notebook_5 (2)

July 28, 2025

In Notebook 5 wollen wir das Finite Elemente Paket FEniCSx nutzen. Eine Installationsanleitung sowie ein kleines Tutorial finden Sie im file `intro_dolfinx.ipynb`. In diesem Notebook gehen wir davon aus, dass Sie `dolfinx` bereits korrekt installiert haben und die folgenden Importe funktionieren:

[1]:
```
from mpi4py import MPI
from petsc4py import PETSc
import dolfinx as dfx
import dolfinx.fem.petsc
import meshio
import gmsh
import ufl
import numpy as np
import matplotlib.pyplot as plt
import pyvista

dolfinx.__version__  # Es sollte Version 0.9.0 ausgegeben werden
```

[1]:
```
'0.9.0'
```

1 Notebook 5 - Finite Elemente mit FEniCSx

Namen: Friedward Wenzler, Yueheng Li Erreichte Punktzahl: 65/10

2 Aufgabe 1: Poisson-Gleichung

14/14

2.1 a) Konvergenzverhalten in in 2D

In dieser Aufgabe wollen wir

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

mithilfe des FEM-Pakets `FEniCSx` lösen. Legen Sie dazu zunächst mit `gmsh` ein Gitter an. Vervollständigen Sie die Funktion `build_unit_square_mesh(comm, h)` und nutzen Sie innerhalb der Funktion den Aufruf `dolfinx.io.gmshio.model_to_mesh` um ihr Gitter in `FEniCSx` einzulesen.

[2]:
```
from mpi4py import MPI
import dolfinx as dfx
import gmsh
from typing import Tuple, Callable, Union, List

COMM_WORLD = MPI.COMM_WORLD


def build_unit_square_mesh(comm: MPI.Intracomm, h: float) -> Tuple[dfx.mesh.
 ↪Mesh, dfx.mesh.MeshTags, dfx.mesh.MeshTags]:
    gmsh.initialize()
    gmsh.option.setNumber("General.Terminal", 0)

    # Create model
    model = gmsh.model()
    # Create points of the area, all with same granularity h
    a1 = model.geo.addPoint(0.0, 0.0, 0.0, h)
    a2 = model.geo.addPoint(1.0, 0.0, 0.0, h)
    a3 = model.geo.addPoint(1.0, 1.0, 0.0, h)
    a4 = model.geo.addPoint(0.0, 1.0, 0.0, h)
    # Create lines of the area
    l1 = model.geo.addLine(a1, a2)
    l2 = model.geo.addLine(a2, a3)
    l3 = model.geo.addLine(a3, a4)
    l4 = model.geo.addLine(a4, a1)
    # Create plane surface
    cl = model.geo.addCurveLoop([l1, l2, l3, l4]) # defines boundary lines as␣
 ↪loop needed for defining surface
    pl = model.geo.addPlaneSurface([cl])
    # physical group
    model.geo.addPhysicalGroup(1, [l1, l2, l3, l4], tag=1) # define boundary␣
 ↪lines as boundary, for tag of boundary nodes of generated mesh
    model.geo.addPhysicalGroup(2, [pl], tag=0) # define surface, all genrated␣
 ↪nodes
    # Generate the mesh
    model.geo.synchronize()
    model.mesh.generate(2)

    # convert gsmh mesh to dolfinx mesh
    mesh, cell_markers, facet_markers = dfx.io.gmshio.model_to_mesh(
        model, comm, rank=0, gdim=2
    )
    gmsh.finalize()
    return mesh, cell_markers, facet_markers

h = 0.1
mesh,_,_ = build_unit_square_mesh(COMM_WORLD, h)
```

Wir legen uns zunächst ein kleines Testbeispiel an. Wir betrachten als Lösung den Ausdruck

$$u(x, y) = \sin^2(\pi x) \sin^2(\pi y)$$

und als Diffusionskoeffizient wählen wir

$$\kappa(x, y) = \exp(x + y)\,.$$

Bestimmen Sie die rechte Seite so f so, dass u eine Lösung des obigen Problems ist und legen Sie die numpy und ufl Ausdrücke `u_np`, `u_ufl`, `kappa_ufl` und `f_ufl` entsprechend an.

[3]:
```
# symbolic derivation of f for manufactured solution
import sympy as sp
from sympy.tensor.array import derive_by_array

x_sym, y_sym = sp.symbols('x, y')
u_sym = sp.sin(sp.pi * x_sym)**2 * sp.sin(sp.pi * y_sym)**2
kappa_sym = sp.exp(x_sym + y_sym)
grad_u_sym = derive_by_array(u_sym, (x_sym, y_sym))
kappa_grad_u_sym = kappa_sym * grad_u_sym
f_sym = -(sp.diff(kappa_grad_u_sym[0], x_sym) + sp.diff(kappa_grad_u_sym[1],␣
 ↪y_sym))
print(sp.simplify(f_sym))
print(sp.simplify(sp.simplify(f_sym)))
print(sp.simplify(sp.simplify(sp.simplify(f_sym))))
```

```
2*pi*(4*pi*sin(pi*x)**2*sin(pi*y)**2 - pi*sin(pi*x)**2 -
sin(pi*x)*sin(pi*y)*sin(pi*(x + y)) - pi*sin(pi*y)**2)*exp(x + y)
-2*pi*(-4*pi*sin(pi*x)**2*sin(pi*y)**2 + pi*sin(pi*x)**2 +
sin(pi*x)*sin(pi*y)*sin(pi*(x + y)) + pi*sin(pi*y)**2)*exp(x + y)
-2*pi*(-4*pi*sin(pi*x)**2*sin(pi*y)**2 + pi*sin(pi*x)**2 +
sin(pi*x)*sin(pi*y)*sin(pi*(x + y)) + pi*sin(pi*y)**2)*exp(x + y)
```

[4]:
```
u_np = lambda x: np.sin(np.pi * x[0])**2 * np.sin(np.pi * x[1])**2
u_ufl = lambda x: ufl.sin(ufl.pi * x[0])**2 * ufl.sin(ufl.pi * x[1])**2
kappa_ufl = lambda x: ufl.exp(x[0] + x[1])
f_ufl = lambda x: -2 * ufl.pi * (
                        -4 * ufl.pi * ufl.sin(ufl.pi * x[0])**2 * ufl.sin(ufl.
 ↪pi * x[1])**2
                        + ufl.pi * ufl.sin(ufl.pi * x[0])**2
                        + ufl.sin(ufl.pi * x[0]) * ufl.sin(ufl.pi * x[1]) * ufl.
 ↪sin(ufl.pi * (x[0] + x[1]))
                        + ufl.pi * ufl.sin(ufl.pi * x[1])**2
                    ) * ufl.exp(x[0] + x[1])
```

Mit der Funktion `get_all_boundary_dofs` können Sie die benötigten DoFs auf dem Rand auslesen, um später die Randbedingungen zu assemblieren.

[5]:
```python
def get_all_boundary_dofs(V: dfx.fem.FunctionSpace) -> np.ndarray:
    tdim = V.mesh.topology.dim
    fdim = tdim - 1
    V.mesh.topology.create_connectivity(fdim, tdim)
    boundary_facets = dfx.mesh.exterior_facet_indices(V.mesh.topology)
    boundary_dofs = dfx.fem.locate_dofs_topological(V, fdim, boundary_facets)
    return boundary_dofs
```

Orientieren Sie sich an dem Notebook `intro_dolfinx.ipynb`, und schreiben Sie sich eine Methode `solve_Poisson`, welche obiges Problem numerisch löst.

[6]:
```python
def solve_Poisson(mesh: dfx.mesh.Mesh, kappa_ufl: Callable, f_ufl: Callable,␣
↪fem_type : str="Lagrange", fem_degree : int=1) -> dfx.fem.Function:
    V = dfx.fem.functionspace(mesh, (fem_type, fem_degree)) # define finite␣
↪element space, parameters specify the type of finite elements of the lecture

    # homogenous boundary condition
    boundary_dofs = get_all_boundary_dofs(V)
    uB = dfx.fem.Function(V)
    uB.x.array[:] = 0.0
    # with uB.localForm() as loc:
    #     loc.set(0)
    bc = dfx.fem.dirichletbc(uB, boundary_dofs)

    # define u and the test functions v as functions of the finite element space
    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)

    x = ufl.SpatialCoordinate(V.mesh)
    kappa = kappa_ufl(x)
    f = f_ufl(x)

    a = dfx.fem.form(kappa * ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx) #␣
↪bilinear
    rhs_form = dfx.fem.form(f * v * ufl.dx) # linear

    # assemble PETSc matrix
    A = dfx.fem.petsc.assemble_matrix(a, bcs=[bc])
    A.assemble()

    # create rhs vector, localForm is portion of processor
    b = dfx.fem.petsc.create_vector(rhs_form)
    # set values to zero
    with b.localForm() as loc_b:
        loc_b.set(0)
    # assemble vector
    dfx.fem.petsc.assemble_vector(b, rhs_form)
```

```
    # Apply Dirichlet boundary condition to the vector
    dfx.fem.petsc.apply_lifting(b, [a], [[bc]])
    b.ghostUpdate(
        addv=PETSc.InsertMode.ADD_VALUES, mode=PETSc.ScatterMode.REVERSE
    )
    dfx.fem.petsc.set_bc(b, [bc])

    # get a LinearSolver
    solver = PETSc.KSP().create(V.mesh.comm)
    solver.setOperators(A)
    solver.setType(PETSc.KSP.Type.PREONLY)
    solver.getPC().setType(PETSc.PC.Type.CHOLESKY)
    uh = dfx.fem.Function(V) # approximation
    solver.solve(b, uh.x.petsc_vec)
    uh.x.scatter_forward()
    return uh
```

Mit der folgenden Funktion `get_hmax` können Sie sich den größten Umkreisdurchmesser aller Gitter-Zellen zurückgeben lassen.

[7]:
```
def get_hmax(mesh: dfx.mesh.Mesh) -> float:
    tdim = mesh.topology.dim
    c_map = mesh.topology.index_map(tdim)
    num_cells_local = c_map.size_local + c_map.num_ghosts
    cells = np.arange(num_cells_local, dtype=np.int32)
    hs = mesh.h(tdim,cells)
    return np.max(hs)
```

Erstellen Sie einen Konvergenzplot, der den Fehler der numerischen Lösung in Abhängigkeit von verschiedenen Werten für `h` sowie für die FEM-Polynomgrade 1, 2 und 3 zeigt. Verwenden Sie dabei die L^2-Norm als Fehlermaß und berücksichtigen Sie die unten angegebenen Parameter. Welche Konvergenzraten beobachten Sie?

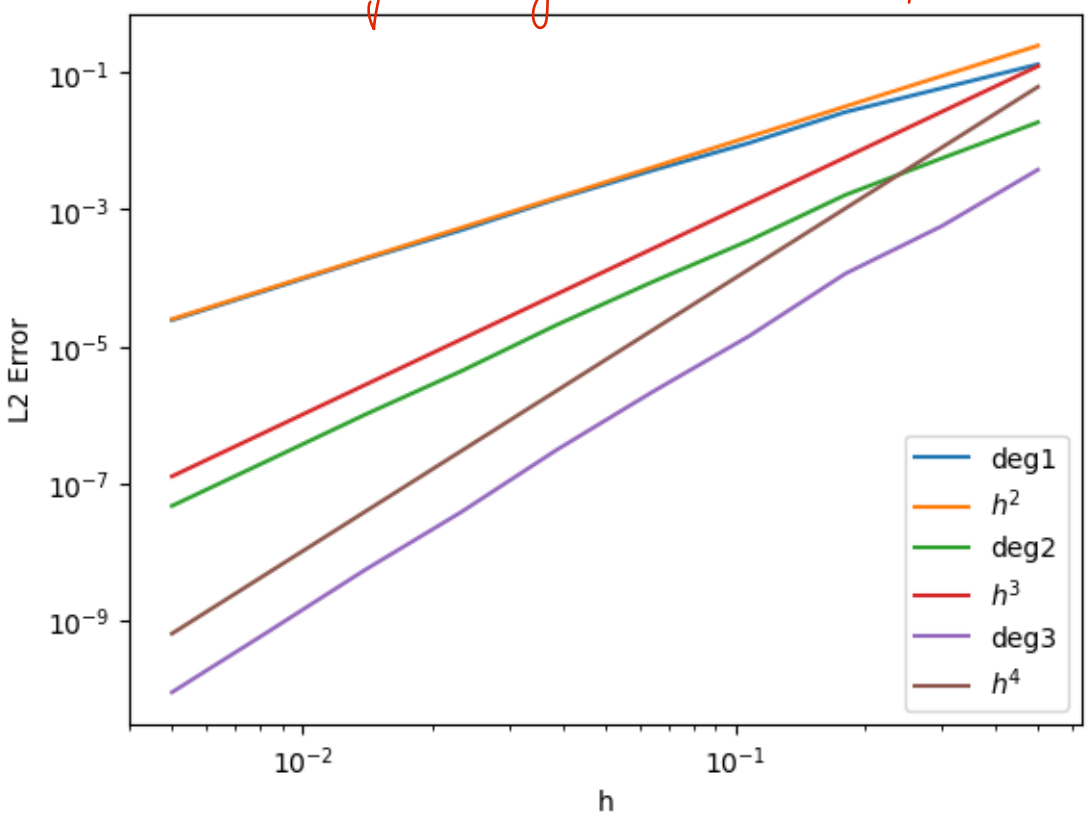
[8]:
```
def l2_error(u_np, uh, fem_type, fem_degree, mesh):
    Vhigherorder = dfx.fem.functionspace(mesh, (fem_type, fem_degree + 1))
    uex = dfx.fem.Function(Vhigherorder)
    uex.interpolate(u_np)
    L2_error = dfx.fem.form(ufl.inner(uh - uex, uh - uex) * ufl.dx)
    error_local = dfx.fem.assemble_scalar(L2_error)
    error_L2 = np.sqrt(mesh.comm.allreduce(error_local, op=MPI.SUM))

    if mesh.comm.rank == 0:
        return error_L2
    else:
        return
```

[9]:
```
fem_type = "Lagrange"
fem_degree = 1
degree_raise = 1
h_range = np.logspace(np.log(0.5), np.log(0.005), 10, base=np.e)
hmax_range = []
errors_l2 = {'deg1':[],'deg2':[],'deg3':[]}
for i, degree in enumerate(range(fem_degree, fem_degree + (len(errors_l2) - 1)␣
 ↪* degree_raise + 1, degree_raise)):
    for h in h_range:
        mesh, _, _ = build_unit_square_mesh(COMM_WORLD, h)
        uh = solve_Poisson(mesh, kappa_ufl, f_ufl, fem_type, degree)
        errors_l2[f"deg{degree}"].append(l2_error(u_np, uh, fem_type, degree,␣
 ↪mesh))

plt.loglog(h_range, errors_l2['deg1'], label="deg1")
plt.loglog(h_range, h_range**2, label="$h^2$")
plt.loglog(h_range, errors_l2['deg2'], label="deg2")
plt.loglog(h_range, h_range**3, label="$h^3$")
plt.loglog(h_range, errors_l2['deg3'], label="deg3")
plt.loglog(h_range, h_range**4, label="$h^4$")
plt.xlabel("h")
plt.ylabel("L2 Error")
plt.legend()
plt.show()
```

Ordnungslinien gestrichelt oder so, machts lesbarer



<– Platz für Ihre Antwort –>

Für p=1 der Fehler etwa h^2 beträgt, für p=2 etwa h^3 und für p=3 etwa h^4 ✓

2.2 b) Vergleich verschiedener Löser in 3D

Das Paket FEniCSx bietet große Flexibilität, so können z.B. auch 3D Probleme ohne größeren Mehraufwand gelöst werden. Hierfür können Thetraedergitter genutzt werden.

In dieser Teilaufgabe betrachten wir

$$\begin{aligned} (\alpha I - \beta\Delta)u &= f \quad \text{in} \quad \Omega = (0,1)^3, \\ u &= 0 \quad \text{auf} \quad \partial\Omega \end{aligned}$$

mit Parametern $\alpha, \beta \geq 0$. Wir nutzen dazu ein einfaches Gitter, welches wir von `dolfinx` direkt über `dolfinx.mesh.create_unit_cube` erzeugen können. In der nächsten Zelle müssen Sie lediglich die `system_form` korrekt anlegen.

[10]:
```
comm = MPI.COMM_WORLD
mesh = dfx.mesh.create_unit_cube(
    comm, 20, 20, 20, cell_type=dfx.mesh.CellType.tetrahedron
```

okay

```
)
V = dfx.fem.functionspace(mesh, ("Lagrange", 1))
uD = dfx.fem.Function(V)
# boundary condition
tdim = mesh.topology.dim
fdim = tdim - 1
mesh.topology.create_connectivity(fdim, tdim)
boundary_facets = dfx.mesh.exterior_facet_indices(mesh.topology)
boundary_dofs = dfx.fem.locate_dofs_topological(V, fdim, boundary_facets)
bc = dfx.fem.dirichletbc(uD, boundary_dofs)
alpha = dfx.fem.Constant(mesh, 1.0)
beta = dfx.fem.Constant(mesh, 1.0)
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)


system_form = dfx.fem.form(alpha * u * v * ufl.dx + beta * ufl.dot(ufl.grad(u),␣
 ↪ufl.grad(v)) * ufl.dx)



A = dfx.fem.petsc.assemble_matrix(system_form, bcs=[bc])
A.assemble()
x = ufl.SpatialCoordinate(V.mesh)
f = (
    -ufl.pi
    * ufl.pi
    * ufl.sin(ufl.pi * x[0])
    * ufl.sin(ufl.pi * x[0])
    * ufl.sin(ufl.pi * x[1])
    * ufl.sin(ufl.pi * x[1])
    * ufl.sin(ufl.pi * x[2])
    * ufl.sin(ufl.pi * x[2])
)
rhs_form = dfx.fem.form(f * v * ufl.dx)
uh = dfx.fem.Function(V)
b = dfx.fem.petsc.create_vector(rhs_form)
with b.localForm() as loc_b:
    loc_b.set(0)
dfx.fem.petsc.assemble_vector(b, rhs_form)
dfx.fem.petsc.apply_lifting(b, [system_form], bcs=[[bc]])
b.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.REVERSE)
dfx.fem.set_bc(b, [bc])
```

Legen Sie nun sechs verschiedene `PETSc.KSP` Objekte an um die folgenden Löser miteinander zu vergleichen. - Direktes Lösen über LU-Zerlegung - Direktes Lösen über Cholesky-Zerlegung - Iteratives Lösen mit dem CG-Verfahren ohne Vorkonditionierung - Iteratives Lösen mit dem CG-Verfahren und Jacobi Vorkonditionierung - Iteratives Lösen mit dem CG-Verfahren und SOR Vorkondition-

ierung - Iteratives Lösen mit dem CG-Verfahren und ICC (unvollständige Cholesky) Vorkonditionierung

Nutzen Sie das Paket `timeit` um die durchschnittliche Zeit zum Lösen über 5 Runden mit je 10 Durchläufen zu testen (`-r 5 -n 10`). Rufen Sie vor den timings auf dem KSP Objekt die Funktion `KSP_object.setUp()` auf, damit die Zerlegungen und Vorkonditionierer bereits davor angelegt werden. Lösen Sie außerdem je einmal vor den timings, um den Einfluss von caching-Effekten zu reduzieren. Setzen Sie bei den iterativen Verfahren `a_tol` und `r_tol` auf 10^{-5}.

Wenn Sie die Ergebnisse auch mit der Situation in 2D vergleichen wollen, können Sie dazu `dolfinx.mesh.create_unit_square` ganz analog benutzen.

[11]:
```
import timeit
from statistics import mean

repeat = 5
number = 10

solver = PETSc.KSP().create(V.mesh.comm)
solver.setOperators(A)

print("lu")
# lu
solver.setType(PETSc.KSP.Type.PREONLY)
solver.getPC().setType(PETSc.PC.Type.LU)
solver.setUp()
print("lol")
solver.solve(b, uh.x.petsc_vec)
print("LU: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.petsc_vec),␣
 ↪repeat=repeat, number=number)))

# cholesky
solver.getPC().setType(PETSc.PC.Type.CHOLESKY)
solver.setUp()
solver.solve(b, uh.x.petsc_vec)
print("Cholesky: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.petsc_vec),␣
 ↪repeat=repeat, number=number)))

# cg
solver.setType(PETSc.KSP.Type.CG)
solver.getPC().setType(PETSc.PC.Type.NONE)
solver.setTolerances(atol=1e-5, rtol=1e-5)
solver.solve(b, uh.x.petsc_vec)
print("CG: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.petsc_vec),␣
 ↪repeat=repeat, number=number)))

# cg jacobi
solver.getPC().setType(PETSc.PC.Type.JACOBI)
```

```
solver.solve(b, uh.x.petsc_vec)
print("CG Jacobi: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.
 ↪petsc_vec), repeat=repeat, number=number)))

# cg sor
solver.getPC().setType(PETSc.PC.Type.SOR)
solver.solve(b, uh.x.petsc_vec)
print("CG SOR: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.petsc_vec),␣
 ↪repeat=repeat, number=number)))

# cg icc
solver.getPC().setType(PETSc.PC.Type.ICC)
solver.solve(b, uh.x.petsc_vec)
print("CG ICC: ", mean(timeit.repeat(lambda: solver.solve(b, uh.x.petsc_vec),␣
 ↪repeat=repeat, number=number)))
```

```
lu
lol
LU:  0.01863451742101461
Cholesky:  0.02001681518740952
CG:  0.021716082631610335
CG Jacobi:  0.028570071840658785
CG SOR:  0.02896404783241451
CG ICC:  0.0242639543954283
```

#Iterationen wäre hilfreich

1,5 | 5

3 Aufgabe 2: Wellengleichung

Im Folgenden lösen wir die Wellengleichung

$$\begin{cases} \partial_{tt} u &= \nabla \cdot (\kappa \nabla u) + f, & \text{in } \Omega \times (0,T], \\ u(x,y,t) &= 0, & \text{auf } \Gamma \times (0,T], \\ u(x,y,0) &= u^0(x,y), & \text{in } \Omega, \\ \partial_t u(x,y,0) &= v^0(x,y), & \text{in } \Omega. \end{cases}$$

auf dem Einheitsquadrat $\Omega = (0,1)^2$ mit $\Gamma = \partial\Omega$ und Endzeit $T = 1$ mit dem θ-Verfahren zur Zeitintegration, welches folgendermaßen definiert ist:

$$M\partial_\tau^2 u_h^n = -Au_h^{n,\theta} + b^{n,\theta}, \qquad n = 1, \dots, N-1$$

mit

$$\partial_\tau^2 u_h^n = \frac{1}{\tau^2}(u_h^{n+1} - 2u_h^n + u_h^{n-1})$$

und

$$u_h^{n,\theta} = \theta u_h^{n+1} + (1-2\theta)u_h^n + \theta u_h^{n-1}$$

und $b^{n,\theta}$ analog. Welche Verfahren erhalten Sie in den Fällen $\theta = 0$ bzw. $\theta = \frac{1}{4}$?

<– Platz für Ihre Antwort –>

$$\theta = 0$$

: Leapfrog Verfahren

$$\theta = 0.25$$

: Crank Nicolson Verfahren

Schreiben Sie sich zuerst eine Funktion `wave_startup` analog zu Abschnitt 15.6.2 (unter (15.41)) im Skript.

[13]:
```
from typing import Union

def wave_startup(
    V: dfx.fem.FunctionSpace,
    u0: dfx.fem.Function,
    v0: dfx.fem.Function,
    bc: dfx.fem.bcs.DirichletBC,
    f0: Union[dfx.fem.Function, dfx.fem.Constant],
    f1: Union[dfx.fem.Function, dfx.fem.Constant],
    kappa: Union[dfx.fem.Function, dfx.fem.Constant],
    tau: float
) -> dfx.fem.Function:
    # uh^1
    # create mass matrix
    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)
    m = dfx.fem.form(u * v * ufl.dx)
    M = dfx.fem.petsc.assemble_matrix(m, bcs=[bc])
    M.assemble()
    # assemble A
    a = dfx.fem.form(kappa * ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx) #␣
 ↪bilinear
    A = dfx.fem.petsc.assemble_matrix(a, bcs=[bc])
    A.assemble()
    # create bh^0
    # b_form = dfx.fem.form(f0 * v * ufl.dx)
    # b0 = dfx.fem.petsc.create_vector(b_form)

    RHS = (u0 + tau * v0) * v * ufl.dx + tau**2 / 2 * (- kappa * ufl.dot(ufl.
 ↪grad(u0), ufl.grad(v)) * ufl.dx + f0 * v * ufl.dx)
    rhs_form = dfx.fem.form(RHS)

    # create rhs vector, localForm is portion of processor
    b = dfx.fem.petsc.create_vector(rhs_form)
    # set values to zero
    with b.localForm() as loc_b:
        loc_b.set(0)
    # assemble vector
    dfx.fem.petsc.assemble_vector(b, rhs_form)
```

```
    # Apply Dirichlet boundary condition to the vector
    dfx.fem.petsc.apply_lifting(b, [a], [[bc]])
    b.ghostUpdate(
        addv=PETSc.InsertMode.ADD_VALUES, mode=PETSc.ScatterMode.REVERSE
    )
    dfx.fem.petsc.set_bc(b, [bc])

    # get a LinearSolver
    solver = PETSc.KSP().create(V.mesh.comm)
    solver.setOperators(M)
    solver.setType(PETSc.KSP.Type.PREONLY)
    solver.getPC().setType(PETSc.PC.Type.CHOLESKY)
    u1 = dfx.fem.Function(V) # approximation
    # solver.solve(M @ u0 + tau * M @ v0 + (tau**2)/2 * (-A @ u0 + b), u1.x.
↪petsc_vec)
    solver.solve(b, u1.x.petsc_vec)
    u1.x.scatter_forward()
    return u1
```

Wir legen uns wieder ein kleines Testbeispiel an. Wir betrachten als Lösung den Ausdruck

$$u(x, y, t) = \sin^2(\pi x) \sin^2(\pi y) \exp(t)$$

und als Materialparameter wählen wir

$$\kappa(x, y) = \exp(x + y)\,.$$

Bestimmen Sie die rechte Seite so f so, dass u eine Lösung der Wellengleichung ist und legen Sie die numpy und ufl Ausdrücke `u_np`,`u_ufl`,`kappa_ufl` und `f_ufl` entsprechend an.

[14]:
```
x_sym, y_sym, t_sym = sp.symbols('x y t')
u_sym = sp.sin(sp.pi * x_sym)**2 * sp.sin(sp.pi * y_sym)**2 * sp.exp(t_sym)
du_dt_sym = sp.diff(u_sym, t_sym, 1)
print(du_dt_sym)
kappa_sym = sp.exp(x_sym + y_sym)
du_dtt_sym = sp.diff(u_sym, t_sym, 2)
grad_u_sym = derive_by_array(u_sym, (x_sym, y_sym))
kappa_grad_u_sym = kappa_sym * grad_u_sym
f_sym = du_dtt_sym - (sp.diff(kappa_grad_u_sym[0], x_sym) + sp.
↪diff(kappa_grad_u_sym[1], y_sym))
print(f_sym)
print(sp.simplify(sp.simplify(f_sym)))
print(sp.simplify(sp.simplify(sp.simplify(f_sym))))
```

```
exp(t)*sin(pi*x)**2*sin(pi*y)**2
4*pi**2*exp(t)*exp(x + y)*sin(pi*x)**2*sin(pi*y)**2 - 2*pi*exp(t)*exp(x +
y)*sin(pi*x)**2*sin(pi*y)*cos(pi*y) - 2*pi**2*exp(t)*exp(x +
y)*sin(pi*x)**2*cos(pi*y)**2 - 2*pi*exp(t)*exp(x +
y)*sin(pi*x)*sin(pi*y)**2*cos(pi*x) - 2*pi**2*exp(t)*exp(x +
```

```
y)*sin(pi*y)**2*cos(pi*x)**2 + exp(t)*sin(pi*x)**2*sin(pi*y)**2
(8*pi**2*exp(x + y)*sin(pi*x)**2*sin(pi*y)**2 - 2*pi**2*exp(x + y)*sin(pi*x)**2
- 2*pi*exp(x + y)*sin(pi*x)*sin(pi*y)*sin(pi*(x + y)) - 2*pi**2*exp(x +
y)*sin(pi*y)**2 + sin(pi*x)**2*sin(pi*y)**2)*exp(t)
(8*pi**2*exp(x + y)*sin(pi*x)**2*sin(pi*y)**2 - 2*pi**2*exp(x + y)*sin(pi*x)**2
- 2*pi*exp(x + y)*sin(pi*x)*sin(pi*y)*sin(pi*(x + y)) - 2*pi**2*exp(x +
y)*sin(pi*y)**2 + sin(pi*x)**2*sin(pi*y)**2)*exp(t)
```

[15]:
```python
def u_np(t_eval):
    expr = lambda x: np.sin(np.pi * x[0])**2 * np.sin(np.pi * x[1])**2 * np.
 ↪exp(t_eval)
    return expr
initial_value = lambda x: u_np(0)(x)
initial_velocity = lambda x: u_np(0)(x)
def bnd_func(t_eval):
    expr = lambda x: 0.0 * x[0]
    return expr

u_ufl = lambda x: ufl.sin(ufl.pi * x[0])**2 * ufl.sin(ufl.pi * x[1])**2 * ufl.
 ↪exp(x[2])
kappa_ufl = lambda x: ufl.exp(x[0] + x[1])
def f_ufl(t_eval, x):
    exp_ufl = ufl.exp(x[0] + x[1])
    sin_x_ufl = ufl.sin(ufl.pi * x[0])
    sin_y_ufl = ufl.sin(ufl.pi * x[1])
    return ufl.exp(t_eval) * (
        8 * ufl.pi**2 * exp_ufl * sin_x_ufl**2 * sin_y_ufl**2
        - 2 * ufl.pi**2 * exp_ufl * sin_x_ufl**2
        - 2 * ufl.pi * exp_ufl * sin_x_ufl * sin_y_ufl * ufl.sin(ufl.pi * (x[0]␣
 ↪+ x[1]))
        - 2 * ufl.pi**2 * exp_ufl * sin_y_ufl**2
        + sin_x_ufl**2 * sin_y_ufl**2)
```

Vervollständigen Sie die Funktion `plot_sol_pyvista`. Diese können Sie verwenden, um Snapshots Ihrer berechneten Lösung zu generieren. Dies wird insbesondere in Aufgabe 3 relevant werden.

[16]:
```python
def plot_sol_pyvista(V: dfx.fem.FunctionSpace, sol: dfx.fem.Function, filename:␣
 ↪str) -> None:
    # grid for pyvista
    tdim = V.mesh.topology.dim
    fdim = tdim - 1
    V.mesh.topology.create_connectivity(fdim, tdim)
    topology, cell_types, geometry = dfx.plot.vtk_mesh(V.mesh, tdim)
    grid = pyvista.UnstructuredGrid(topology, cell_types, geometry)
    grid.point_data["u"] = sol.x.array.real
    grid.set_active_scalars("u")
    # plot grid
    u_plotter = pyvista.Plotter()
```

```
    u_plotter.add_mesh(grid, show_edges=True)
    u_plotter.view_xy()
    u_plotter.show(screenshot=f"{filename}.png")
```

Vervollständigen Sie die Funktion `solve_wave`, welche die obige Wellengleichung numerisch löst und als Zeitintegrationsverfahren das θ-Verfahren nutzt.

[17]:
```
from pathlib import Path
from dolfinx.nls.petsc import NewtonSolver
from ufl import Measure


def solve_wave(
    V: dfx.fem.FunctionSpace,
    boundary_dofs: np.ndarray,
    t0: float,
    T: float,
    num_steps: int,
    initial_value: Callable,
    initial_velocity: Callable,
    kappa_ufl: Callable,
    f: Callable,
    bnd_func: Callable,
    theta: float = 0.25,
    snapshots: List[int] = [],
) -> dfx.fem.Function:
    tau = (T - t0) / num_steps
    cool_mesh = V.mesh
    dx = Measure("dx", domain=V.mesh.ufl_domain())
    print(mesh.ufl_domain())
    x = ufl.SpatialCoordinate(cool_mesh)
    kappa = kappa_ufl(x)
    tn_minus = dolfinx.fem.Constant(cool_mesh, t0)
    tn = dolfinx.fem.Constant(cool_mesh, tau)
    tn_plus = dolfinx.fem.Constant(cool_mesh, 2*tau)
    expr_f_tn_minus = f(tn_minus,x)
    expr_f_tn = f(tn, x)
    expr_f_tn_plus = f(tn_plus, x)
    # homogenous boundary condition
    # boundary_dofs = get_all_boundary_dofs(V)
    uB = dfx.fem.Function(V)
    uB.interpolate(bnd_func(t0))
    bc = dfx.fem.dirichletbc(uB, boundary_dofs)
    # create bs
    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)
    # create u0, u1, u2
```

```
   u_n_minius = dfx.fem.Function(V)
   u0 = dfx.fem.Function(V)
   v0 = dfx.fem.Function(V)
   u0.interpolate(initial_value)
   v0.interpolate(initial_velocity)
   f0_expr = dfx.fem.Expression(expr_f_tn_minus, V.element.
↪interpolation_points(), comm=COMM_WORLD)
   f0 = dfx.fem.Function(V)
   f0.interpolate(f0_expr)
   f1_expr = dfx.fem.Expression(f_ufl(tau, x), V.element.
↪interpolation_points(), comm=COMM_WORLD)
   f1 = dfx.fem.Function(V)
   f1.interpolate(f1_expr) # why need f1
   un_minus = u0
   un = dfx.fem.Function(V)
   un.interpolate(wave_startup(V, u0, v0, bc, f0, f1, kappa, tau)) # uh^1
   un_plus = dfx.fem.Function(V)

   RHS = (2 * un - un_minus) * v * dx - tau**2 * kappa * ufl.dot(ufl.grad((1 -␣
↪2 * theta) * un + theta * un_minus), ufl.grad(v)) * dx + tau**2 * ((1 - 2 *␣
↪theta) * expr_f_tn + theta * expr_f_tn_minus) * v * dx # tau^2 ((1 - 2 *␣
↪theta)b_h^n + theta * b_h^{n-1})
   LHS = un_plus * v * dx + tau**2 * kappa * ufl.dot(ufl.grad(theta *␣
↪un_plus), ufl.grad(v)) * dx - tau**2 * theta * expr_f_tn_plus * v * dx #␣
↪-tau^2 * theta * b_h^{n+1}

   t = t0 + tau
   # solve nonlinear https://jsdokken.com/FEniCS-workshop/src/deep_dive/
↪lifting.html
   problem = dfx.fem.petsc.NonlinearProblem(RHS - LHS, un_plus, bcs=[bc])
   solver = NewtonSolver(V.mesh.comm, problem)
   # solver.convergence_criterion = "residual"
   for n in range(1, num_steps):
       # t = t0 + (n+1)*tau
       t = t0 + n*tau
       # update tn
       tn_minus.value = t - tau
       tn.value = t
       tn_plus.value = t + tau
       # boundary conditions always 0

       solver.solve(un_plus)

       un_minus.x.array[:] = un.x.array
       un.x.array[:] = un_plus.x.array

       if n in snapshots:
```

```
            results_folder = Path("results")
            results_folder.mkdir(exist_ok=True, parents=True)
            filename = results_folder / f"wave_{n}"
            plot_sol_pyvista(V, un, filename)


    return un


h = 0.05
mesh, _, _ = build_unit_square_mesh(COMM_WORLD, h)
V = dfx.fem.functionspace(mesh, (fem_type, fem_degree))
boundary_dofs = get_all_boundary_dofs(V)
uh = solve_wave(
    V, boundary_dofs, 0.0, 1.0, 20, initial_value, initial_velocity,␣
 ↪kappa_ufl,f_ufl, bnd_func
)
uh
```

```
<Mesh #33>
```

[17]:
```
Coefficient(FunctionSpace(Mesh(blocked element (Basix element (P, triangle, 1,
equispaced, unset, False, float64, []), (2,)), 33), Basix element (P, triangle,
5, gll_warped, unset, False, float64, [])), 98)
```

Erstellen Sie einen Konvergenzplot, der den Fehler der numerischen Lösung in Abhängigkeit von verschiedenen Werten für h sowie für die FEM-Polynomgrade 1, 2 und 3 zeigt. Verwenden Sie dabei die L^2-Norm als Fehlermaß. Wählen Sie $\theta = \frac{1}{4}$ und die unten aufgeführten Parameter. Erklären Sie was Sie sehen.

fehlt – 7P

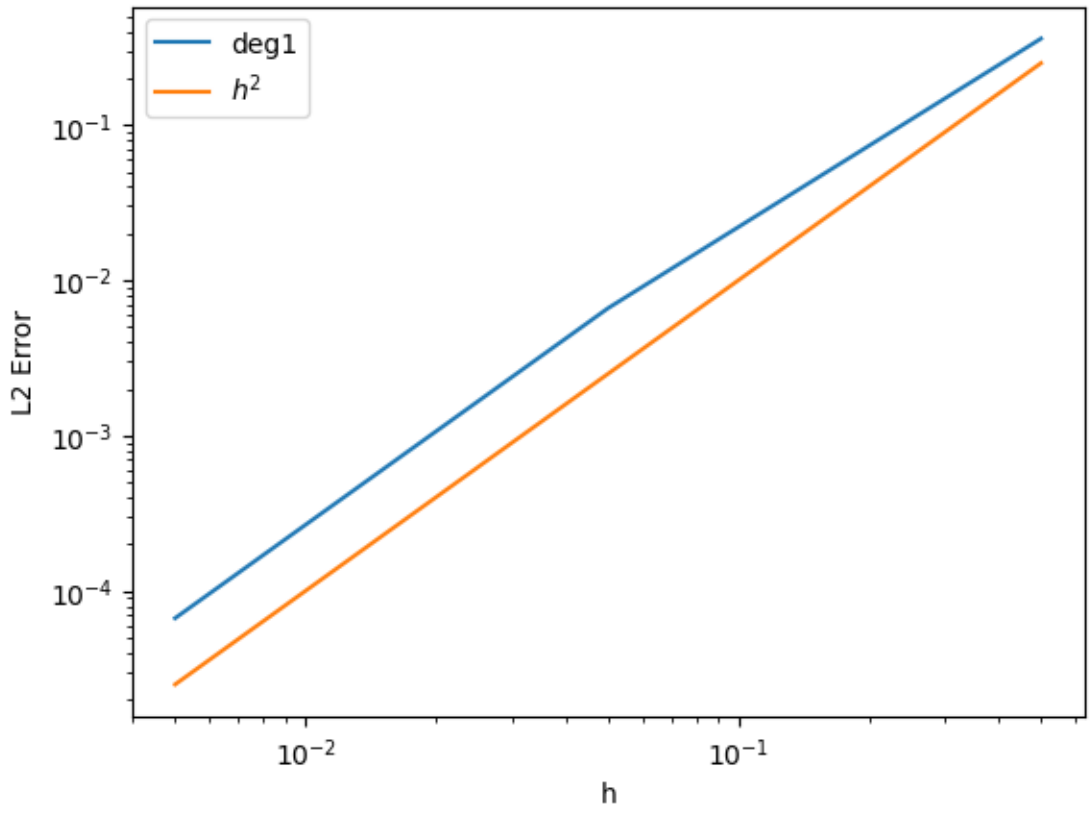
[18]:
```
fem_type = "Lagrange"
fem_degree = 1
degree_raise = 1
h_range = np.logspace(np.log(0.5), np.log(0.005), 3, base=np.e)
t0 = 0.0
T = 1.0
num_timesteps = 250
errors_l2 = {'deg1':[],'deg2':[],'deg3':[]}
errors_l2 = {'deg1':[]}
hmax_range = []
for i, degree in enumerate(range(fem_degree, fem_degree + (len(errors_l2) - 1)␣
 ↪* degree_raise + 1, degree_raise)):
    print(degree)
    for h in h_range:
        mesh, _, _ = build_unit_square_mesh(COMM_WORLD, h)
        V = dfx.fem.functionspace(mesh, (fem_type, degree))
        boundary_dofs = get_all_boundary_dofs(V)
```

etwas wenig

```
        uh = solve_wave(V, boundary_dofs, t0, T, num_timesteps, initial_value,␣
 ↪initial_velocity, kappa_ufl,f_ufl, bnd_func)
        errors_l2[f"deg{degree}"].append(l2_error(u_np(T), uh, fem_type,␣
 ↪degree, mesh))

plt.loglog(h_range, errors_l2['deg1'], label="deg1")
plt.loglog(h_range, h_range**2, label="$h^2$")
# plt.loglog(h_range, errors_l2['deg2'], label="deg2")
# plt.loglog(h_range, h_range**3, label="$h^3$")
# plt.loglog(h_range, errors_l2['deg3'], label="deg3")
# plt.loglog(h_range, h_range**4, label="$h^4$")
plt.xlabel("h")
plt.ylabel("L2 Error")
plt.legend()
plt.show()
```

```
1
<Mesh #34>
<Mesh #35>
<Mesh #36>
```



<– Platz für Ihre Antwort –>

Für theta=1/4 ist die Ordung 2

Erstellen Sie außerdem ein Konvergenzplot, der den Fehler der numerischen Lösung in Abhängigkeit von verschiedenen Werten für τ und die Werte $\theta \in \{0, \frac{1}{4}\}$ zeigt. Nutzen Sie die vorgegebenen Parameter und erklären Sie anschließend, was Sie beobachten.

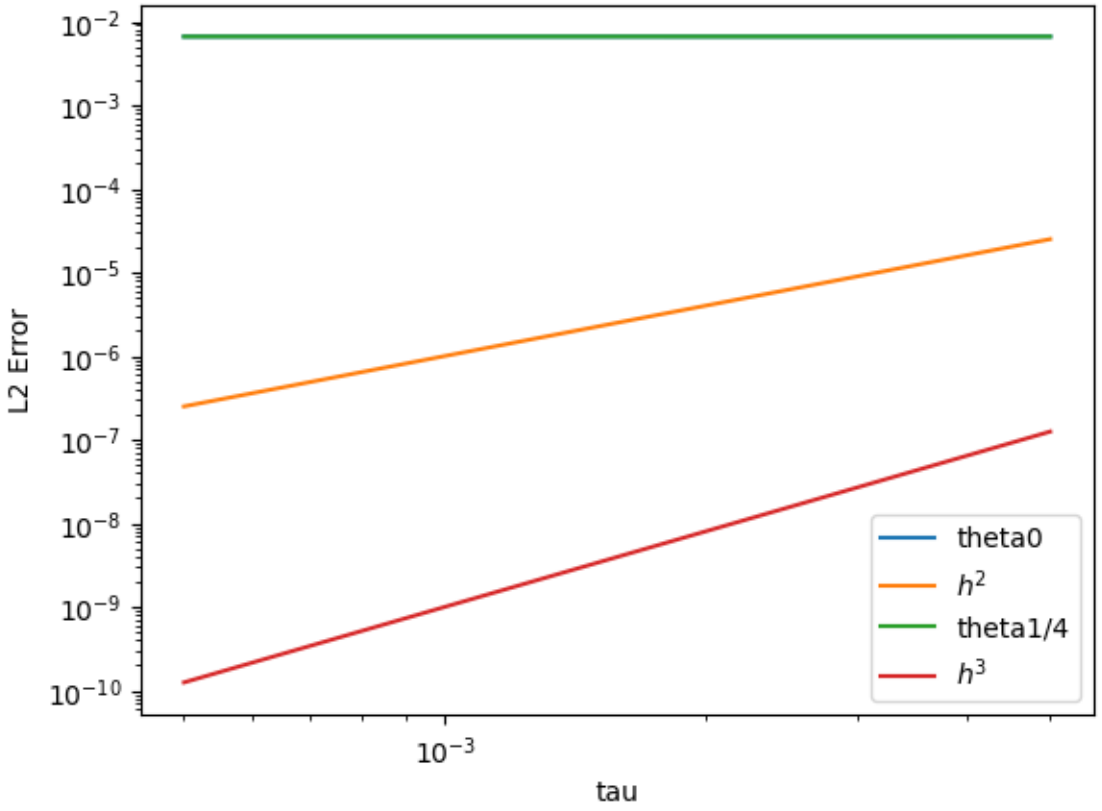
[19]:
```
fem_type = "Lagrange"
fem_degree = 5
degree_raise = 1
h = 0.05
t0 = 0.0
T = 1.0
num_timesteps = np.geomspace(200,2000,10,dtype=int)
errors_l2 = {"theta0": [], "theta1/4": []}
mesh, _, _ = build_unit_square_mesh(COMM_WORLD, h)
V = dfx.fem.functionspace(mesh, (fem_type, degree)) falscher degree -0,5P
boundary_dofs = get_all_boundary_dofs(V)
for i, theta in enumerate([0, 0.25]):
    for num_timestep in num_timesteps:
        print(num_timestep)
        uh = solve_wave(V, boundary_dofs, t0, T, num_timestep, initial_value,␣
 ↪initial_velocity, kappa_ufl,f_ufl, bnd_func) theta nicht verwendet -0,5P
        errors_l2[list(errors_l2.keys())[i]].append(l2_error(u_np(T), uh,␣
 ↪fem_type, degree, mesh))
        print(l2_error(u_np(T), uh, fem_type, degree, mesh))

taus = (T - t0) / num_timesteps
plt.loglog(taus, errors_l2['theta0'], label="theta0")
plt.loglog(taus, taus**2, label="$h^2$")
plt.loglog(taus, errors_l2['theta1/4'], label="theta1/4")
plt.loglog(taus, taus**3, label="$h^3$")
plt.xlabel("tau")
plt.ylabel("L2 Error")
plt.legend()
plt.show()
```

```
200
<Mesh #37>
0.006610848835533364
258
<Mesh #37>
0.0066104764921560664
333
<Mesh #37>
0.0066102245654244465
430
<Mesh #37>
```

```
0.006610417822528752
556
<Mesh #37>
0.006610692750107583
718
<Mesh #37>
0.006610907316965735
928
<Mesh #37>
0.006611056783978278
1198
<Mesh #37>
0.00661116029070755
1548
<Mesh #37>
0.006611236198372343
2000
<Mesh #37>
0.006611293641166928 4
200
<Mesh #37>
0.006610848835533364
258
<Mesh #37>
0.006610476492156066 4
333
<Mesh #37>
0.006610224565424446 5
430
<Mesh #37>
0.006610417822528752
556
<Mesh #37>
0.006610692750107583
718
<Mesh #37>
0.006610907316965735
928
<Mesh #37>
0.006611056783978278
1198
<Mesh #37>
0.00661116029070755
1548
<Mesh #37>
0.006611236198372343
2000
<Mesh #37>
```

```
0.0066112936411669284
```



<– Platz für Ihre Antwort –>

theta=0, Ord=2 theta=1/4, Ord=3

4 Aufgabe 3: Doppelspalt

Im Folgenden wollen wir uns noch ein Beispiel der Wellengleichung mit gemischten Randbedingungen anschauen. Dafür betrachten wir die Wellengleichung aus Aufgabe 2 auf einem etwas komplizierteren Gebiet, dessen Triangulierung in `double_slit.msh` gegeben ist. Des Weiteren betrachten wir die Daten $\kappa = 1$, $u_0 = v_0 = f = 0$ und die Randbedingungen

$$\begin{cases} u(x,y,t) & = \frac{1}{10\pi}\cos(10\pi t), & (x,y) \in \Gamma_D = \{0\} \times [0,1], \\ \partial_n u(x,y,t) & = 0, & (x,y) \in \Gamma \setminus \Gamma_D. \end{cases}$$

Lesen Sie das Gitter mithilfe der Funktion `dolfinx.io.gmshio.read_from_msh` ein und plotten Sie es mit `pyvista`. Sie können sich dafür am Tutorial in `intro_dolfinx.ipynb` orientieren.

[146]:
```
mesh,_,_ = dfx.io.gmshio.read_from_msh("double_slit.msh", COMM_WORLD)
# grid for pyvista
tdim = mesh.topology.dim
```

```
fdim = tdim - 1
mesh.topology.create_connectivity(tdim, tdim)
topology, cell_types, geometry = dfx.plot.vtk_mesh(mesh, tdim)
grid = pyvista.UnstructuredGrid(topology, cell_types, geometry)
# plot grid
u_plotter = pyvista.Plotter()
u_plotter.add_mesh(grid, show_edges=True)
u_plotter.view_xy()
u_plotter.show()
```

```
Info    : Reading 'double_slit.msh'…
Info    : 33 entities
Info    : 19102 nodes
Info    : 38204 elements
Info    : Done reading 'double_slit.msh'
```

Definieren Sie sich die benötigten Daten und rufen Sie `solve_wave` mit den gegebenen Parametern auf.

[147]:
```
print("")
```

[148]:
```
def inhom_bnd(x):
    return np.isclose(x[0], 0.0)
V = dfx.fem.functionspace(mesh, (fem_type, fem_degree))
inhom_bc_facets = dfx.mesh.locate_entities_boundary(mesh, fdim, inhom_bnd)
inhom_boundary_dofs = dfx.fem.locate_dofs_topological(V, fdim, inhom_bc_facets)
uB = dfx.fem.Function(V)
dirichlet_bc = dfx.fem.dirichletbc(uB, inhom_boundary_dofs)
initial_value = lambda x: 0.0 * x[0] + 0.0 * x[1]
initial_velocity = lambda x: 0.0 * x[0] + 0.0 * x[1]
kappa = lambda x: 1.0 + ufl.as_ufl(0.0 * x[0] + 0.0 * x[1])
def f(t_eval, x):
    return ufl.as_ufl(0.0 * x[0] + 0.0 * x[1])
def bnd_func(t_eval):
    return lambda x: (1 / (10 * np.pi)) * np.cos(10 * np.pi * t_eval) + 0.0 *␣
 ↪x[0] + 0.0 * x[1]
```

[ ]:
```
t0 = 0.0
T = 10.0
num_timesteps = 10000
theta = 0.25
snapshots = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

print(mesh)
print(mesh.ufl_domain())
uh = solve_wave(V, inhom_boundary_dofs, t0, T, num_timesteps, initial_value,␣
 ↪initial_velocity, kappa, f, bnd_func, theta, snapshots)
```

```
<dolfinx.mesh.Mesh object at 0x7f0ca0162d50>
<Mesh #124>
<Mesh #124>
```

falsche Aufgabe und f.f.

[ ]: