte zwischen 2 Gitterpunkten

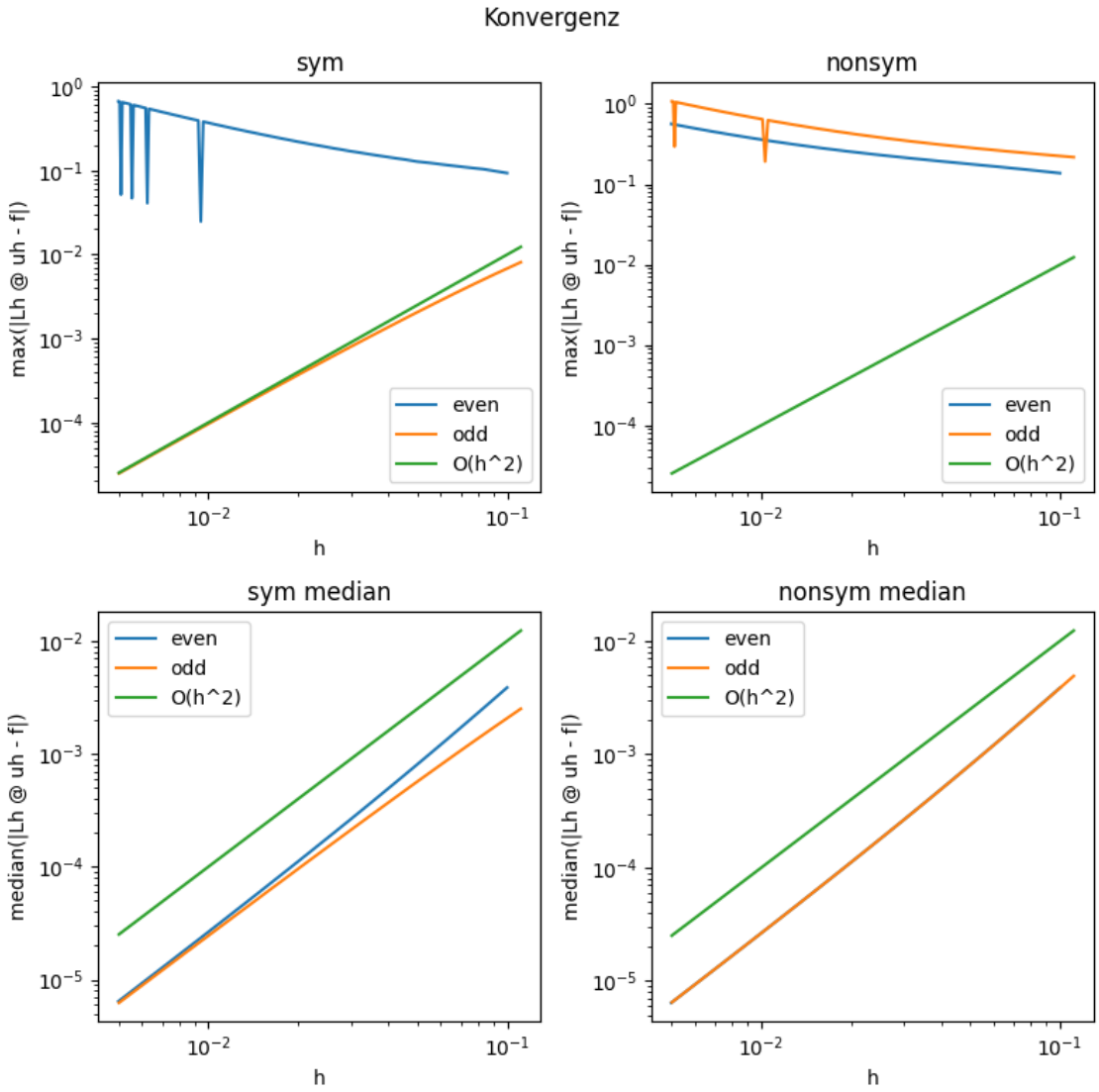
nichtsymmetrische Approximation wurde die Abschätzung nicht gezeigt und hier auch nicht zu sehen

Erstellen Sie einen Konvergenzplot für die Diskretisierung der Ableitung. Unterscheiden Sie erneut zwischen gerader und ungerader Anzahl Gitterpunkte.

[222]:
```
# Konvergenzplot für die diskrete Ableitung
N_min = 10
N_max = 201
error_sym_even = np.empty(((N_max - N_min + 1) // 2))
error_sym_odd = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_even = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_odd = np.empty(((N_max - N_min + 1) // 2))
error_sym_even_median = np.empty(((N_max - N_min + 1) // 2))
error_sym_odd_median = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_even_median = np.empty(((N_max - N_min + 1) // 2))
error_nonsym_odd_median = np.empty(((N_max - N_min + 1) // 2))
for i, N in enumerate(range(N_min, N_max, 2)):
  Lh_nonsym_even, Lh_sym_even, _, fh_even, uh_even = dis(a, b, N + 1, D)
  Lh_nonsym_odd, Lh_sym_odd, _, fh_odd, uh_odd = dis(a, b, N, D)
  uh_sym_even = scipy.sparse.linalg.spsolve(Lh_sym_even, fh_even,␣
 ↪use_umfpack=True)
  uh_nonsym_even = scipy.sparse.linalg.spsolve(Lh_nonsym_even, fh_even,␣
 ↪use_umfpack=True)
  uh_sym_odd = scipy.sparse.linalg.spsolve(Lh_sym_odd, fh_odd, use_umfpack=True)
```

```
  uh_nonsym_odd = scipy.sparse.linalg.spsolve(Lh_nonsym_odd, fh_odd,␣
 ↪use_umfpack=True)
  error_sym_even[i] = np.max(np.abs(Lh_sym_even @ uh_even - fh_even))
  error_nonsym_even[i] = np.max(np.abs(Lh_nonsym_even @ uh_even - fh_even))
  error_sym_odd[i] = np.max(np.abs(Lh_sym_odd @ uh_odd - fh_odd))
  error_nonsym_odd[i] = np.max(np.abs(Lh_nonsym_odd @ uh_odd - fh_odd))
  # median, max is high because of incontinuity
  error_sym_even_median[i] = np.median(np.abs(Lh_sym_even @ uh_even - fh_even))
  error_sym_odd_median[i] = np.median(np.abs(Lh_sym_odd @ uh_odd - fh_odd))
  error_nonsym_even_median[i] = np.median(np.abs(Lh_nonsym_even @ uh_even -␣
 ↪fh_even))
  error_nonsym_odd_median[i] = np.median(np.abs(Lh_nonsym_odd @ uh_odd -␣
 ↪fh_odd))

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(8, 8))
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_sym_even, label="even")
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_sym_odd, label="odd")
ax1.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1)**2, label="O(h^2)")
ax1.set_xlabel("h")
ax1.set_ylabel("max(|Lh @ uh - f|)")
ax1.set_xscale("log")
ax1.set_yscale("log")
ax1.set_title("sym")
ax1.legend()

ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_nonsym_even, label="even")
ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_nonsym_odd, label="odd")
ax2.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1)**2, label="O(h^2)")
ax2.set_xlabel("h")
ax2.set_ylabel("max(|Lh @ uh - f|)")
ax2.set_xscale("log")
ax2.set_yscale("log")
ax2.set_title("nonsym")
ax2.legend()

ax3.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_sym_even_median, label="even")
```

```
ax3.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_sym_odd_median, label="odd")
ax3.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1)**2, label="O(h^2)")
ax3.set_xlabel("h")
ax3.set_ylabel("median(|Lh @ uh - f|)")
ax3.set_xscale("log")
ax3.set_yscale("log")
ax3.set_title("sym median")
ax3.legend()

ax4.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16)), error_nonsym_even_median, label="even")
ax4.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), error_nonsym_odd_median, label="odd")
ax4.plot((b - a) * np.reciprocal(np.array(list(range(N_min, N_max, 2)),␣
 ↪dtype=np.float16) - 1), (b - a) * np.reciprocal(np.array(list(range(N_min,␣
 ↪N_max, 2)), dtype=np.float16) - 1)**2, label="O(h^2)")
ax4.set_xlabel("h")
ax4.set_ylabel("median(|Lh @ uh - f|)")
ax4.set_xscale("log")
ax4.set_yscale("log")
ax4.set_title("nonsym median")
ax4.legend()
plt.suptitle("Konvergenz")
plt.show()
```



symmetrische Approximation analog die Abschätzung für die geraden Anzahl an Gitterpunkten

auffällig sind die max Error welche für geringe h höher werden, wegen dem Median is die Konvergenz im Großen und Ganzen, also insbesondere ohne die Unstetigkeitsstelle $O(h^2)$

4,5/5

3 Aufgabe 2 - Schnelle Löser auf Würfeln mit homogener Diffusion

In dieser Aufgabe wollen wir uns mit den Inhalten von Abschnitt 14.3 beschäftigen. Genauer wollen wir uns näher mit der Lithium-Batterie aus Abbildung 14.1 beschäftigen. Dabei arbeiten wir stets auf

$$\Omega = (0, \pi)^2 \subset \mathbb{R}^2,$$

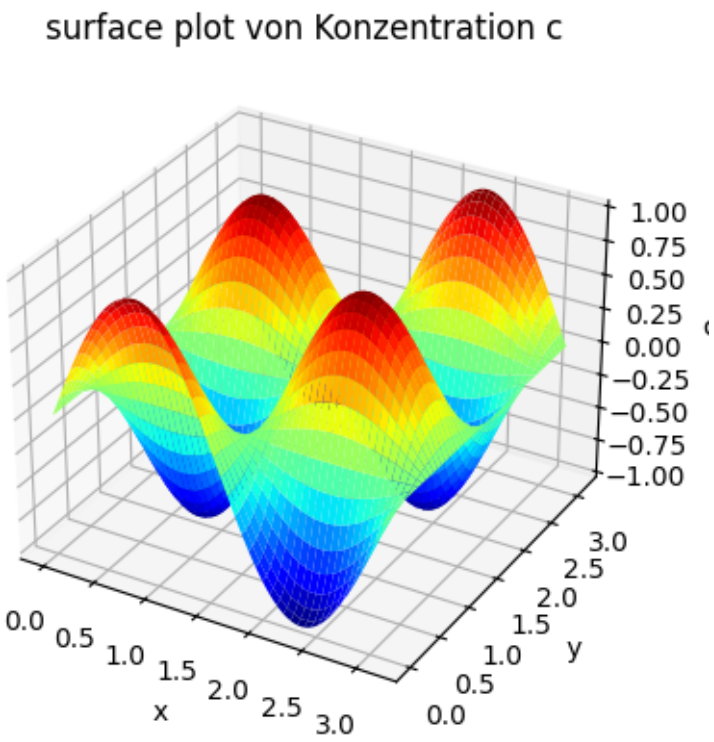
wobei wir in x-Richtung Dirichlet-RB und in y-Richtung Neumann-RB verwenden. Später wollen wir die diskrete Sinus- und Kosinustransformation nutzen. Legen Sie zunächst die benötigten

Gitter an. Achten Sie darauf, dass die Gitter zu den benötigten Transformationen in die jeweiligen Koordinatenrichtungen passen. Nutzen Sie dazu die Funktion `np.meshgrid(...)`.

[223]:
```
# Space parameters
a = 0
b = np.pi
N = 2**10
h = np.pi / N
# set up grids
xh = np.linspace(a, b, N + 1)
yh = np.linspace(a, b - h, N) + 0.5 * h
Xh, Yh = np.meshgrid(xh, yh)
Xh_dst = Xh[:, 1:-1]
Yh_dst = Yh[:, 1:-1]
```

In der folgenden Zelle ist eine Funktion c sowie ihre Ableitungen $-\partial_{xx}c$ und $-\partial_{yy}c$ gegeben. Stellen Sie c über Ihrem Gitter in einem Schaubild dar. Sie können dafür z.B. 'plt.plot_surface' benutzen.

[224]:
```
# Beispiel fuer c
c = lambda x, y: np.sin(2 * x) * np.cos(3 * y)
Lx_c = lambda x, y: 4 * np.sin(2 * x) * np.cos(3 * y)
Ly_c = lambda x, y: 9 * np.sin(2 * x) * np.cos(3 * y)

## plot
fig, ax = plt.subplots(1, 1, figsize=(4, 4), subplot_kw={"projection": "3d"})
ax.plot_surface(Xh, Yh, c(Xh, Yh), cmap=cm.jet)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("c")
ax.set_title("surface plot von Konzentration c")
plt.show()
```



3.1 a) Frequenztests

Bevor uns nun mit dem Lösen des Randwertproblems beschäftigen, wollen wir zunächst an den Ableitungen des oben definierten c testen, dass die im Skript berechneten Frequenzen korrekt sind.

Schreiben Sie einen Test, in dem Sie c an Ihrem Gitter auswerten und über die diskrete Sinustransformation eine Approximation an $-\partial_{xx}c$ berechnen. Überprüfen Sie anschließend ob diese Approximation hinreichend nahe an der Auswertung von `Lx_c(...)` auf Ihrem Gitter ist. Setzen Sie dafür `rtol` auf 0 und `atol` auf $10h^2$.

[225]:
```
%%ipytest -vv
def test_dxx():
    # Test für Ableitung -d_xx
    c_coeff = scipy.fft.dst(c(Xh_dst, Yh_dst), type=1, axis=1)
    minus_cxx_coeff = c_coeff * (- np.arange(1, N, 1)**2)
    minus_cxx = scipy.fft.idst(minus_cxx_coeff, type=1, axis=1)
    np.testing.assert_allclose(minus_cxx, -1*Lx_c(Xh_dst, Yh_dst), atol=10 *␣
 ↪h**2, rtol=0)
```

```
========================================= test session starts
=========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
```

```
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_dxx PASSED

[100%]

========================================= 1 passed in

0.26s =========================================
```

Schreiben Sie auch einen Test für die diskrete Kosinustransformation zur Approximation von $-\partial_{yy}c$. Vergleichen Sie mit der Auswertung von `Ly_c`.

[226]:
```
%%ipytest -vv
def test_dyy():
    # Test für Ableitung -d_yy
    c_coeff = scipy.fft.dct(c(Xh, Yh), type=2, axis=0, orthogonalize=True)
    minus_cyy_coeff = np.transpose(np.transpose(c_coeff) * (np.concat((np.
 ↪array([1]), (- np.arange(1, N, 1)**2)), axis=0)))
    minus_cyy = scipy.fft.idct(minus_cyy_coeff, type=2, axis=0,␣
 ↪orthogonalize=True)
    np.testing.assert_allclose(minus_cyy, -1*Ly_c(Xh, Yh), atol=10 * h**2,␣
 ↪rtol=0)
```

```
======================================== test session starts

=========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_dyy PASSED

[100%]

========================================= 1 passed in

0.37s =========================================
```

Testen Sie zuletzt auch die Kombination aus diskreter Sinus- und Kosinustransformation zur Approximation von $-\Delta c$. Die Reihenfolge sollte dabei die folgende sein: 1. Auswertung von c 2. diskrete Sinustransformation in x-Richtung 3. diskrete Kosinustransformation in y-Richtung 4. Multiplikation mit den erforderlichen Frequenzen 5. inverse diskrete Kosinustransformation in y-Richtung 6. inverse diskrete Sinustransformation in x-Richtung

Warum liefert eine solche Schachtelung der Transformationen das korrekte Ergebnis?

[227]:
```
%%ipytest -vv
def test_mlaplace():
    # Ableitung -d_xx -d_yy
    c_grid = c(Xh, Yh)
    c_dst = scipy.fft.dst(c_grid[:,1:-1], type=1, axis=1)
    zeroes = np.expand_dims(c_grid[:,0], 1)
    c_dst_full = np.concat((zeroes, c_dst, zeroes), axis=1)
    c_dst_dct = scipy.fft.dct(c_dst_full, type=2, axis=0, orthogonalize=True)
    minus_pois_coeff = c_dst_dct * (np.concat((np.array([1]), -1 * np.arange(1,␣
 ↪N, 1)**2, np.array([1])), axis=0)) + np.transpose(np.transpose(c_dst_dct) *␣
 ↪(np.concat((np.array([1]), -1 * np.arange(1, N, 1)**2), axis=0)))
    minus_pois_dst = scipy.fft.idct(minus_pois_coeff, type=2, axis=0,␣
 ↪orthogonalize=True)
    minus_pois = np.concat((zeroes, scipy.fft.idst(minus_pois_dst[:,1:-1],␣
 ↪type=1, axis=1), zeroes), axis=1)
    np.testing.assert_allclose(minus_pois, -Lx_c(Xh, Yh) - Ly_c(Xh, Yh),␣
 ↪atol=10 * h**2, rtol=0)
```

```
========================================= test session starts
==========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_mlaplace PASSED
[100%]

========================================== 1 passed in
0.51s ==========================================
```

<– Platz für Ihre Antwort –>

Die Fourierreihe von f bezüglich x und y kann als Sinustransformation für x und davon die Cosinustransformation für y gesehen werden

Die diskreten Fourierkoeffizienten können also als Verkettung von DST und DCT erhalten werden

2 maliges Ableiten der Fourierreihe multipliziertdie Fourierkoeffizienten mit $-k^2 - l^2$ Eigtl mit EW der FD-Matrix

Die inversen DST und DCT liefern also die Fourierkoeffizienten von $-\Delta c$

3.2 b) Ein schneller Löser für $(I - \alpha\Delta)u = f$ mit gemischten Randdaten

Schreiben Sie nun eine Funktion `solve_mixed_Laplace()`, welche als Parameter `h`, `alpha` und die ausgewertete rechte Seite `rhs` bekommt und die Lösung $u(x, y)$ des homogenen gemischten

Randwertproblems

$$(I - \alpha\Delta)u = f, \qquad \text{in } (0,\pi)^2 \tag{1}$$
$$u(0,y) = u(\pi,y) = 0, \qquad \text{für } y \in [0,\pi] \tag{2}$$
$$\partial_n u(x,0) = \partial_n u(x,\pi) = 0 \qquad \text{für } x \in [0,\pi] \tag{3}$$

approximiert. Nutzen Sie dazu die Techniken aus Abschnitt 14.3 im Skript.

Sei $A_h = I + \alpha(A_h^{Dir} \bigotimes I) + \alpha(I \bigotimes A_h^{Neu})$ und $U_N = (S_N \bigotimes C_N)$

also

$U_N A_h U_N^{-1} = I + U_N(\alpha A_h^{Dir} \bigotimes I)U_N^{-1} + U_N(I \bigotimes \alpha A_h^{Neu})U_N^{-1} = I + \alpha(S_N A_h^{Dir} S_N^{-1} \bigotimes I) + \alpha(I \bigotimes C_N A_h^{Neu} C_N^{-1}) = I + (diag(\alpha\lambda_1, ..., \alpha\lambda_{N-1}) \bigotimes I) + (I \bigotimes diag(\alpha\tilde{\lambda}_1, ..., \alpha\tilde{\lambda}_N))$

[228]:
```
# solve I - alpha Laplace
def solve_mixed_Laplace(h, alpha, rhs, rhs_is_lambda=True, dtype=np.float32):
    N = int(np.pi // h)
    xh = np.linspace(0, np.pi, N + 1, dtype=dtype)[1:-1]
    yh = np.linspace(0, np.pi - h, N, dtype=dtype) + 0.5 * h
    Xh, Yh = np.meshgrid(xh, yh)
    # lambdas
    sinus_lambda = alpha * (4 / h**2) * np.sin(0.5 * h * np.arange(1, N, 1,␣
 ↪dtype=dtype))**2
    cosinus_lambda = alpha * (4 / h**2) * np.sin(0.5 * h * np.arange(0, N, 1,␣
 ↪dtype=dtype))**2
    # diag of U_N A_h U_N^{-1} as matrix as f is not vector
    #diag = np.kron(sinus_lambda, np.ones_like(cosinus_lambda)) + np.kron(np.
 ↪ones_like(sinus_lambda), cosinus_lambda)
    diag_matrix = 1 + np.tile(sinus_lambda, (N, 1)) + np.tile(np.
 ↪expand_dims(cosinus_lambda, 1), (1, N - 1))
    # solve lgs U_N A_h U_N^{-1} U_N u = U_N rhs
    U_N = lambda x: scipy.fft.dst(scipy.fft.dct(x, type=2, axis=0,␣
 ↪orthogonalize=True), type=1, axis=1)
    if rhs_is_lambda:
      U_N_rhs = U_N(rhs(Xh, Yh))
    else:
      U_N_rhs = U_N(rhs)
    #U_N_u = diag**(-1) * U_N_rhs
    U_N_u = np.reciprocal(diag_matrix) * U_N_rhs
    u = scipy.fft.idct(scipy.fft.idst(U_N_u, type=1, axis=1), type=2, axis=0,␣
 ↪orthogonalize=True) # u using inverse transformations
    return u
```

Legen Sie zum Testen Ihres Lösers mit `sympy` eine exakte Lösung $u(x,y)$ an und bestimmen Sie die rechte Seite f so, dass

$$(I - \alpha\Delta)u = f$$

gilt. Achten Sie darauf, dass u auch die korrekten homogenen Randbedingung erfüllt.

[229]:
```python
# param
alpha = 2.0
# method of manufactured solution
import sympy as sp
from functools import partial

x_sp, y_sp, alpha_sp = sp.symbols("x y alpha")
u_sp = sp.sin(x_sp) * sp.cos(y_sp)
f_sp = u_sp - alpha_sp * (sp.diff(u_sp, x_sp, x_sp) + sp.diff(u_sp, y_sp, y_sp))
u_np = sp.lambdify((x_sp, y_sp), u_sp, "numpy")
f_np_alpha = sp.lambdify((x_sp, y_sp, alpha_sp), f_sp, "numpy")
f_np = partial(f_np_alpha, alpha=alpha)
rhs = lambda x, y: f_np(x, y)
print(f"given exact solution: {u_sp}")
print(f"calculated rhs: {f_sp}")
```

```
given exact solution: sin(x)*cos(y)
calculated rhs: 2*alpha*sin(x)*cos(y) + sin(x)*cos(y)
```

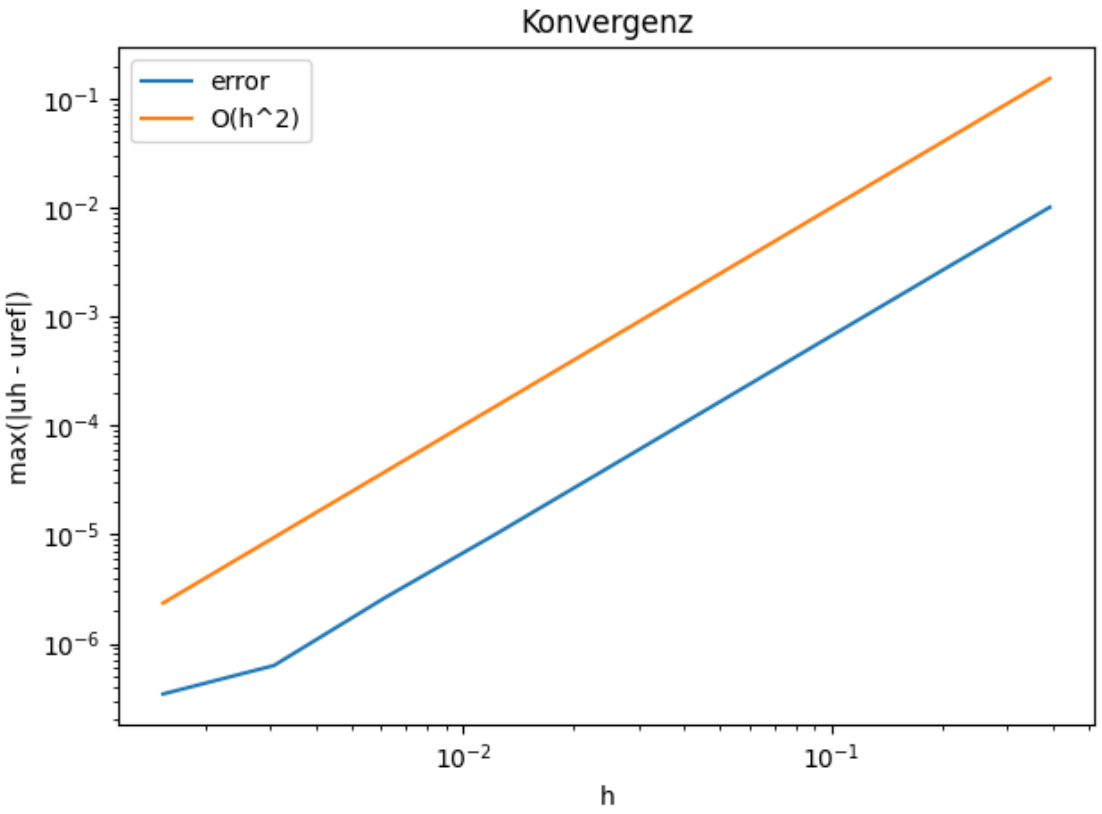
Schreiben Sie nun einen Test, in dem Sie überprüfen, ob die Approximation Ihres Lösers hinreichend nahe an der exakten Lösung $u(x, y)$ liegt. Nutzen Sie dazu die exkate Lösung u und rechte Seite f, die Sie sich im letzten Schritt vorgegeben haben.

[230]:
```python
%%ipytest -vv
def test_solver_mms():
    # grid
    xh = np.linspace(0, np.pi, N + 1)[1:-1]
    h = np.pi / N
    yh = np.linspace(0, np.pi - h, N) + 0.5 * h
    Xh, Yh = np.meshgrid(xh, yh)
    ref = u_np(Xh, Yh) # ref
    sol = solve_mixed_Laplace(h, alpha, rhs)  # approx
    np.testing.assert_allclose(sol, ref, atol=10 * h**2, rtol=0)
```

```
========================================= test session starts
=========================================
platform linux -- Python 3.11.13, pytest-8.3.5, pluggy-1.6.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /content
plugins: anyio-4.9.0, langsmith-0.3.45, typeguard-4.4.3
collecting … collected 1 item

t_11c50f6b742b471cab6a388de5e8291e.py::test_solver_mms PASSED
[100%]

========================================== 1 passed in
0.31s ==========================================
```

Erstellen Sie einen Konvergenzplot für verschiedene Wahlen von `N` bzw. `h`. Was sind hinsichtlich der Effizienz der DCT/DST sinnvolle Wahlen von `N`?

[231]:
```
max_exp = 11
min_exp = 3
errors = np.empty((max_exp - min_exp + 1))
for i, l in enumerate(range(min_exp, max_exp + 1)):
  N = 2**l
  # grid
  xh = np.linspace(0, np.pi, N + 1)[1:-1]
  h = np.pi / N
  yh = np.linspace(0, np.pi - h, N) + 0.5 * h
  Xh, Yh = np.meshgrid(xh, yh)
  ref = u_np(Xh, Yh) # ref
  sol = solve_mixed_Laplace(h, alpha, rhs)  # approx
  errors[i] = np.max(np.abs(ref - sol))
plt.plot([np.pi / 2**l for l in range(min_exp, max_exp + 1)], errors,␣
 ↪label="error")
plt.plot([np.pi / 2**l for l in range(min_exp, max_exp + 1)], np.array([np.pi /␣
 ↪2**l for l in range(min_exp, max_exp + 1)])**2, label="O(h^2)")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("h")
plt.ylabel("max(|uh - uref|)")
plt.title("Konvergenz")
plt.legend()
plt.show()
```



<– Platz für Ihre Antwort –>

Sinnvolle Wahlen sind N für die $2 \cdot N$ eine 2 - er Potenz, da somit der FFT am sinnvollsten das Divide and Conquer genutzt wird

3.3 c) Beispiel: Energie-Minimierung

Im Folgenden betrachten wir eine Anwendung für die schnellen Löser. Das Ziel ist dabei die Energie, bestehend aus der kinetischen Energie und einem Potential $W : \mathbb{R} \to \mathbb{R}$,

$$E(c) = \int_\Omega \frac{1}{2} |\nabla c(x)|^2 + W(c(x)) \, \mathrm{d}x \tag{4}$$

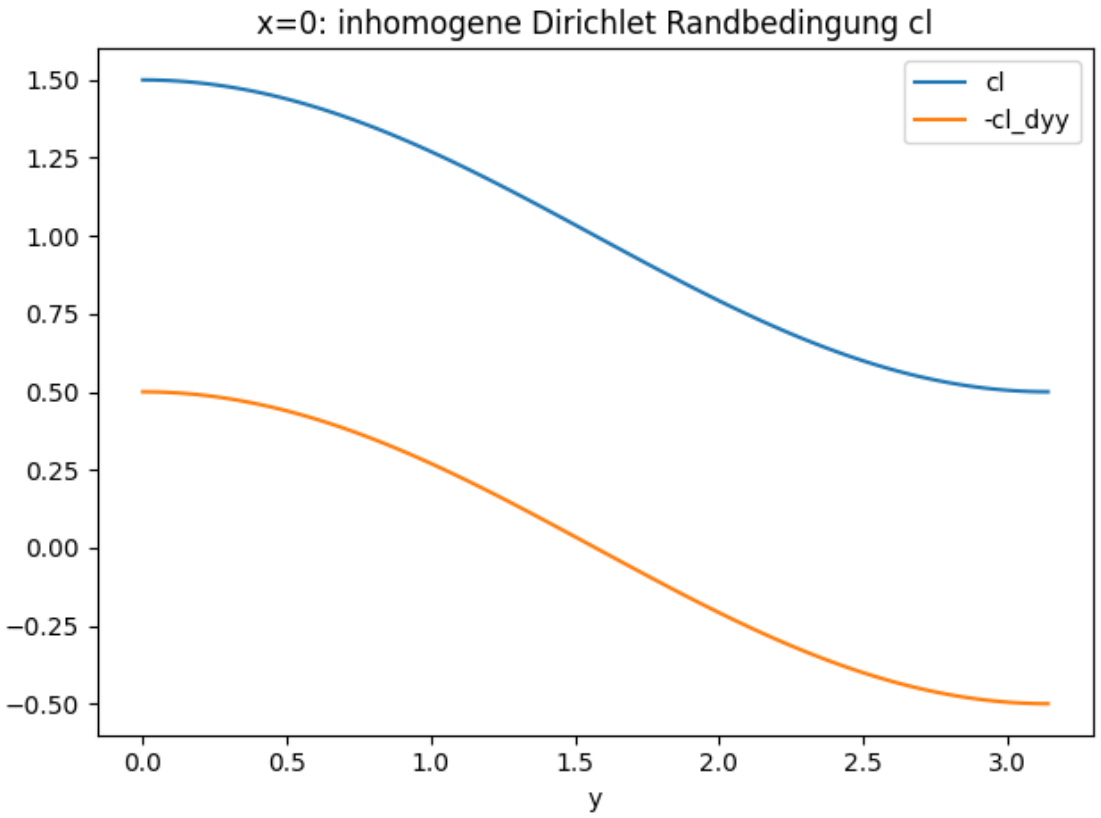
auf $\Omega = (0, \pi)^2$ mit den Randbedingungen $c(0, y) = c_\ell(y)$ und $c(\pi, y) = c_r(y) = 0$ und

$$c_\ell(y) = 1 + \frac{1}{2}\cos(y) \qquad \text{für } y \in [0, \pi]$$

zu minimieren. Legen Sie zunächst c_ℓ an, bestimmen Sie $-\partial_{yy} c_\ell$ und plotten Sie beides in ein gemeinsames Schaubild. Wenn Sie möchten können Sie `sympy` verwenden, Sie können aber auch direkt `lambda`-Ausdrücke schreiben.

[232]:
```
y_sp = sp.symbols("y")
## Linker Randwert g(y)
```

```
cl_sp = 1 + 0.5 * sp.cos(y_sp)
minus_cl_dyy_sp = -1 * sp.diff(cl_sp, y_sp, y_sp)
cl_np = sp.lambdify((y_sp), cl_sp, "numpy")
minus_cl_dyy_np = sp.lambdify((y_sp), minus_cl_dyy_sp, "numpy")

# plot
yh = np.linspace(0, np.pi, 100)
plt.plot(yh, cl_np(yh), label="cl")
plt.plot(yh, minus_cl_dyy_np(yh), label="-cl_dyy")
plt.xlabel("y")
plt.title("x=0: inhomogene Dirichlet Randbedingung cl")
plt.legend()
plt.show()
```

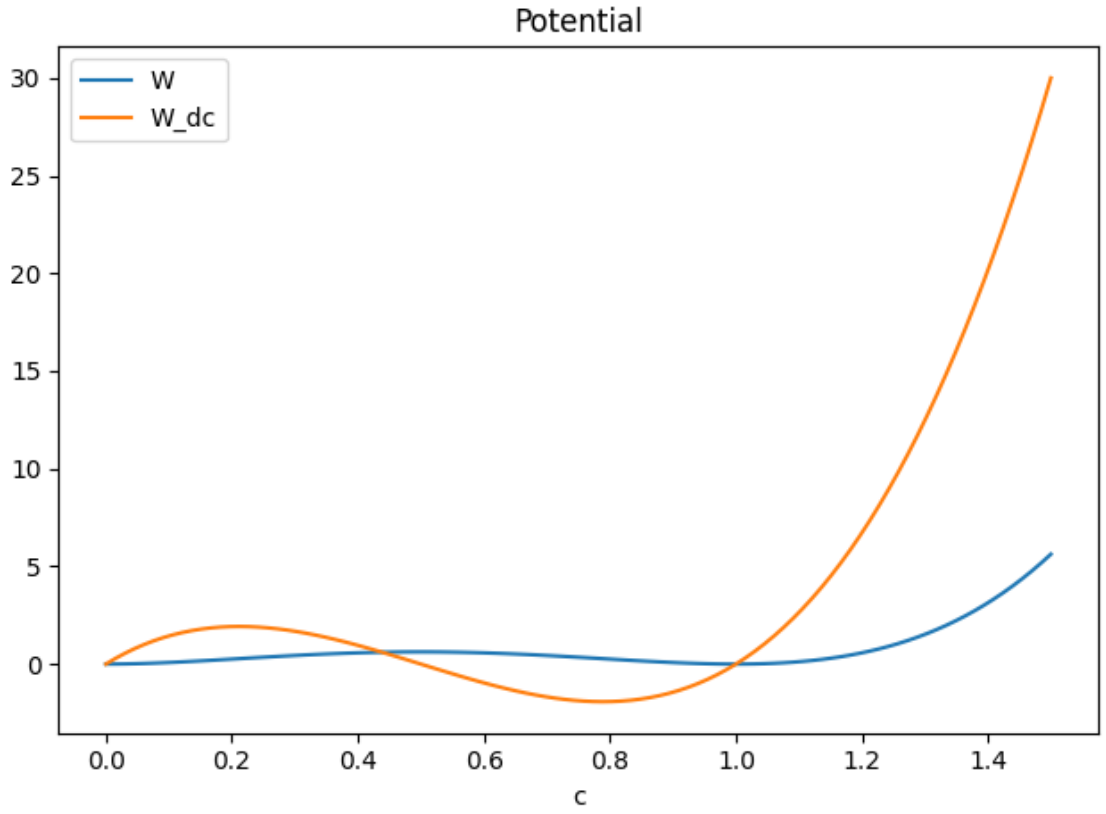


Legen Sie nun das Potential W an und bestimmen Sie auch W'. Plotten Sie anschließend beides in ein gemeinsames Schaubild. Beginnen Sie mit dem Potential

$$W(c) = 10c^2(1-c)^2$$

[233]:
```
c_sp = sp.symbols("c")
W_sp = 10 * c_sp**2 * (1 - c_sp)**2
```

```
W_dc_sp = sp.diff(W_sp, c_sp)
W_np = sp.lambdify((c_sp), W_sp, "numpy")
W_dc_np = sp.lambdify((c_sp), W_dc_sp, "numpy")

# plot
ch = np.linspace(0, 1.5, 100)
plt.plot(ch, W_np(ch), label="W")
plt.plot(ch, W_dc_np(ch), label="W_dc")
plt.xlabel("c")
plt.title("Potential")
plt.legend()
plt.show()
```

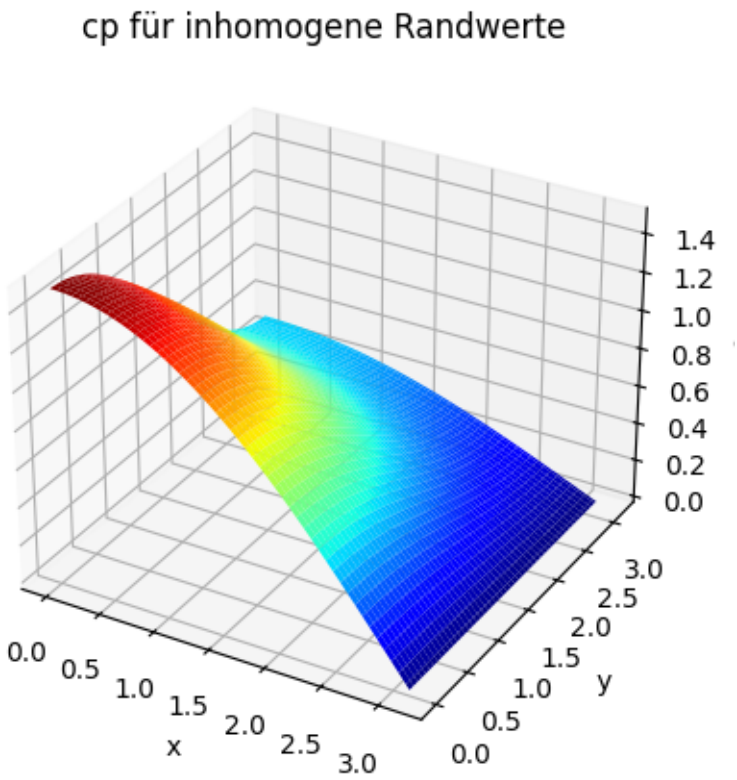


Wir wollen später den schnellen Löser aus Aufgabenteil b) verwenden um mithilfe des Gradientenflusses einen Minimierer zu bestimmen.

Bestimmen Sie dazu zunächst eine Funktion $c_p(x, y) = a(x)b(y)$, welche die Randwerte $c(0, y) = g(y), c(\pi, y) = 0, \partial_n c(x, 0) = \partial_n c(x, \pi) = 0$ erfüllt.

Plotten sie diese Funktion über Ihrem Gitter. Legen Sie auch ein `lambda` für $-\Delta c_p$ an.

[234]:
```
# define c_p and -\Delta c_p
cp = lambda x, y: np.cos(0.5 * x) * cl_np(y)
Lapl_cp = lambda x, y: 0.25 * np.cos(0.5 * x) * cl_np(y) + 0.5 * np.cos(0.5 *␣
 ↪x) * np.cos(y)

# plot
xh = np.linspace(0, np.pi, 100)
yh = np.linspace(0, np.pi, 100)
Xh, Yh = np.meshgrid(xh, yh)
fig, ax = plt.subplots(1, 1, figsize=(4, 4), subplot_kw={"projection": "3d"})
ax.plot_surface(Xh, Yh, cp(Xh, Yh), cmap=cm.jet)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("cp")
fig.tight_layout()
plt.title("cp für inhomogene Randwerte")
plt.show()
```



Die folgende Funktion berechnet für gegebene Approximationen c^n_{hom} und c_p eine diskrete Energie $E(c^n_{hom} + c_p)$. Diese Funktion könnte beim Debuggen der folgenden Aufgabe hilfreich sein. Was macht die Funktion `np.pad`? Welche Logik steckt hinter den aufgefüllten Werten in `c_padded`?

[235]:
```
# Derive energy
x_staggered = np.linspace(a, b, N + 1)[:-1] + 0.5 * h
y_arr = np.pad(x_staggered, (1, 1),␣
 ↪constant_values=(x_staggered[0]-h,x_staggered[-1]+h))

def derive_energy(y_arr, h, c_hom, cp, W, cl):

    c_padded = np.pad(c_hom + cp, (1, 1))
    c_padded[:, 0] = cl(y_arr)
    c_padded[0, 1:-1] = c_padded[1, 1:-1]
    c_padded[-1, 1:-1] = c_padded[-2, 1:-1]
    c_x = 1.0 / (2 * h) * (c_padded[1:-1, :-2] - c_padded[1:-1, 2:])
    c_y = 1.0 / (2 * h) * (c_padded[:-2, 1:-1] - c_padded[2:, 1:-1])

    Wc = W(c_hom + cp)
    discrete_energy = h**2 * (
        0.5 * np.linalg.norm(c_x) ** 2
        + 0.5 * np.linalg.norm(c_y) ** 2
        + np.sum(Wc)
    )
    return discrete_energy
```

<– Platz für Ihre Antwort –>

die aufgefüllten Werte der ersten und letzten Spalte erhalten jeweils die Randbedingung c_l und die homogene Randbedingung

die aufgefüllten Werte der ersten und letzten Zeile, jeweils ohne die Werte der ersten und letzten Spalte, sind die selben Werte wie die erste und letzte Zeile der gegebenen Werte und also wahrscheinlich eine Approximation der Werte am Rand in Anbetracht der homogenenen Neumann Randbedingungen $\frac{u_1-u_0}{h} \approx u_0' = 0$

Wie in Abschnitt 14.3.2 beschrieben zerlegen wir die Lösung c in $c = c_p + c_{hom}^n$ und erhalten für c_{hom}^n die Vorschrift

$$(I - \tau\Delta)c_{\text{hom}}^{n+1} = c_{\text{hom}}^n + \tau\Delta c_p - \tau W'(c_{\text{hom}}^n + c_p) \, .$$

Implementieren Sie dieses Verfahren und überprüfen Sie in jedem Schritt als Abbruchkriterium die diskrete L^2-norm von $c_{hom}^{n+1} - c_{hom}^n$. Brechen Sie die Vorschrift ab, sobald dieser Wert unter eine vorgegebene Toleranz fällt und geben Sie den berechneten Minimierer zurück. Bestimmen Sie außerdem die diskrete Energie Ihres Minimierers und vergleichen Sie diese mit $E(c_p)$.

[236]:
```
# time params
t0 = 0.0
tau = 0.01
Tmax = 0.5
tol = 1e-5
Nt = int(round((Tmax - t0) / tau))
```

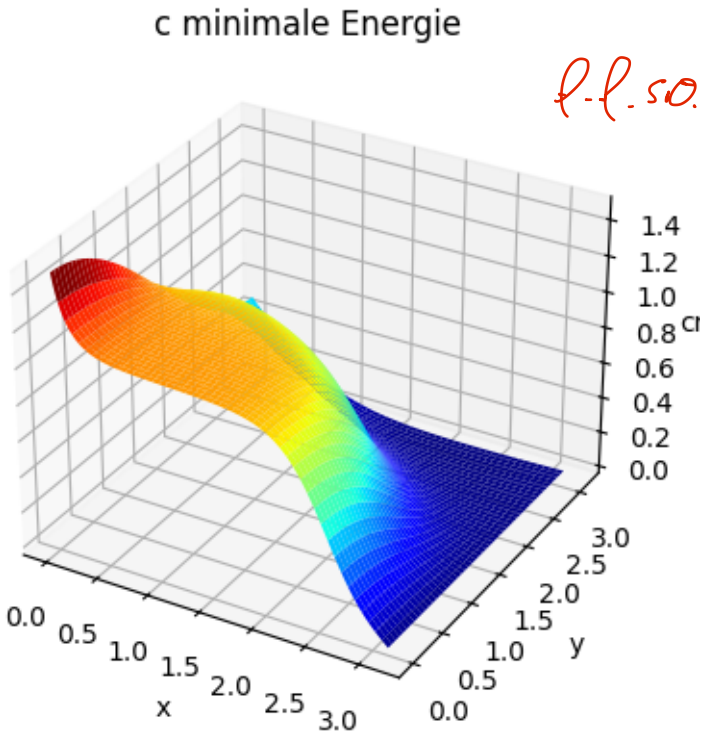
```
ts = np.linspace(t0, Tmax, Nt + 1)
#
def l2_norm(a, b, h1, h2):
  h = h1  * h2
  return h * np.sum(a * b)

def minimize_energy(initial, tau, cp, cp_laplace, W_dc, tol, Nt, Xh, Yh, h,␣
 ↪hard_tol=False):
  """
  params:
  hard_tol: if False then Nt is max iterations
  """
  prev = initial
  n = 0
  while True:
    rhs = prev + tau * cp_laplace(Xh, Yh) - tau * W_dc(prev + cp(Xh, Yh))
    nxt = solve_mixed_Laplace(h, tau, rhs, rhs_is_lambda=False)
    if l2_norm(prev - nxt, prev - nxt, h, h) <= tol or (n >= Nt and not␣
 ↪hard_tol):
      prev = nxt
      break
    else:
      prev = nxt
      n += 1
  return prev

#N = 128
h = np.pi / N
xh = np.linspace(0, np.pi, N + 1)[1:-1]
yh = np.linspace(0, np.pi - h, N) + 0.5 * h
Xh, Yh = np.meshgrid(xh, yh)
c_min_energy = minimize_energy(np.zeros_like(Xh), tau, cp, Lapl_cp, W_dc_np,␣
 ↪tol, Nt, Xh, Yh, np.pi / N, hard_tol=False)
print(f"Energie Minimierer: {derive_energy(y_arr, h, c_min_energy, cp(Xh, Yh),␣
 ↪W_np, cl_np)} +++ Energie cp: {derive_energy(y_arr, h, np.zeros_like(cp(Xh,␣
 ↪Yh)), cp(Xh, Yh), W_np, cl_np)}")
```

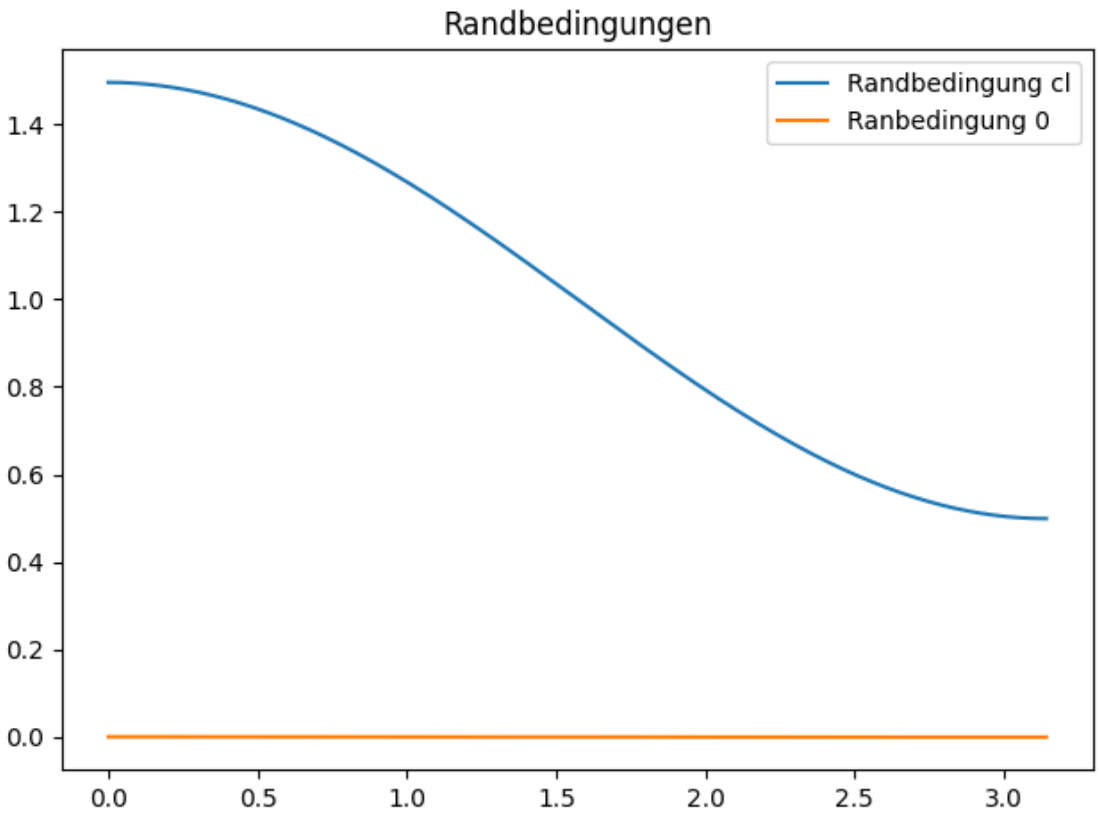
```
Energie Minimierer: 3.703671520594129 +++ Energie cp: 6.959833873422543
```

Plotten Sie den berechneten Minimierer über Ihrem Gitter. Sie können dafür wieder die Funktion `plt.plot_surface` verwenden.

[237]:
```
c_min_energy_inh = c_min_energy + cp(Xh, Yh)
fig, ax = plt.subplots(1, 1, figsize=(4, 4), subplot_kw={"projection": "3d"})
ax.plot_surface(Xh, Yh, c_min_energy_inh, cmap=cm.jet)
ax.set_xlabel("x")
ax.set_ylabel("y")
```

```
ax.set_zlabel("cmin")
plt.title("c minimale Energie")
plt.show()
plt.plot(yh, c_min_energy_inh[:,0], label="Randbedingung cl")
plt.plot(yh, c_min_energy_inh[:,-1], label="Ranbedingung 0")
plt.title("Randbedingungen")
plt.legend()
plt.show()
```




Randbedingungen
Randbedingung cl
Ranbedingung 0
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0