notebook_4_1

July 8, 2025

1 Notebook 4: Lineare finite Elemente in 2D

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

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

```
conda install -c conda-forge meshio
```

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.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import meshio
import scipy
import scipy.sparse
import ipytest
ipytest.autoconfig()
```

2 Aufgabe 1 - Gitter importieren

3/3

Machen Sie sich mit dem Paket `meshio` vertraut. Lesen Sie damit `rectangle_lvl0.msh` ein.

[3]:
```
mesh = meshio.read(r"C:\Users\Li\notebook\meshes\rectangle_lvl0.msh")

print(mesh)
```

```
<meshio mesh object>
  Number of points: 274
  Number of cells:
    line: 20
    line: 10
    line: 20
    line: 10
    triangle: 486
  Cell sets: gmsh:bounding_entities
```

```
Point data: gmsh:dim_tags
Cell data: gmsh:physical, gmsh:geometrical
```

Sie können im Folgenden die Funktion `get_boundary_nodes` verwenden, um auf die Randknoten eines eingelesenen Gitters zuzugreifen. Diese Information haben wir für Sie im Gitter hinterlegt, bei anderen Gittern muss diese Information nicht zwangsläufig vorhanden sein.

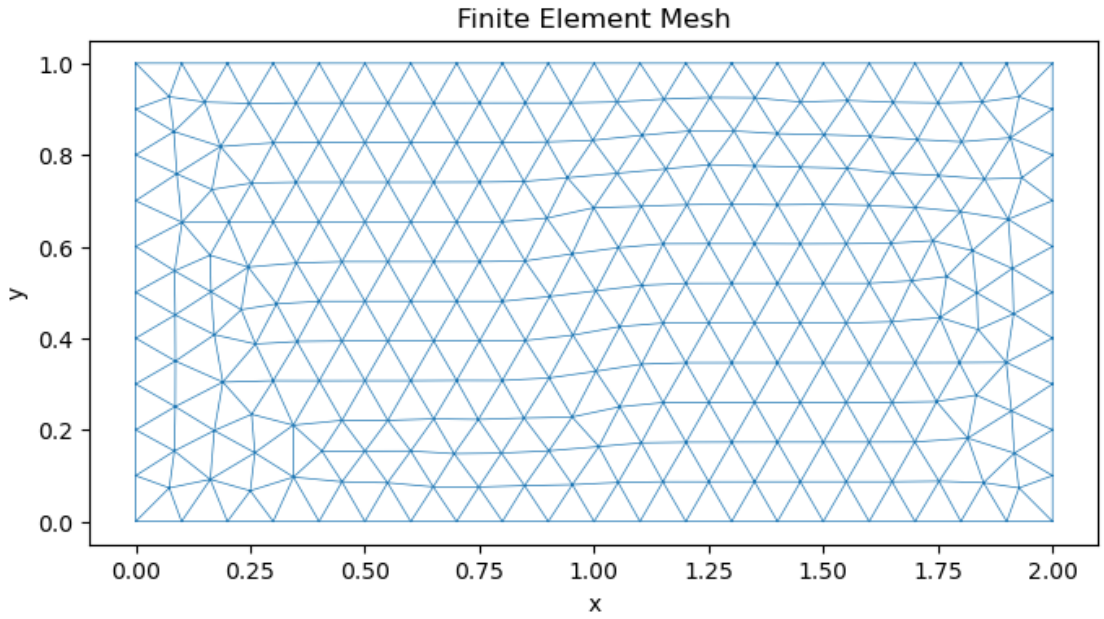
[4]:
```python
def get_boundary_nodes(mesh, boundary_tag=1):
    lines = mesh.cells_dict.get("line", [])
    tags = mesh.cell_data_dict.get("gmsh:physical", {}).get("line", [])
    if not len(lines) or not len(tags):
        raise ValueError("No line elements or physical tags found in mesh.")

    nodes = lines[np.array(tags) == boundary_tag]
    return np.unique(nodes)

# boundary_nodes = get_boundary_nodes(mesh, boundary_tag=1)
# print(f"Number of boundary nodes: {len(boundary_nodes)}")
# print("Boundary node indices:", boundary_nodes)
```

Legen Sie sich die Daten des Gitters in geeigneter Art und Weise zurecht. Orientieren Sie sich dabei an Algorithmus 15.1 im Skript.

[5]:
```python
points_3d = mesh.points
points = points_3d[:, :2]
cell_dofmap = mesh.cells_dict["triangle"]
boundary_nodes = get_boundary_nodes(mesh, boundary_tag=1)

# print('Points:\n', points)
# print('Cell Dofmap:\n', cell_dofmap)
# print('Boundary nodes:\n', boundary_nodes)
```

Plotten Sie das Gitter. Sie können zum Beispiel die Funktion `matplotlib.tri.Triangulation` verwenden.

[6]:
```python
import matplotlib.tri as mtri
triang = mtri.Triangulation(points[:, 0], points[:, 1], triangles=cell_dofmap)

# Plot the triangulation
plt.figure(figsize=(8, 8))
plt.triplot(triang, linewidth=0.5)  # Plot mesh edges
plt.gca().set_aspect('equal')        # Equal scaling on both axes
plt.title('Finite Element Mesh')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



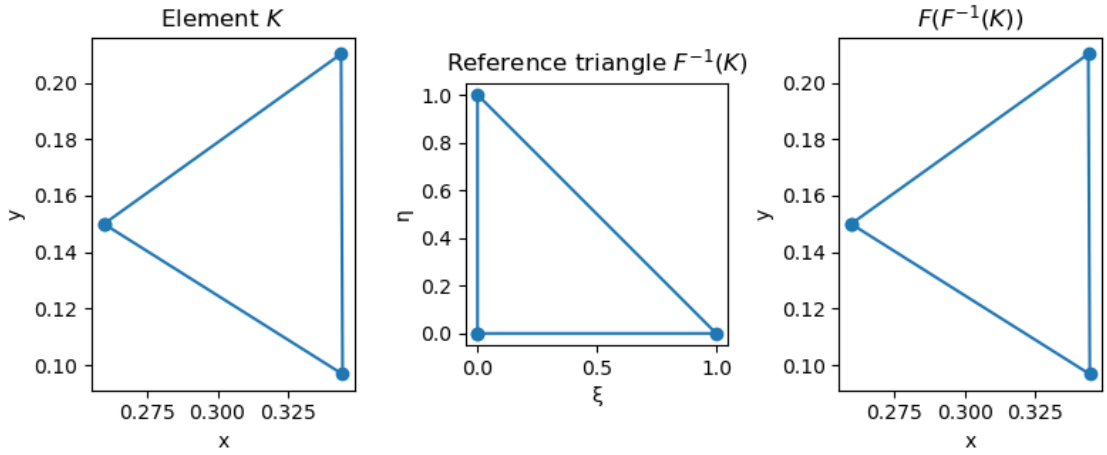
Implementieren Sie die affine Abbildung $F\colon \widehat{K} \to K$ vom Referenz-Dreieck auf ein Dreick der Triangulierung, sowie die Inverse F^{-1}. Plotten Sie ein beliebiges Element K der Triangulierung `rectangle_lvl0.msh`, die inverse Transformation $F^{-1}(K)$, sowie $F(F^{-1}(K))$ in einer Abbildung.

[7]:
```
def map_from_reference_triangle(a1, a2, a3):
    # Build the affine map F: \hat K -> K
    # A · [ , ]^T + a1 = x
    A = np.column_stack((a2 - a1, a3 - a1))
    def F(xi):
        # xi: array of shape (..., 2)
        return a1 + xi.dot(A.T)
    return F
```

[8]:
```
def map_to_reference_triangle(a1, a2, a3):
    # Build the inverse map F^{-1}: K -> \hat K
    A = np.column_stack((a2 - a1, a3 - a1))
    invA = np.linalg.inv(A)
    def Finv(x):
        # x: array of shape (..., 2)
        return (x - a1).dot(invA.T)
    return Finv
```

[9]:
```
# Select the first cell in the triangulation
cell = cell_dofmap[0]
a1, a2, a3 = points[cell[0]], points[cell[1]], points[cell[2]]

# Create the mapping functions
```

```
F = map_from_reference_triangle(a1, a2, a3)
Finv = map_to_reference_triangle(a1, a2, a3)

# Define reference triangle vertices
ref_tri = np.array([[0.0, 0.0],
                    [1.0, 0.0],
                    [0.0, 1.0]])

# 1. Original element K
K = np.vstack([a1, a2, a3, a1])  # close the loop

# 2. Reference triangle F^{-1}(K)
K_ref = np.vstack([ref_tri, ref_tri[0]])  # close the loop

# 3. Mapped back triangle F(F^{-1}(K))
mapped_back = F(Finv(np.vstack([a1, a2, a3])))
mapped_back = np.vstack([mapped_back, mapped_back[0]])  # close the loop

# Plotting the three triangles side by side
fig, axes = plt.subplots(1, 3, figsize=(8, 5))

# Plot original element K
axes[0].plot(K[:, 0], K[:, 1], '-o')
axes[0].set_title('Element $K$')
axes[0].set_aspect('equal')
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')

# Plot reference triangle
axes[1].plot(K_ref[:, 0], K_ref[:, 1], '-o')
axes[1].set_title('Reference triangle $F^{-1}(K)$')
axes[1].set_aspect('equal')
axes[1].set_xlabel(' ')
axes[1].set_ylabel(' ')

# Plot F(F^{-1}(K))
axes[2].plot(mapped_back[:, 0], mapped_back[:, 1], '-o')
axes[2].set_title('$F(F^{-1}(K))$')
axes[2].set_aspect('equal')
axes[2].set_xlabel('x')
axes[2].set_ylabel('y')

plt.tight_layout()
plt.show()
```



3 Aufgabe 2 - Finite Elemente Matrizen assemblieren

Im Folgenden wollen wir die Gleichung

$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) &= f \quad \text{in} \quad \Omega = (0,2) \times (0,1), \\ u &= 0 \quad \text{auf} \quad \partial\Omega \end{aligned}$$

mit $f(x,y) = \sin(4\pi(x+y)) \cdot (x+1)^3$ und $\kappa = 1$ mit der Methode der Finiten Elemente lösen. Wir beschränken uns dabei auf lineare finite Elemente auf Dreiecksgittern.

[10]:
```
f = lambda x: np.sin(4 * np.pi * (x[:, 0] + x[:, 1])) * (x[:, 0] + 1) ** 3
kappa = lambda x: 1
```

3.0.1 Quadraturregeln auf Dreiecken

Vervollständigen Sie die Funktion `get_quadrature`, welche die Quadraturgewichte und Quadraturpunkte auf dem Referenzdreieck zurückgibt. Implementieren Sie die beiden Quadraturformeln auf Dreiecken, die Polynome vom Grad 1 und Grad 2 exakt integrieren.

[11]:
```
def get_quadrature(degree=1):
    if degree == 1:
        # 1-point quadrature (centroid rule)
        # integrates all linear functions exactly
        qp = np.array([[1/3, 1/3]])  # quadrature point ( , )
        qw = np.array([1/2])         # weight = area of reference triangle
    elif degree == 2:
        # 3-point quadrature (degree-2 exact)
        # integrates all quadratics exactly
        qp = np.array([
            [1/6, 1/6],
```

```
            [2/3, 1/6],
            [1/6, 2/3]
        ])
        qw = np.array([1/6, 1/6, 1/6])  # each weight = area/3 = 1/6
    else:
        raise ValueError("Only degree=1 or 2 are supported.")
    return qp, qw

#
# qp1, qw1 = get_quadrature(degree=1)
# qp2, qw2 = get_quadrature(degree=2)
# print("Degree 1 quadrature points:", qp1, "weights:", qw1)
# print("Degree 2 quadrature points:", qp2, "weights:", qw2)
```

3.0.2 Basisfunktionen auf dem Referenzelement

Vervollständigen Sie die Funktionen `phiref` und `Dphiref`, welche die Basisfunktionen sowie die Gradienten auf dem Referenzdreieck auswerten.

[12]:
```
def phiref(local_index, x):
    x = np.array(x)
    # Single point
    if x.ndim == 1:
        xi, eta = x
        if local_index == 0:
            return 1 - xi - eta
        elif local_index == 1:
            return xi
        elif local_index == 2:
            return eta
        else:
            raise ValueError(f"Invalid local_index {local_index}")
    # Multiple points
    xi = x[:, 0]
    eta = x[:, 1]
    if local_index == 0:
        return 1 - xi - eta
    elif local_index == 1:
        return xi
    elif local_index == 2:
        return eta
    else:
        raise ValueError(f"Invalid local_index {local_index}")

def Dphiref(local_index, x=None):
    # gradients in (d/d , d/d ) for phi1, phi2, phi3
    grads = {
```

```
        0: np.array([-1.0, -1.0]),  # grad of (1 -  - )
        1: np.array([ 1.0,  0.0]),  # grad of
        2: np.array([ 0.0,  1.0])   # grad of
    }
    if local_index not in grads:
        raise ValueError(f"Invalid local_index {local_index}")
    g = grads[local_index]
    if x is None:
        return g
    # tile gradient for each evaluation point
    x = np.array(x)
    if x.ndim == 1:
        return g[np.newaxis, :]
    else:
        return np.tile(g, (x.shape[0], 1))

#
# pts = np.array([[0.2, 0.1], [0.5, 0.4]])
# for i in range(3):
#     print(f"phi_{i}(pts) =", phiref(i, pts), " grad =", Dphiref(i))
```

3.0.3 Assemblierung der Steifigkeitsmatrix

Vervollständigen Sie die Funktion `local_stiffness`, welche die lokalen Bestandteil der Steifigkeitsmatrix zu einem Element K mit den Eckpunkten a_1, a_2, a_3 assembliert.

[13]:
```
def local_stiffness(kappa, a1, a2, a3, quad_degree):
    # Get reference quadrature
    qp_ref, wq = get_quadrature(degree=quad_degree)  # qp_ref shape (nq,2), wq␣
 ↪shape (nq,)

    # Build affine map: x = F( ) = a1 + A · [ ; ]
    A = np.column_stack((a2 - a1, a3 - a1))           # 2×2 Jacobian matrix
    detJ = np.linalg.det(A)
    invA = np.linalg.inv(A)

    # Compute physical gradients of basis functions:  _i^K = (inv(A))^T  _i^ref
    gradsK = np.zeros((3, 2))
    for i in range(3):
        grad_ref = Dphiref(i)            # shape (2,)
        gradsK[i, :] = invA.T.dot(grad_ref)

    # Initialize local stiffness
    localK = np.zeros((3, 3))

    # Loop over quadrature points
    for (xi, eta), w in zip(qp_ref, wq):
```

```
        # Map to physical point
        x_phys = a1 + A.dot(np.array([xi, eta]))
        # Evaluate kappa at this point; ensure scalar
        k_val = kappa(x_phys if x_phys.ndim == 2 else x_phys[np.newaxis, :])
        if isinstance(k_val, np.ndarray):
            k_val = float(k_val.flat[0])
        # Physical quadrature weight
        w_phys = w * abs(detJ)
        # Assemble contributions
        for i in range(3):
            for j in range(3):
                localK[i, j] += k_val * w_phys * np.dot(gradsK[i], gradsK[j])

    return localK

#
# a1, a2, a3 = points[cell_dofmap[0][0]], points[cell_dofmap[0][1]],␣
 ↪points[cell_dofmap[0][2]]
# Kloc = local_stiffness(kappa, a1, a2, a3, quad_degree=2)
# print("Local stiffness matrix on element 0:\n", Kloc)
```

Zuvor geschriebene Fkt verwenden?

Vervollständigen Sie die globale Assemblierungsroutine für die Steifigkeitsmatrix.

[18]:
```
def assemble_stiffness(points, cell_dofmap, kappa, quad_degree=1):
    n_points = points.shape[0]
    # Lists to build COO-format sparse matrix
    rows = []
    cols = []
    data = []

    # Loop over all elements
    for cell in cell_dofmap:
        # Vertex coordinates of current element
        a1, a2, a3 = points[cell[0]], points[cell[1]], points[cell[2]]
        # Compute local stiffness matrix (3×3)
        Kloc = local_stiffness(kappa, a1, a2, a3, quad_degree)

        # Scatter into global matrix
        for i_local, i_global in enumerate(cell):
            for j_local, j_global in enumerate(cell):
                rows.append(i_global)
                cols.append(j_global)
                data.append(Kloc[i_local, j_local])

    # Assemble COO and convert to CSR (sums duplicate entries)
    K_global = scipy.sparse.coo_matrix((data, (rows, cols)),
                                       shape=(n_points, n_points))
```

```
    return K_global.tocsr()

#
K = assemble_stiffness(points, cell_dofmap, kappa, quad_degree=2)
print("Global stiffness matrix shape:", K.shape)
print("Number of nonzeros:", K.nnz)
```

```
Global stiffness matrix shape: (274, 274)
Number of nonzeros: 1792
```

Testen Sie Ihre assemblierte Steifigkeitsmatrix auf Symmetrie.

[20]:
```
diff = K - K.T

# Since K is sparse, examine the nonzero entries of the difference
# Convert to COO to access the data array
diff_coo = diff.tocoo()
max_abs_diff = np.max(np.abs(diff_coo.data)) if diff_coo.data.size > 0 else 0.0

print(f"Maximum absolute difference between K and K^T: {max_abs_diff:.2e}")

# Check against a numerical tolerance
tol = 1e-12
if max_abs_diff < tol:
    print("The stiffness matrix is symmetric within tolerance!")
else:
    print("The stiffness matrix is NOT symmetric!")
```

```
Maximum absolute difference between K and K^T: 0.00e+00
The stiffness matrix is symmetric within tolerance!
```

3.0.4 Assemblierung der Massematrix

Vervollständigen Sie die Funktion `local_mass`, welche die lokalen Bestandteil der Massematrix zu einem Element K mit den Eckpunkten a_1, a_2, a_3 assembliert.

[21]:
```
def local_mass(a1, a2, a3, quad_degree):
    # Get quadrature points and weights on reference triangle
    qp_ref, wq = get_quadrature(degree=quad_degree)  # qp_ref shape (nq,2), wq␣
 ↪shape (nq,)

    # Build affine map Jacobian A and its determinant
    A = np.column_stack((a2 - a1, a3 - a1))  # 2×2
    detJ = abs(np.linalg.det(A))

    # Precompute reference basis at all quadrature points
    nq = qp_ref.shape[0]
    phi_vals = np.zeros((3, nq))
    for i in range(3):
```

```
        # phiref can take array of shape (nq,2)
        phi_vals[i, :] = phiref(i, qp_ref)

    # Initialize local mass matrix
    Mloc = np.zeros((3, 3))

    # Loop quadrature points
    for k in range(nq):
        w_phys = wq[k] * detJ
        for i in range(3):
            for j in range(3):
                Mloc[i, j] += w_phys * phi_vals[i, k] * phi_vals[j, k]

    return Mloc

#
# cell = cell_dofmap[0]
# a1, a2, a3 = points[cell[0]], points[cell[1]], points[cell[2]]
# Mloc = local_mass(a1, a2, a3, quad_degree=2)
# print("Local mass matrix on element 0:\n", Mloc)
```

Vervollständigen Sie die globale Assemblierungsroutine für die Massematrix.

[22]:
```
# assemble mass
def assemble_mass(points, cell_dofmap, quad_degree=2):
    n_points = points.shape[0]
    # Lists to build a COO-format sparse matrix
    rows = []
    cols = []
    data = []

    # Loop over all elements
    for cell in cell_dofmap:
        # Extract vertex coordinates for this element
        a1, a2, a3 = points[cell[0]], points[cell[1]], points[cell[2]]
        # Compute local mass matrix (3×3)
        Mloc = local_mass(a1, a2, a3, quad_degree)

        # Scatter local entries into global matrix
        for i_local, i_global in enumerate(cell):
            for j_local, j_global in enumerate(cell):
                rows.append(i_global)
                cols.append(j_global)
                data.append(Mloc[i_local, j_local])

    # Assemble COO and convert to CSR (duplicate entries are summed)
    M_global = scipy.sparse.coo_matrix((data, (rows, cols)),
```

```
                                        shape=(n_points, n_points))
    return M_global.tocsr()

#          :
M = assemble_mass(points, cell_dofmap, quad_degree=2)
print("Global mass matrix shape:", M.shape)
print("Number of nonzeros:", M.nnz)
```

```
Global mass matrix shape: (274, 274)
Number of nonzeros: 1792
```

Testen Sie Ihre assemblierte Massematrix auf Symmetrie.

[23]:
```
# Test symmetry of the global mass matrix M
diffM = M - M.T
diffM_coo = diffM.tocoo()
max_abs_diff_M = np.max(np.abs(diffM_coo.data)) if diffM_coo.data.size > 0 else␣
 ↪0.0

print(f"Maximum absolute difference between M and M^T: {max_abs_diff_M:.2e}")

tol = 1e-12
if max_abs_diff_M < tol:
    print("The mass matrix is symmetric within tolerance!")
else:
    print("The mass matrix is NOT symmetric!")
```

```
Maximum absolute difference between M and M^T: 2.17e-19
The mass matrix is symmetric within tolerance!
```

3.0.5 Assemblierung des Lastvektors

Vervollständigen Sie die Funktion `local_load`, welche die lokalen Bestandteil des Lastvektors zu einem Element K mit den Eckpunkten a_1, a_2, a_3 assembliert.

[24]:
```
def local_load(rhs, a1, a2, a3, quad_degree):
    # Get quadrature points and weights on the reference triangle
    qp_ref, wq = get_quadrature(degree=quad_degree)  # qp_ref: (nq,2), wq: (nq,)

    # Build affine map Jacobian A and its determinant
    A = np.column_stack((a2 - a1, a3 - a1))  # 2×2
    detJ = abs(np.linalg.det(A))

    # Precompute reference basis at quadrature points
    nq = qp_ref.shape[0]
    phi_vals = np.zeros((3, nq))
    for i in range(3):
        phi_vals[i, :] = phiref(i, qp_ref)
```

```
    # Initialize local load vector
    b_loc = np.zeros(3)

    # Loop over quadrature points
    for k in range(nq):
        xi, eta = qp_ref[k]
        w_phys = wq[k] * detJ
        # Map to physical point
        x_phys = a1 + A.dot(np.array([xi, eta]))
        # Evaluate RHS at this point
        f_val = rhs(x_phys[np.newaxis, :])
        if isinstance(f_val, np.ndarray):
            f_val = float(f_val.flat[0])
        # Accumulate contributions
        for i in range(3):
            b_loc[i] += w_phys * f_val * phi_vals[i, k]

    return b_loc

#
cell = cell_dofmap[0]
a1, a2, a3 = points[cell[0]], points[cell[1]], points[cell[2]]
b_loc = local_load(f, a1, a2, a3, quad_degree=2)
print("Local load vector on element 0:\n", b_loc)
```

```
Local load vector on element 0:
 [-0.00182027 -0.00165179 -0.0003838 ]
```

Vervollständigen Sie die globale Assemblierungsroutine für den Lastvektor.

[25]:
```
def assemble_load(rhs, points, cell_dofmap, quad_degree=2):
    n_points = points.shape[0]
    b_global = np.zeros(n_points)

    # Loop over all elements
    for cell in cell_dofmap:
        # Get the coordinates of the triangle vertices
        a1 = points[cell[0]]
        a2 = points[cell[1]]
        a3 = points[cell[2]]

        # Compute local load vector on this element
        b_loc = local_load(rhs, a1, a2, a3, quad_degree)

        # Scatter into the global load vector
        for i_local, i_global in enumerate(cell):
            b_global[i_global] += b_loc[i_local]
```

```
    return b_global

#
b = assemble_load(f, points, cell_dofmap, quad_degree=2)
print("Global load vector length:", b.shape[0])
print("First 10 entries of b:", b[:10])
```

```
Global load vector length: 274
First 10 entries of b: [ 1.75410643e-03  3.57197016e-04 -4.10576816e-02
4.42186800e-05
  4.55650335e-03  1.65591609e-03 -6.78005739e-03 -8.55021014e-03
  4.73615433e-03  1.44737422e-02]
```

3.0.6 Homegene Dirichlet Randbedingungen

Vervollständigen Sie die Funktion `apply_hom_dirichlet_bc`, um homogene Dirichlet Randbedingungen zu berücksichtigen. Orientieren Sie sich an Abschnitt 15.4.6 im Skript. Geben Sie die Matrix A_{hom}, den Lastvektor b_{hom} und die Diagonalmatrix Π zurück.

[26]:
```
def apply_hom_dirichlet_bc(mesh, A, b=None, boundary_tag=1):
    # get boundary node indices
    boundary_nodes = get_boundary_nodes(mesh, boundary_tag)
    n = A.shape[0]

    # build Pi = diag(pi_i), pi_i = 0 on boundary, 1 otherwise
    mask = np.ones(n, dtype=float)
    mask[boundary_nodes] = 0.0
    Pi = scipy.sparse.diags(mask, 0, format='csr')

    # compute A_hom = Pi A Pi + (I - Pi) to fix Dirichlet rows/cols
    I = scipy.sparse.identity(n, format='csr')
    A_hom = Pi.dot(A.dot(Pi)) + (I - Pi)

    # modify load vector
    if b is not None:
        b_hom = Pi.dot(b)
    else:
        b_hom = None

    return A_hom, b_hom, Pi

# A = global stiffness matrix (CSR), b = global load vector, mesh already␣
 ↪defined
Ahom, bhom, fill_bc = apply_hom_dirichlet_bc(mesh, K, b, boundary_tag=1)
```

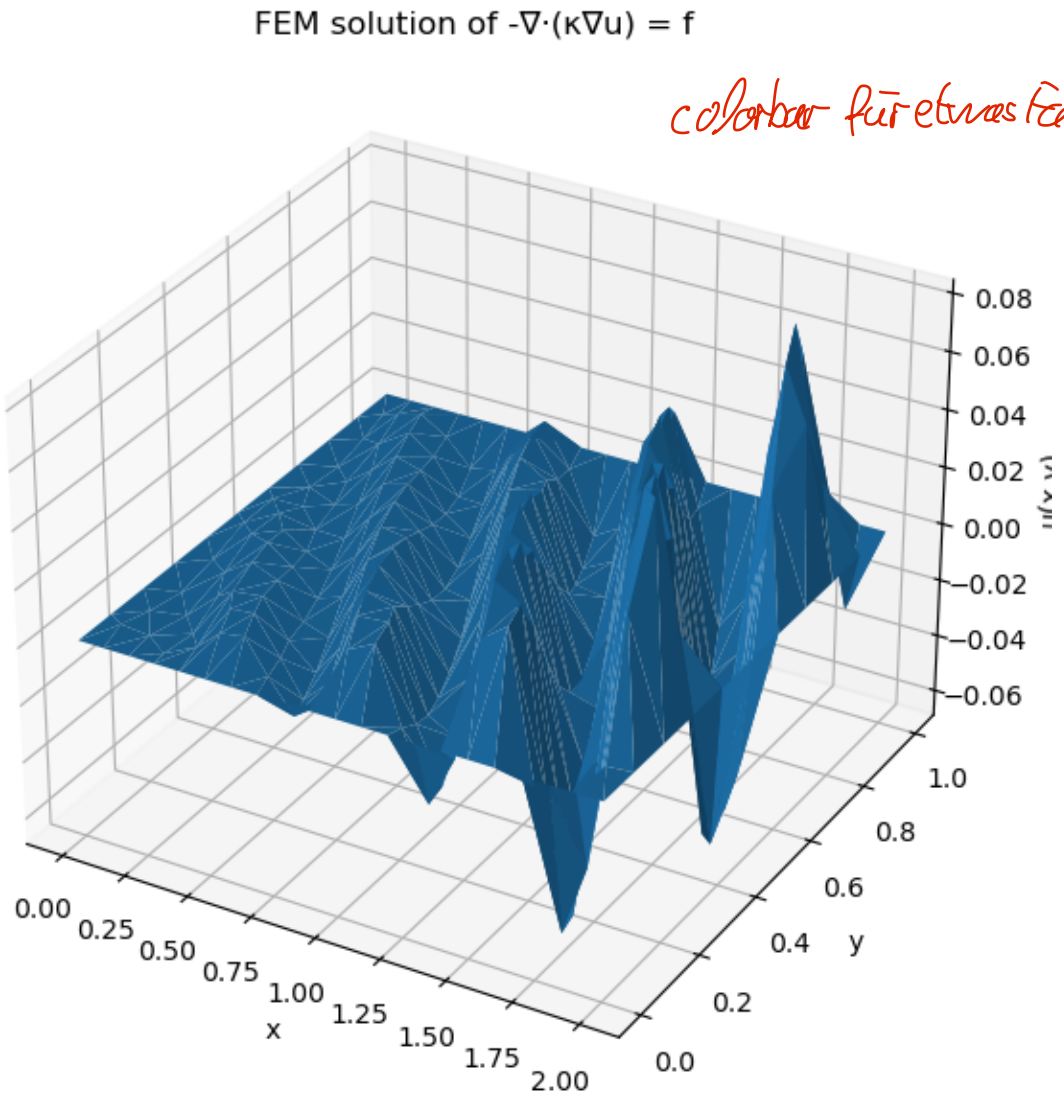
Lösen Sie das homegene System $A_{hom}u = b_{hom}$ und befüllen anschließend die Knoten der homogenen Randbedingungen.

[27]:
```python
import scipy.sparse.linalg as spla
# Solve the modified system Ahom u = bhom
u = spla.spsolve(Ahom, bhom)

# Enforce homogeneous Dirichlet values (should already be zero from Ahom)
boundary_nodes = get_boundary_nodes(mesh, boundary_tag=1)
u[boundary_nodes] = 0.0

# Optional check: maximum absolute value on boundary nodes
max_bc = np.max(np.abs(u[boundary_nodes]))
print(f"Max absolute value at Dirichlet nodes (should be 0): {max_bc:.2e}")
```

```
Max absolute value at Dirichlet nodes (should be 0): 0.00e+00
```

Plotten Sie nun Ihre Lösung, benutzen Sie dafür die Funktion `plot_trisurf`.

[28]:
```python
# Create a triangulation for plotting
triang = mtri.Triangulation(points[:, 0], points[:, 1], triangles=cell_dofmap)

# Set up 3D figure
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Plot the solution over the mesh
# u is the FEM solution array of length N
surf = ax.plot_trisurf(triang, u, linewidth=0.2, antialiased=True)

# Add labels and title
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title('FEM solution of - ·( u) = f')

plt.tight_layout()
plt.show()
```



4 Aufgabe 3 - Konvergenzplot

Im Folgenden wollen wir unsere Assemblierung mit Hilfe eines Konvergenzplots testen. Nutzen Sie dafür die exakte Lösung $u(x, y) = \sin(\pi x)^2 \sin(\pi y)^2$ von

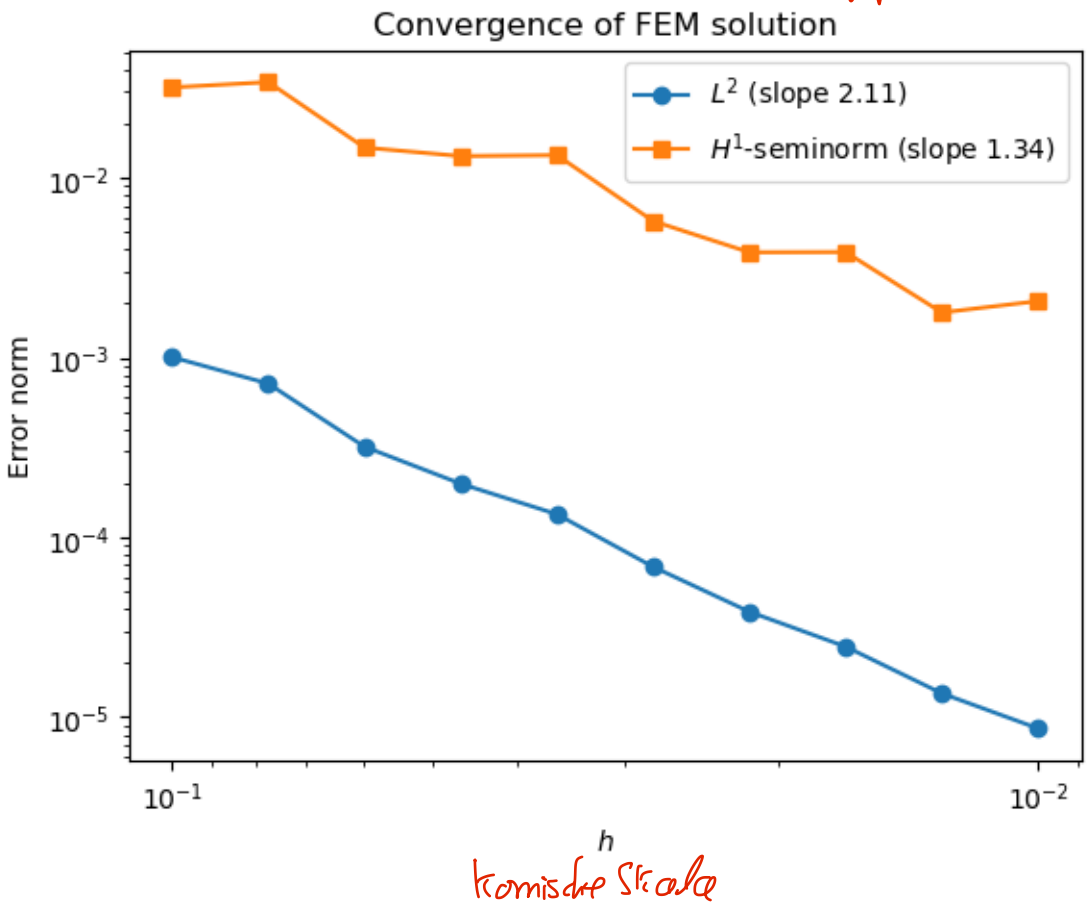
$$\begin{aligned} -\nabla \cdot (\kappa \nabla u) &= f \qquad \text{in} \qquad \Omega = (0, 2) \times (0, 1), \\ u &= 0 \qquad \text{auf} \qquad \partial\Omega \end{aligned}$$

mit $\kappa(x, y) = \cos(\pi x)\cos(\pi y) + 2$. Berechnen Sie die rechte Seite f so, dass u obige Gleichung löst. Nutzen Sie für Ihren Konvergenzplot die Gitter `meshes/rectangle_lvl0.msh` bis `meshes/rectangle_lvl9.msh`. Die gmsh vorgegebenen Werte für h für die verschiedenen Gitterlevels finden Sie im vorliegenden array `hs`. Messen Sie Ihren Fehler gegen die interpolierte exakte Lösung $I_h u$ in der L^2 wie auch der H^1 Norm. Welche Konvergenzordnungen beobachten Sie?

[29]:
```
hs = np.geomspace(0.1, 0.01, 10)
```

[31]:
```
# Exact solution and its derivatives
def u_exact(x):
    """Exact solution u(x,y) = sin^2(pi x) sin^2(pi y) at points x (N×2)."""
    return np.sin(np.pi * x[:,0])**2 * np.sin(np.pi * x[:,1])**2

# Right-hand side f derived from -div(kappa grad u)
def f_rhs(x):
    """
    Compute f(x,y) so that -div(kappa grad u) = f, where
      u = sin^2(pi x) sin^2(pi y),
      kappa = cos(pi x) cos(pi y) + 2.
    x may be shape (2,) or (N,2). Returns array of shape (N,).
    """
    x = np.atleast_2d(x)
    X = x[:,0]; Y = x[:,1]
    # derivatives of u
    u_x  =  np.pi * np.sin(2*np.pi*X) * np.sin(np.pi*Y)**2
    u_y  =  np.pi * np.sin(2*np.pi*Y) * np.sin(np.pi*X)**2
    u_xx = 2*(np.pi**2) * np.cos(2*np.pi*X) * np.sin(np.pi*Y)**2
    u_yy = 2*(np.pi**2) * np.cos(2*np.pi*Y) * np.sin(np.pi*X)**2
    # kappa and its derivatives
    kappa_val = np.cos(np.pi*X)*np.cos(np.pi*Y) + 2
    kx = -np.pi * np.sin(np.pi*X) * np.cos(np.pi*Y)
    ky = -np.pi * np.cos(np.pi*X) * np.sin(np.pi*Y)
    # compute divergence term
    term_x = kx * u_x  +  kappa_val * u_xx
    term_y = ky * u_y  +  kappa_val * u_yy
    f = -(term_x + term_y)
    return f

# Diffusion coefficient
kappa_fun = lambda x: (np.cos(np.pi*x[:,0])*np.cos(np.pi*x[:,1]) + 2)

# Mesh levels and mesh-sizes
levels = range(10)
hs = np.geomspace(0.1, 0.01, 10)

err_L2 = []
err_H1_semi = []

for level, h in zip(levels, hs):
    # Read mesh and setup dofmap
    mesh = meshio.read(rf"C:\Users\Li\notebook\meshes\rectangle_lvl{level}.msh")
    pts3 = mesh.points
    points = pts3[:, :2]
```

```
    cell_dofmap = mesh.cells_dict["triangle"]
    boundary_nodes = get_boundary_nodes(mesh, boundary_tag=1)

    # Assemble global matrices and RHS
    K = assemble_stiffness(points, cell_dofmap, kappa_fun, quad_degree=2)
    M = assemble_mass(points, cell_dofmap, quad_degree=2)
    b = assemble_load(f_rhs, points, cell_dofmap, quad_degree=2)

    # Apply BC and solve
    Ahom, bhom, Pi = apply_hom_dirichlet_bc(mesh, K, b, boundary_tag=1)
    u_h = spla.spsolve(Ahom, bhom)
    u_h[boundary_nodes] = 0.0

    # Compute errors against nodal interpolant of exact solution
    u_e = u_exact(points)
    err = u_e - u_h
    eL2 = np.sqrt(err @ (M @ err))
    eH1 = np.sqrt(err @ (K @ err))
    err_L2.append(eL2)
    err_H1_semi.append(eH1)

err_L2 = np.array(err_L2)
err_H1_semi = np.array(err_H1_semi)

rate_L2 = np.polyfit(np.log(hs), np.log(err_L2), 1)[0]
rate_H1 = np.polyfit(np.log(hs), np.log(err_H1_semi), 1)[0]

plt.figure()
plt.loglog(hs, err_L2, 'o-', label=f'$L^2$ (slope {rate_L2:.2f})')
plt.loglog(hs, err_H1_semi, 's-', label=f'$H^1$-seminorm (slope {rate_H1:.2f})')
plt.gca().invert_xaxis() muss nicht sein, natürlicher andersrum
plt.xlabel('$h$')
plt.ylabel('Error norm')
plt.legend()
plt.title('Convergence of FEM solution')
plt.show()

print(f"Observed convergence order: L2 = {rate_L2:.2f}, H1-seminorm = {rate_H1:.
 ↪2f}")
```




```
Observed convergence order: L2 = 2.11, H1-seminorm = 1.34
```

<- Platz für Ihre Antwort ->

Die Auswertung ergibt für den L^2-Fehler eine Steigung von ca. 2,11, was gut mit dem theoretischen $O(h^2)$ übereinstimmt; für den H^1-Seminorm beträgt die Steigung etwa 1,34 und liegt damit geringfügig über 1.

Lineare Konvergenz was mit der in der Vorlesung angegebenen Konvergenzordnung im Einklag is

5 Aufgabe 4 - Vergleich verschiedener Löser

[35]:
```
from scipy.sparse.linalg import spsolve, splu, cg
import timeit
```

Nutzen Sie das Paket `timeit` um die Zeit zu messen, die verschiedene Löser zum Lösen des linearen Gleichungssystems

$$L_h u_h = b_h$$

mit $L_h = \alpha M + \beta A$ und $\alpha, \beta \geq 0$. Vergleichen Sie - direktes Lösen über `scipy.sparse.linalg.spsolve` - Berechnen der LU-Zerlegung über SuperLU mit `scipy.sparse.linalg.splu` - Lösen mit der berechneten LU-Zerlegung aus SuperLU mit `.solve` - iteratives Lösen mit dem cg-Verfahren ohne Vorkonditionierer - Lösen mit dem cg-Verfahren mit diagonaler (Jacobi) Vorkonditionierung

Messen Sie bei den iterativen Verfahren neben der Zeit auch die Anzahl der benötigten Iterationen (aber außerhalb der Zeitmessung). Sie können dabei `scipy.sparse.linalg.cg` die Funktion `cb` mit

```
n_iterations = 0
def cb(x):
    global n_iterations
    n_iterations +=  1
```

als `callback` übergeben. Was beobachten Sie im Vergleich?

[36]:
```
# Parameters
filename = "meshes/rectangle_lvl9.msh"
alpha, beta = 1.0, 1.0
tol = 1e-8
maxiter = 1000
n_runs = 5  # number of repetitions for timing

# Read mesh and extract data
mesh = meshio.read(filename)
points = mesh.points[:, :2]
cell_dofmap = mesh.cells_dict["triangle"]

# Assemble stiffness and mass matrices, and load vector
K = assemble_stiffness(points, cell_dofmap, kappa_fun, quad_degree=2)
M = assemble_mass(points, cell_dofmap, quad_degree=2)
L = alpha * M + beta * K
b = assemble_load(f_rhs, points, cell_dofmap, quad_degree=2)

# Apply homogeneous Dirichlet BC
L_hom, b_hom, Pi = apply_hom_dirichlet_bc(mesh, L, b, boundary_tag=1)
L_hom = L_hom.tocsr()
L_csc = L_hom.tocsc()

# 1) Direct solve with spsolve
t_direct = timeit.timeit(lambda: spsolve(L_hom, b_hom),
                         number=n_runs)

# 2) LU factorization time (SuperLU)
t_lu_fact = timeit.timeit(lambda: splu(L_csc),
                          number=n_runs)
# Precompute LU for solve timing
lu = splu(L_csc)
```

```
t_lu_solve = timeit.timeit(lambda: lu.solve(b_hom),
                           number=n_runs)

# 3) CG without preconditioner: count iterations via callback
n_it_np = 0
def cb_np(xk):
    global n_it_np
    n_it_np += 1

n_it_np = 0
_ = cg(L_hom, b_hom, callback=cb_np, atol=tol, maxiter=maxiter)[0]
iters_np = n_it_np

t_cg_np = timeit.timeit(lambda: cg(L_hom, b_hom, atol=tol, maxiter=maxiter)[0],
                        number=n_runs)

# 4) CG with Jacobi preconditioning
diag = L_hom.diagonal()
M_prec = scipy.sparse.diags(1.0 / diag)

n_it_pc = 0
def cb_pc(xk):
    global n_it_pc
    n_it_pc += 1

n_it_pc = 0
_ = cg(L_hom, b_hom, M=M_prec, callback=cb_pc, atol=tol, maxiter=maxiter)[0]
iters_pc = n_it_pc

t_cg_pc = timeit.timeit(lambda: cg(L_hom, b_hom, M=M_prec, atol=tol,␣
 ↪maxiter=maxiter)[0],
                        number=n_runs)

# Print results (average per run)
print("Solver timings (average over runs):")
print(f"  spsolve:             {t_direct/n_runs:.3e} s")
print(f"  LU factorization:    {t_lu_fact/n_runs:.3e} s")
print(f"  LU solve (reuse LU): {t_lu_solve/n_runs:.3e} s")
print(f"  CG (no prec):        {t_cg_np/n_runs:.3e} s, iterations = {iters_np}")
print(f"  CG (Jacobi prec):    {t_cg_pc/n_runs:.3e} s, iterations = {iters_pc}")
```

```
Solver timings (average over runs):
  spsolve:             4.522e-01 s
  LU factorization:    4.484e-01 s
  LU solve (reuse LU): 9.278e-03 s
  CG (no prec):        4.903e-01 s, iterations = 353
  CG (Jacobi prec):    4.025e-01 s, iterations = 277
```

<- Platz für Ihre Antwort ->

Direktlösung vs. LU-Zerlegung: Die Laufzeit von spsolve stimmt nahezu mit der reinen LU-Zerlegung überein, was zeigt, dass der Aufwand überwiegend in der Faktorisierung steckt.

Nach einmaliger Faktorisierung lässt sich mit demselben LU-Ergebnis der rechte Vektor sehr schnell mehrfach lösen, wodurch die Laufzeit sofort stark abfällt.

CG-Verfahren: In jeder Iteration werden mehrere Sparse-Matrix–Vektor-Multiplikationen sowie einige Vektor-Skalarprodukte ausgeführt, sodass die Gesamtlaufzeit annähernd proportional zur Anzahl der Iterationen ist.

[ ]: