Lecture 03 - FEM II

Nonlinear problems, mixed problems, eigenvalue problems

Lambert Theisen

1 Objectives

In this lecture, we will continue to use the `dolfinx` library from the FEniCS project to solve some more PDE boundary value problems:

1. A nonlinear Poisson equation
2. Stokes equations (a *mixed finite element* problem) with heat transport;
3. (Linear elasticity);
4. Schrödinger equation (eigenvalue problem).

1.1 Internals of the FEniCS project

More information can be found in their paper [1].

> **Note**
>
> Any questions, so far?

2 Nonlinear Poisson

In the previous lecture, we have solved the linear Poisson equation. We will now solve a nonlinear variant of the Poisson equation, which reads

$$-\nabla \cdot (q(u)\nabla u) = f$$

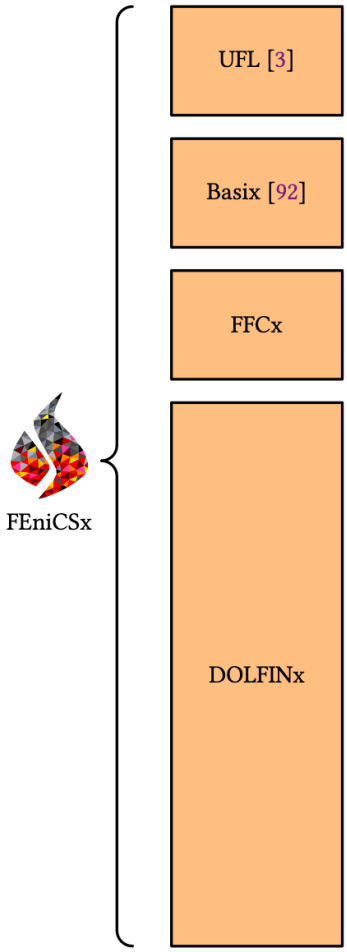
on the unit square $\Omega := (0,1)^2$ with $q(u) = 1 + u^2$. Here, q is a nonlinear diffusion coefficient, which depends on the solution u itself.

> **What changes compared to linear Poisson?**
>
> For linear Poisson, the diffusion coefficient is known before solving. Here, the coefficient depends on the unknown solution, so the discrete algebraic system is nonlinear and must be solved iteratively.

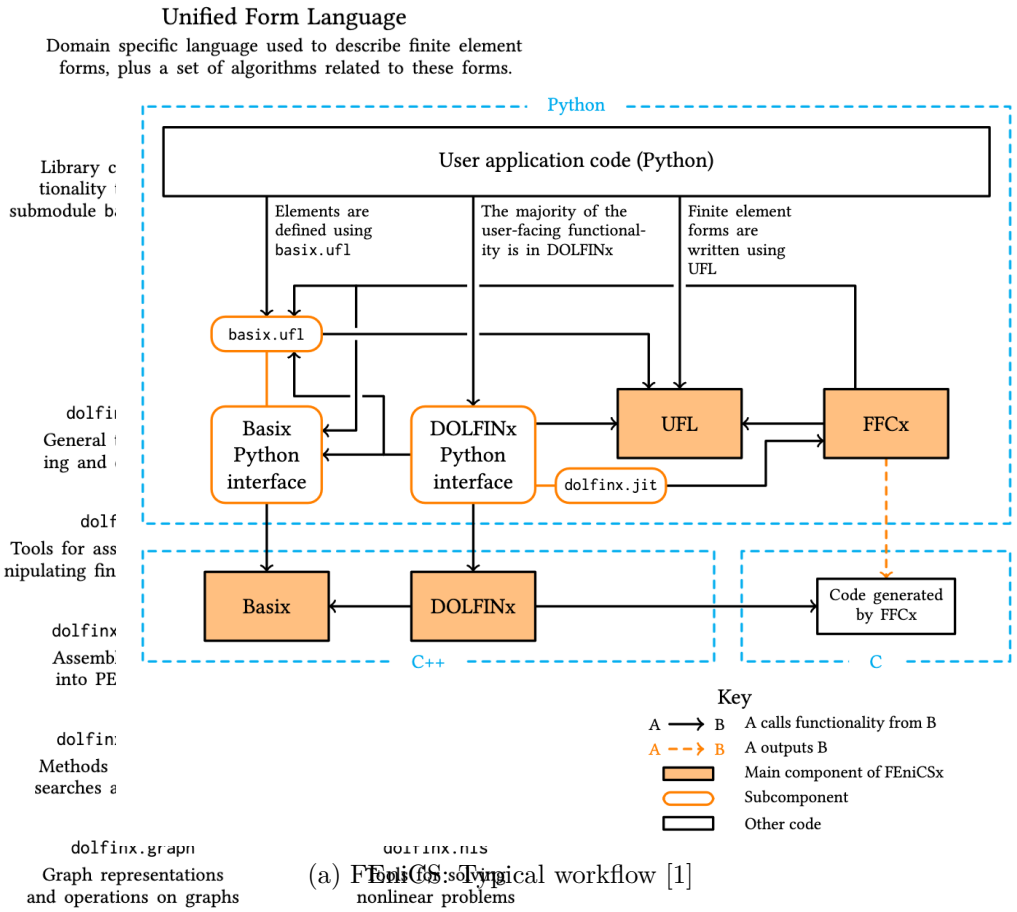
Linear Poisson

$$-\nabla \cdot (k\nabla u) = f$$

Known coefficient k.



(a) FEniCS: packages overview [1]



(a) FEniCS: Typical workflow [1]

Nonlinear Poisson

$$-\nabla \cdot (q(u)\nabla u) = f$$

Coefficient $q(u)$ depends on the solution.

2.1 Reminder: Newton's method

Newton's method solves a nonlinear scalar equation $g(x) = 0$ by repeatedly replacing g with its tangent line at the current iterate. If $g'(x_k) \neq 0$, the update is

$$x_{k+1} = x_k - \frac{g(x_k)}{g'(x_k)}.$$

For a nonlinear finite element problem, the same idea is applied to the residual $F(u_h; v) = 0$: assemble a Jacobian, solve a linear correction problem, update the current approximation, and repeat.

```
import numpy as np
import matplotlib.pyplot as plt

def g(x):
    return x**2 - 2

def dg(x):
    return 2 * x

xs = np.linspace(0.4, 2.1, 400)
root = np.sqrt(2)

xk = 0.7
iterates = [xk]
for _ in range(5):
    xk = xk - g(xk) / dg(xk)
    iterates.append(xk)

n_steps = len(iterates) - 1

for frame in range(n_steps):
    fig, ax = plt.subplots(figsize=(4.5, 3.2))
    ax.axhline(0.0, color="black", linewidth=0.8)
    ax.plot(xs, g(xs), color="tab:blue", linewidth=2, label=r"$g(x)=x^2-2$")
    ax.scatter([root], [0], color="black", zorder=4, label=r"$\sqrt{2}$")

    for k in range(frame + 1):
        xk_val = iterates[k]
        x_next = iterates[k + 1]
        tangent_xs = np.linspace(min(xk_val, x_next), max(xk_val, x_next), 50)
        tangent = g(xk_val) + dg(xk_val) * (tangent_xs - xk_val)
        ax.plot(tangent_xs, tangent, color="tab:orange", linewidth=1.8)
        ax.plot(
```

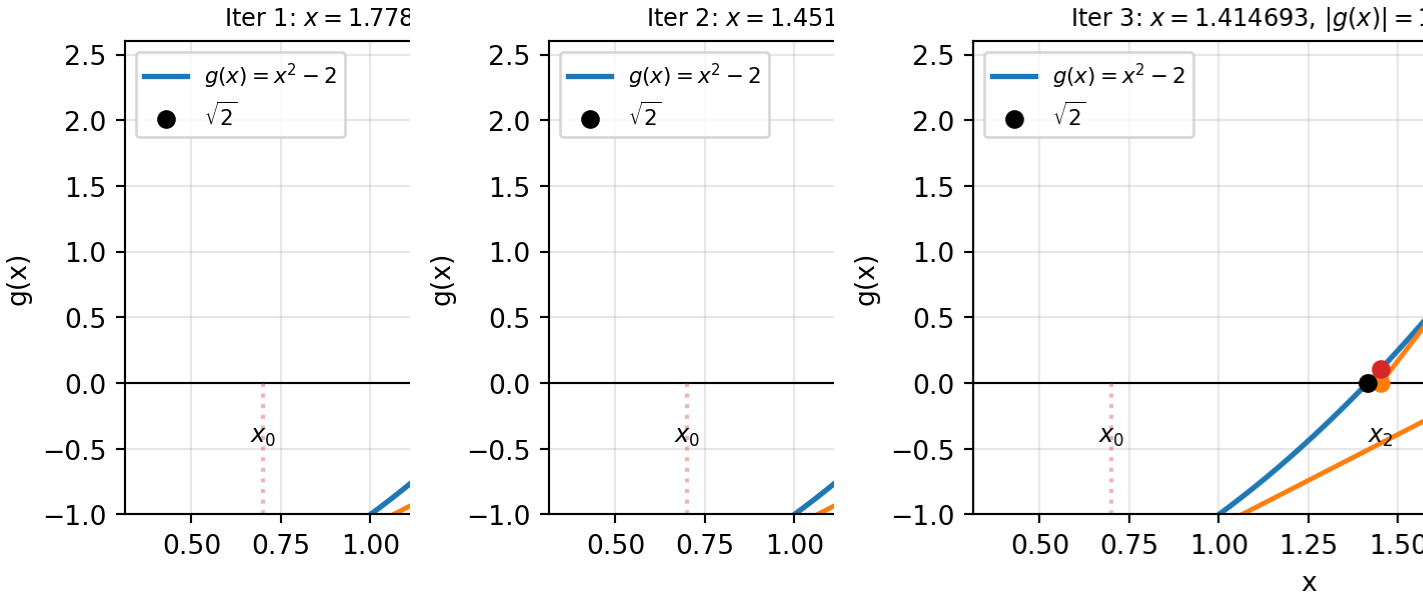
```
            [xk_val, xk_val], [0, g(xk_val)], color="tab:red", alpha=0.35,
            linestyle=":"
        )
        ax.scatter([xk_val], [g(xk_val)], color="tab:red", zorder=3)
        ax.scatter([x_next], [0], color="tab:orange", zorder=3)
        ax.text(xk_val, -0.45, rf"$x_{k}$", ha="center", fontsize=9)

    ax.set_title(rf"Iter {frame + 1}: $x={iterates[frame + 1]:.6f}$, "
                 rf"$|g(x)|={abs(g(iterates[frame + 1])):.2e}$",
                 fontsize=9)
    ax.set_xlabel("x")
    ax.set_ylabel("g(x)")
    ax.set_ylim(-1.0, 2.6)
    ax.grid(True, alpha=0.3)
    ax.legend(loc="upper left", fontsize=8)
    plt.show()
```

- Validating by checking the residual, in that case `g(x)` and the current change (`correction`) should be small.

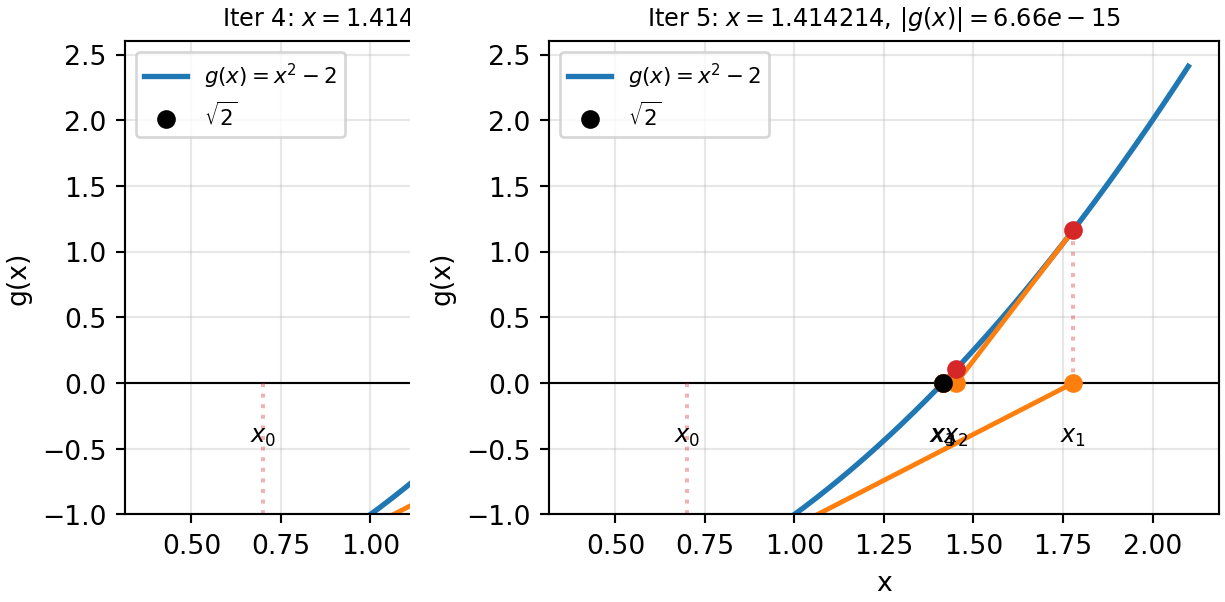
> **From scalar Newton to PDE Newton**
>
> The scalar derivative $g'(x_k)$ becomes the Jacobian form $J(u_h; \delta u, v)$. The scalar correction $x_{k+1} - x_k$ becomes a finite element function δu_h.

```
xk = 1.0
tolerance = 1.0e-12
max_iterations = 8
for k in range(max_iterations):
    residual = g(xk)
    jacobian = dg(xk)
    correction = -residual / jacobian
    print(
        f"k={k}: x={xk:.12f}, "
        f"g(x)={residual:.3e}, correction={correction:.3e}"
    )
    xk = xk + correction
    if abs(residual) < tolerance:
        break
```

```
k=0: x=1.000000000000, g(x)=-1.000e+00, correction=5.000e-01
k=1: x=1.500000000000, g(x)=2.500e-01, correction=-8.333e-02
k=2: x=1.416666666667, g(x)=6.944e-03, correction=-2.451e-03
k=3: x=1.414215686275, g(x)=6.007e-06, correction=-2.124e-06
k=4: x=1.414213562375, g(x)=4.511e-12, correction=-1.595e-12
k=5: x=1.414213562373, g(x)=4.441e-16, correction=-1.570e-16
```



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3



(d) Iteration 4 (e) Iteration 5

Figure 3: Newton's method for $g(x) = x^2 - 2$

2.2 Imports

We import the usual FEniCSx modules, as well as `numpy` for the exact solution and `matplotlib` for plotting.

> **Note**
>
> Note that in FEniCSx `0.10.0.post5`, the `NonlinearProblem` class has been deprecated in favor of the PETSc `SNES` solver, see this changelog for details.

```
from mpi4py import MPI
from petsc4py import PETSc

import numpy as np
import ufl

from dolfinx import fem, mesh
from dolfinx.fem.petsc import NonlinearProblem
```

2.3 Mesh and function space

We use a uniform mesh of the unit square and continuous, piecewise linear finite elements, i.e., the `P1` element obtained with `("Lagrange", 1)`.

> **Notation**
>
> The continuous problem is posed on $\Omega = (0,1)^2$. The finite element solution u_h lives in a finite-dimensional space $V_h \subset H^1(\Omega)$ built on the mesh.

```
domain = mesh.create_unit_square(MPI.COMM_WORLD, 32, 32)
V = fem.functionspace(domain, ("Lagrange", 1))

# domain = mesh.create_unit_square(
#     MPI.COMM_WORLD,
#     32, 32,
#     cell_type=mesh.CellType.quadrilateral,
# )

# V = fem.functionspace(domain, ("Q", 2))   # tensor-product Q2 element
```

```
import matplotlib.pyplot as plt


def mesh_triangulation(domain):
    import matplotlib.tri as mtri

    tdim = domain.topology.dim
    domain.topology.create_connectivity(tdim, 0)

    cells = domain.topology.connectivity(tdim, 0).array.reshape(-1, 3)
```

```
    nverts = (
        domain.topology.index_map(0).size_local
        + domain.topology.index_map(0).num_ghosts
    )
    coords = domain.geometry.x[:nverts, :2]
    tri = mtri.Triangulation(coords[:, 0], coords[:, 1], triangles=cells)
    return coords, tri


def plot_mesh(domain, ax=None):
    import numpy as np
    import matplotlib.pyplot as plt
    import matplotlib.tri as mtri

    if ax is None:
        fig, ax = plt.subplots()

    coords, tri = mesh_triangulation(domain)

    ax.triplot(tri, color="black", linewidth=0.4)
    ax.set_aspect("equal")
    ax.set_xlabel("x")
    ax.set_ylabel("y")

    return ax


fig, ax = plt.subplots()
plot_mesh(domain, ax=ax)
ax.set_title(f"32 x 32 / P1 / {V.dofmap.index_map.size_global} dofs")
plt.show()
```

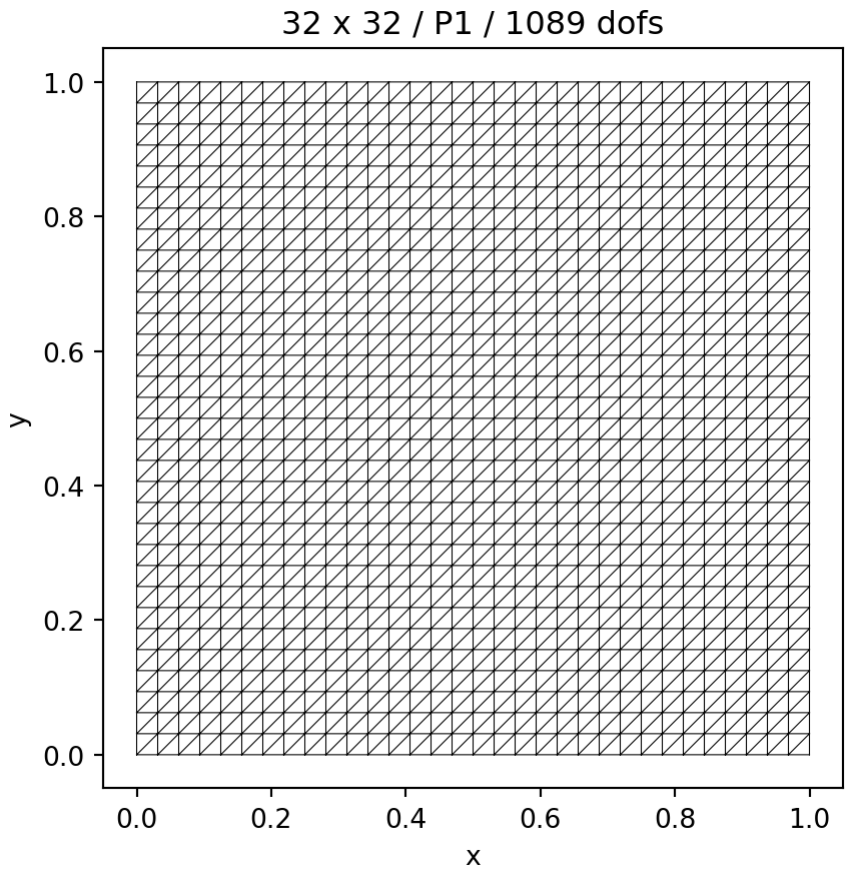


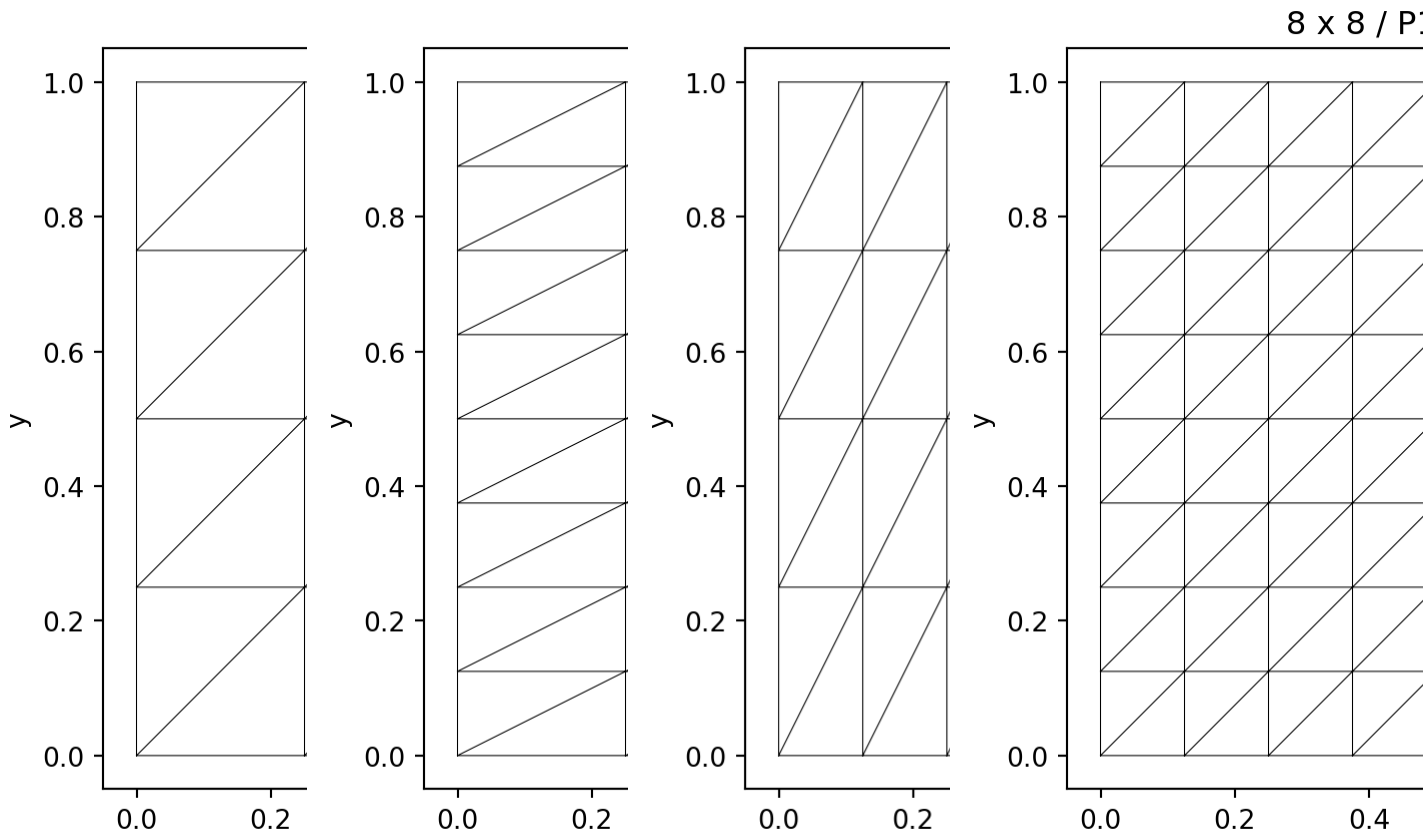
Figure 4: Triangular unit-square mesh.

2.4 Remember: Different mesh resolutions

We specify the mesh resolution of the unit square by the number of cells in each direction, i.e., `Nx` and `Ny`. Refining the mesh increases the number of degrees of freedom and gives the finite element space more flexibility.

> Reading the refinement plot
>
> Each panel uses the same domain and element family, but a different number of cells. The title reports the mesh size and the number of scalar `P1` degrees of freedom.

```
for Nx in [4, 8]:
    for Ny in [4, 8]:
        dom = mesh.create_unit_square(MPI.COMM_WORLD, Nx, Ny)
        V2 = fem.functionspace(dom, ("Lagrange", 1))

        fig, ax = plt.subplots()
        plot_mesh(dom, ax=ax)
        ax.set_title(f"{Nx} x {Ny} / P1 / {V2.dofmap.index_map.size_global} dofs")
        ax.set_aspect("equal")
        plt.show()
```



(a) Nx = 4, Ny = 4 (b) Nx = 4, Ny = 8 (c) Nx = 8, Ny = 4 (d) Nx = 8, Ny = 8

Figure 5: Effect of mesh resolution on the unit-square triangulation.

2.5 Manufactured exact solution and boundary condition

We use

$$u_{\text{exact}}(x, y) = 1 + x + 2y.$$

as a guess for the exact solution. The corresponding force term is therefore *manufactured* (reverse-engineered) by plugging u_{exact} into the PDE:

$$\begin{aligned} f &= -\nabla \cdot (q(u_{\text{exact}})\nabla u_{\text{exact}}) \\ &= -10(1 + x + 2y). \end{aligned}$$

> **!** Why manufacture a solution?
>
> Because u_{exact} is known, we can verify the numerical result quantitatively instead of relying only on visual inspection.

```
u_vals = np.linspace(0, 4, 200)
q_vals = 1 + u_vals**2

fig, ax = plt.subplots()
ax.plot(u_vals, q_vals)
ax.set_xlabel("u")
ax.set_ylabel("q(u)")
ax.grid(True, alpha=0.3)
plt.show()
```

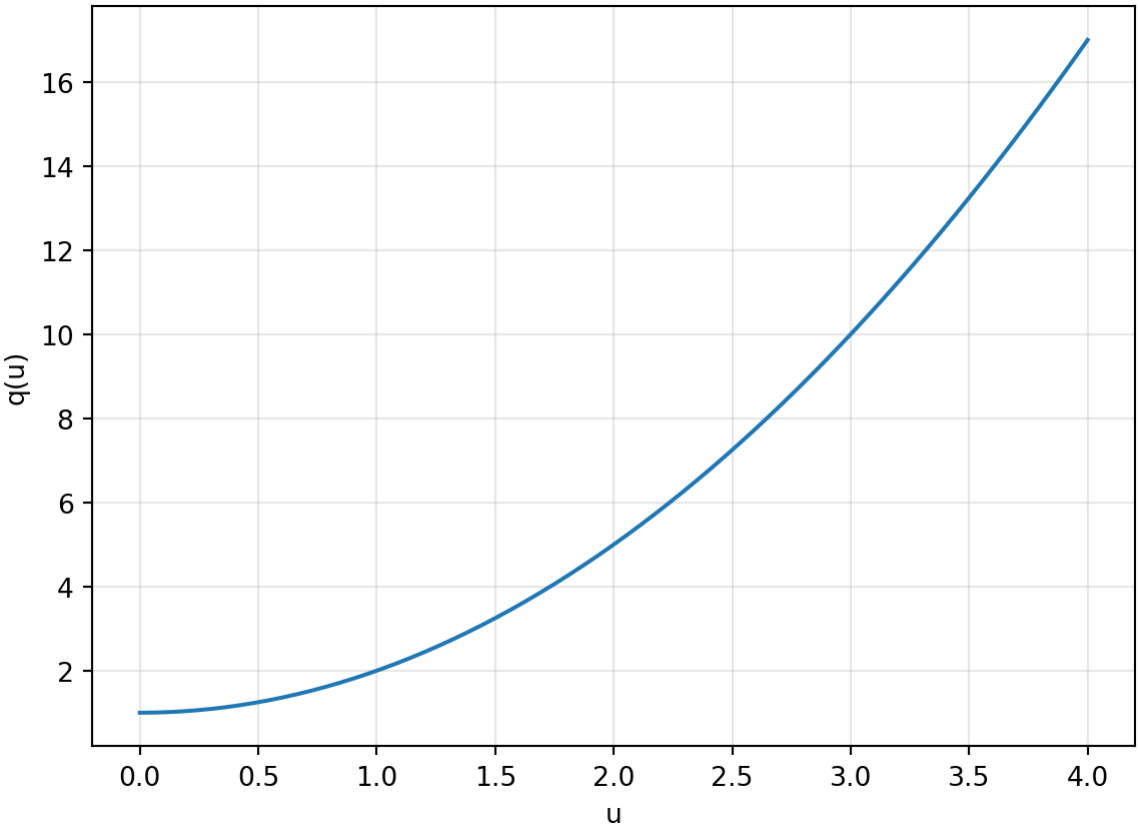


Figure 6: Nonlinear diffusion coefficient $q(u) = 1 + u^2$.

2.5.1 Exact solution

```
coords, tri = mesh_triangulation(domain)
z_exact = 1 + coords[:, 0] + 2 * coords[:, 1]

fig = plt.figure(figsize=(9, 4), constrained_layout=True)

# 2D contour/field plot
ax0 = fig.add_subplot(1, 2, 1)
p0 = ax0.tripcolor(tri, z_exact, shading="gouraud", cmap="viridis")
levels = np.linspace(z_exact.min(), z_exact.max(), 9)
c0 = ax0.tricontour(tri, z_exact, levels=levels, colors="white", linewidths=0.8)
ax0.clabel(c0, inline=True, fontsize=8, fmt="%.1f")
fig.colorbar(p0, ax=ax0, shrink=0.85)

ax0.set_title(r"Contour plot of $u_\mathrm{exact}$")
ax0.set_xlabel("x")
ax0.set_ylabel("y")
ax0.set_aspect("equal")

# 3D height-field plot
```

```
ax1 = fig.add_subplot(1, 2, 2, projection="3d")
p1 = ax1.plot_trisurf(
    tri,
    z_exact,
    cmap="viridis",
    linewidth=0.2,
    edgecolor="black",
    alpha=0.95,
)
fig.colorbar(p1, ax=ax1, shrink=0.65, pad=0.08)

ax1.set_title(r"Height field of $u_\mathrm{exact}$")
ax1.set_xlabel("x")
ax1.set_ylabel("y")
ax1.set_zlabel(r"$u_\mathrm{exact}$")
ax1.view_init(elev=28, azim=-135)

plt.show()
```

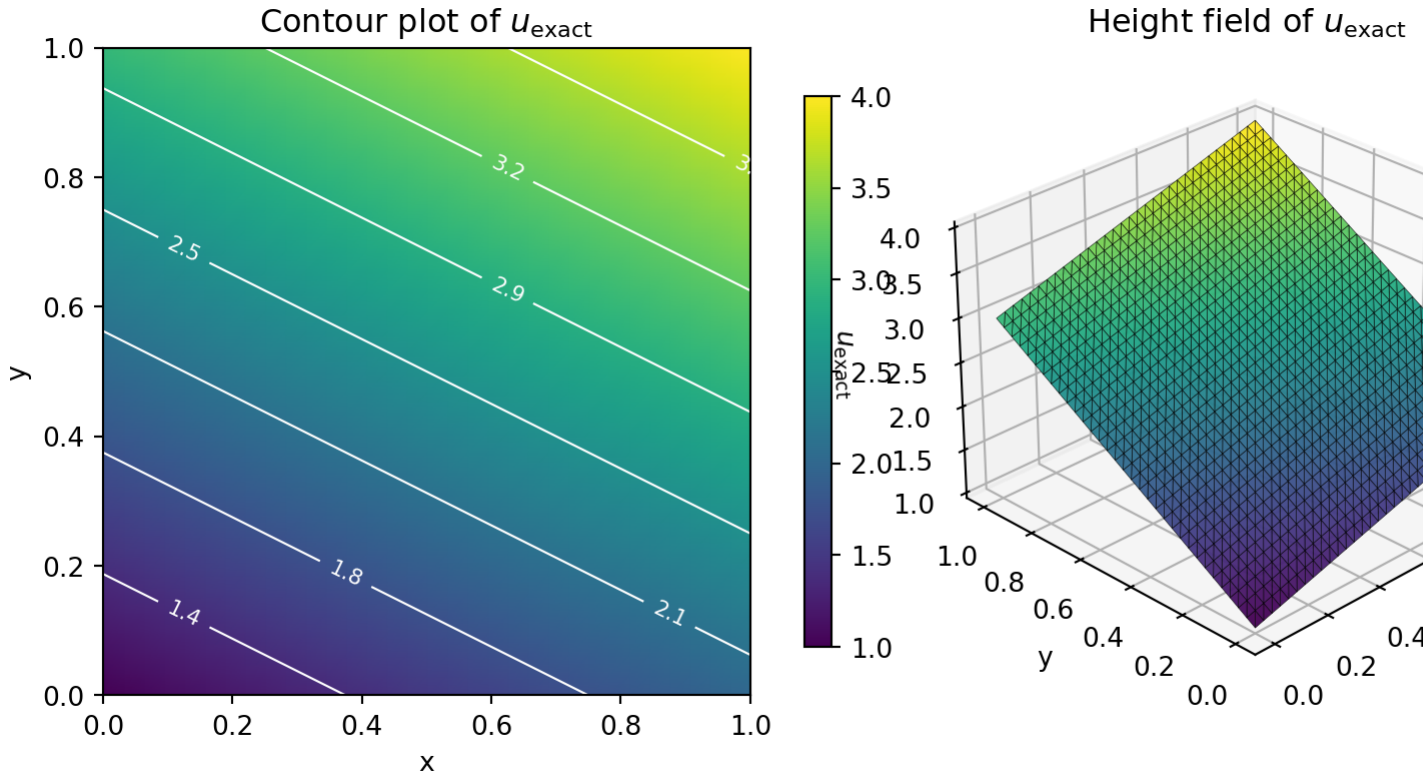


Figure 7: Manufactured exact solution.

2.5.2 Setup BCs and initial guess

```
from ufl.algorithms import expand_indices, expand_derivatives
x = ufl.SpatialCoordinate(domain)

u_exact = 1 + x[0] + 2 * x[1]

def q(u):
    return 1 + u**2

f = -ufl.div(q(u_exact) * ufl.grad(u_exact))

u_D = fem.Function(V)
u_D.interpolate(lambda x: (1 + x[0] + 2 * x[1]).astype(PETSc.ScalarType))

fdim = domain.topology.dim - 1
boundary_facets = mesh.locate_entities_boundary(
    domain,
    fdim,
    lambda x: np.full(x.shape[1], True, dtype=bool),
)
boundary_dofs = fem.locate_dofs_topological(V, fdim, boundary_facets)
bc = fem.dirichletbc(u_D, boundary_dofs)
```

2.6 UFL Symbolic manipulation example

- As you noted, we wrote something like this in the code:

```
f = -ufl.div(q(u_exact)*ufl.grad(u_exact))
```

But what does that actually do? Let's break it down:

1. `ufl.grad(u_exact)` computes the gradient of the exact solution, which is

$$\nabla u_{\text{exact}} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

2. `q(u_exact)` evaluates the nonlinear diffusion coefficient at the exact solution, giving

$$q(u_{\text{exact}}) = 1 + (1 + x + 2y)^2.$$

---

3. `q(u_exact)*ufl.grad(u_exact)` multiplies the diffusion coefficient by the gradient, resulting in

$$q(u_{\text{exact}})\nabla u_{\text{exact}} = \begin{bmatrix} 1 + (1 + x + 2y)^2 \\ 2(1 + (1 + x + 2y)^2) \end{bmatrix}.$$

4. Finally, `ufl.div(...)` computes the divergence of the above vector field, yielding

$$
\begin{aligned}
&\nabla \cdot (q(u_{\text{exact}})\nabla u_{\text{exact}}) \\
&= \frac{\partial}{\partial x}\left(1 + (1 + x + 2y)^2\right) \\
&+ \frac{\partial}{\partial y}\left(2(1 + (1 + x + 2y)^2)\right) \\
&= 2(1 + x + 2y) + 4(1 + x + 2y) \\
&= 10(1 + x + 2y).
\end{aligned}
$$

5. The negative sign in front of `ufl.div(...)` gives us the final expression for the manufactured force term.

- We can check that also in the code:

2.6.1 Inspecting the manufactured source term

The source term is defined symbolically in UFL as

$$
f = -\nabla \cdot (q(u_{\text{exact}})\nabla u_{\text{exact}}).
$$

For $u_{\text{exact}} = 1 + x + 2y$, we have $\nabla u_{\text{exact}} = (1, 2)$ and $q(u) = 1 + u^2$. Thus

$$
\begin{aligned}
f &= -\left(2u_{\text{exact}}\, 1^2 + 2u_{\text{exact}}\, 2^2\right) \\
&= -10(1 + x + 2y).
\end{aligned}
$$

The following cell shows how UFL stores the expression before and after symbolic expansion.

```
f_expanded_derivatives = expand_derivatives(f)
f_expanded_indices = expand_indices(f_expanded_derivatives)
f_expected = -10 * (1 + x[0] + 2 * x[1])
f_expected_expanded = expand_indices(expand_derivatives(f_expected))

print("1) Compact UFL expression:")
print(f)

print("\n2) After expanding derivatives:")
print(f_expanded_derivatives)

print("\n3) After expanding tensor indices:")
print(f_expanded_indices)

print("\n4) Hand-derived expression:")
print(f_expected_expanded)

print("\nThe manufactured source is f = -10 * (1 + x[0] + 2*x[1]).")
```

```
1) Compact UFL expression:
-1 * (div({ A | A_{i_8} = (grad(1 + x[0] + 2 * x[1]))[i_8] * (1 + (1 + x[0] + 2 * x[1]) ** 2)

2) After expanding derivatives:
-1 * (sum_{i_9} ({ A | A_{i_8, i_{17}} = ({ A | A_{i_{16}} = ({ A | A_{i_{15}} = ({ A | A_{i_{

3) After expanding tensor indices:
-1 * (2 * (1 + x[0] + 2 * x[1]) + 2 * 4 * (1 + x[0] + 2 * x[1]))

4) Hand-derived expression:
-10 * (1 + x[0] + 2 * x[1])

The manufactured source is f = -10 * (1 + x[0] + 2*x[1]).
```

2.7 Unknown, test function, residual

For nonlinear problems, the unknown is a `fem.Function`, not a `TrialFunction`.

> **i** Residual form
>
> The weak form is written as a residual $F(u_h; v) = 0$ for all test functions v. Newton's method repeatedly linearizes this residual around the current iterate.

```python
uh = fem.Function(V)
uh.name = "u"

# Initial guess. This does not have to equal the boundary data.
uh.interpolate(lambda x: np.ones(x.shape[1], dtype=PETSc.ScalarType))

v = ufl.TestFunction(V)

F = q(uh) * ufl.inner(ufl.grad(uh), ufl.grad(v)) * ufl.dx - f * v * ufl.dx
```

2.8 Explicit Jacobian

This is the manually supplied Gâteaux derivative of the residual with respect to `uh`.

Our weak residual is

$$F(u; v) = \int_\Omega q(u)\nabla u \cdot \nabla v \, dx - \int_\Omega f v \, dx,$$

This is the same Gâteaux derivative as in Lecture 02. The only difference is that we now differentiate the residual form $F(u; v)$ instead of an energy functional, and the test function v is held fixed while δu plays the role of the perturbation direction.

Equivalently, using the limit definition from Lecture 02,

$$J(u; \delta u, v) = \lim_{\varepsilon \to 0} \frac{F(u + \varepsilon \delta u; v) - F(u; v)}{\varepsilon} = \frac{d}{d\varepsilon} F(u + \varepsilon \delta u; v)\Big|_{\varepsilon=0}.$$

So the derivative can be written either as a limit quotient or as differentiation with respect to the scalar parameter ε; these are the same definition.

Using the derivative notation, the Gâteaux derivative in direction δu is

Insert $u + \varepsilon\delta u$ into the residual:

$$F(u + \varepsilon\delta u; v) = \int_\Omega q(u + \varepsilon\delta u)\, \nabla(u + \varepsilon\delta u) \cdot \nabla v \, dx - \int_\Omega f v \, dx.$$

The second integral does not depend on u, so its derivative is zero. For the first integral, use the product rule:

$$\frac{d}{d\varepsilon}\Big[q(u + \varepsilon\delta u)\, \nabla(u + \varepsilon\delta u)\Big] = \frac{d}{d\varepsilon} q(u + \varepsilon\delta u)\, \nabla(u + \varepsilon\delta u) + q(u + \varepsilon\delta u)\, \frac{d}{d\varepsilon}\nabla(u + \varepsilon\delta u).$$

Now evaluate at $\varepsilon = 0$:

$$\frac{d}{d\varepsilon} q(u + \varepsilon\delta u)\Big|_{\varepsilon=0} = q'(u)\, \delta u, \qquad \frac{d}{d\varepsilon}\nabla(u + \varepsilon\delta u)\Big|_{\varepsilon=0} = \nabla\delta u.$$

Therefore,

$$\begin{aligned} J(u; \delta u, v) = &\int_\Omega q(u)\nabla\delta u \cdot \nabla v \, dx \\ &+ \int_\Omega q'(u)\delta u\, \nabla u \cdot \nabla v \, dx. \end{aligned}$$

Since $q(u) = 1 + u^2$, we have $q'(u) = 2u$, so the second term becomes

$$\int_\Omega 2u\, \delta u\, \nabla u \cdot \nabla v \, dx.$$

This is the bilinear form solved in each Newton step for the correction δu.

Code mapping

`du` represents the Newton update direction δu. The two terms in `J_manual` correspond directly to the two integrals above.

Which PDE operator is this?

The Jacobian is the Fréchet derivative of the nonlinear diffusion operator

$$\mathcal{N}(u) = -\nabla \cdot (q(u)\nabla u).$$

For a Newton correction δu, the corresponding strong linearized operator is

$$\mathcal{N}'(u)[\delta u] = -\nabla \cdot (q(u)\nabla\delta u + q'(u)\delta u\, \nabla u)\,.$$

Thus, each Newton step solves a **linear variable-coefficient elliptic problem** for δu, with coefficients frozen at the current iterate u. The first term is diffusion with coefficient $q(u)$. The second term is not purely a convection term by itself; it is still written in divergence form.
If we define the frozen vector field

$$b(u) = q'(u)\nabla u,$$

then

$$-\nabla \cdot (q'(u)\delta u\, \nabla u) = -\nabla \cdot (\delta u\, b) = -b \cdot \nabla \delta u - (\nabla \cdot b)\, \delta u.$$

After expansion, this contributes a **first-order convection-like term** $-b \cdot \nabla \delta u$ and a **zeroth-order reaction term** $-(\nabla \cdot b)\, \delta u$. So the linearized problem can be viewed as diffusion plus lower-order terms.
For $q(u) = 1 + u^2$, we have $q'(u) = 2u$, so

$$-\nabla \cdot \left((1 + u^2)\nabla \delta u + 2u\, \delta u\, \nabla u\right).$$

Testing this operator with v and integrating by parts gives exactly the two integrals in $J(u; \delta u, v)$. It is not a new nonlinear PDE; it is the linear PDE solved inside one Newton step.

```
du = ufl.TrialFunction(V)

J_manual = (
    q(uh) * ufl.inner(ufl.grad(du), ufl.grad(v)) * ufl.dx
    + 2 * uh * du * ufl.inner(ufl.grad(uh), ufl.grad(v)) * ufl.dx
)
```

2.9 Infinite dimensional Newton's method

With that notation, we have the following algorithm:

1. Start with an initial guess u^0
2. For $k = 0, 1, 2, \ldots$ until convergence:
    1. Assemble the Jacobian $J(u^k; \delta u, v)$ and the residual $F(u^k; v)$
    2. Solve for the Newton update δu in $J(u^k; \delta u, v) = -F(u^k; v)$
    3. Update the solution: $u^{k+1} = u^k + \delta u$

2.10 Optional: automatic Jacobian for comparison

You can swap `J_manual` for `J_auto` in the `NonlinearProblem` constructor below.

Internally, UFL differentiates the residual expression symbolically and generates the same variational derivative.

i Manual vs automatic Jacobian

A manual Jacobian is useful for teaching and debugging. The automatic Jacobian is often preferable in production code because it avoids algebraic mistakes when the residual changes.

```
J_auto = ufl.derivative(F, uh, du)
```

```
# output J_auto
print(J_auto)

# showcase UFL derivative engine with a simple example
F_simple = ufl.sin(uh) * v * ufl.dx
J_simple = ufl.derivative(F_simple, uh, du)
print(J_simple)  # stored lazily

from ufl.algorithms.ad import expand_derivatives
J_expanded = expand_derivatives(J_simple)

print(J_expanded)
print(J_expanded.integrals()[0].integrand())

# print(ufl.cos(uh) * du * v * ufl.dx == J_expanded)
```

```
{ d/dfj { (1 + u ** 2) * (conj(((grad(v_0)) : (grad(u))))) }, with fh=ExprList(*(u,)), dfh/dfj
  +  { d/dfj { -1 * v_0 * -1 * (div({ A | A_{i_8} = (grad(1 + x[0] + 2 * x[1]))[i_8] * (1 + (1
{ d/dfj { v_0 * sin(u) }, with fh=ExprList(*(u,)), dfh/dfj = ExprList(*(v_1,)), and coefficien
{ v_0 * v_1 * cos(u) } * dx(<Mesh #0>[everywhere], {})
v_0 * v_1 * cos(u)
```

2.11 Solve with PETSc SNES Solver

The important line is:

```
problem = NonlinearProblem(F, uh, bcs=[bc], J=J_manual, ...)
```

That is where the explicit Jacobian is supplied.

> **!** Newton iteration
>
> The nonlinear solve repeatedly assembles the residual and Jacobian, solves a linearized correction problem, and updates the current approximation until the residual is small.

```
petsc_options = {
    "snes_type": "newtonls",
    "snes_linesearch_type": "bt",
    "snes_rtol": 1.0e-10,
    "snes_atol": 1.0e-10,
    "snes_max_it": 25,
    "snes_error_if_not_converged": True,
    "ksp_error_if_not_converged": True,
    "snes_view": None,
    "ksp_monitor": None,
    "snes_monitor": None,
```

```
    # Good for a small serial notebook example.
    # For MPI runs, you may need a parallel LU backend such as MUMPS.
    "ksp_type": "preonly",
    "pc_type": "lu",
    # "pc_factor_mat_solver_type": "mumps",
}
```

```
problem = NonlinearProblem(
    F, uh, bcs=[bc],
    # replace by J_auto or leave out entirely to let FEniCSx derive it
    J=J_manual,
    petsc_options_prefix="nlpoisson_",
    petsc_options=petsc_options,
)
uh = problem.solve()
assert problem.solver.getConvergedReason() > 0

print(f"Converged in {problem.solver.getIterationNumber()} iterations.")
```

```
  0 SNES Function norm 4.478564623279e+01
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 4.478564623279e+01
    1 KSP Residual norm 5.867542781294e-14
  1 SNES Function norm 4.633381267836e+00
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 4.633381267836e+00
    1 KSP Residual norm 9.959152302141e-14
  2 SNES Function norm 1.828141534723e+00
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 1.828141534723e+00
    1 KSP Residual norm 5.216204525426e-14
  3 SNES Function norm 2.306519444363e-01
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 2.306519444363e-01
    1 KSP Residual norm 4.826943267931e-15
  4 SNES Function norm 5.674866171688e-03
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 5.674866171688e-03
    1 KSP Residual norm 7.578138309229e-17
  5 SNES Function norm 3.108314093611e-06
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 3.108314093611e-06
    1 KSP Residual norm 1.689109889380e-20
  6 SNES Function norm 7.123153910601e-13
SNES Object: (nlpoisson_) 1 MPI process
  type: newtonls
  maximum iterations=25, maximum function evaluations=10000
  tolerances: relative=1e-10, absolute=1e-10, solution=1e-08
  total number of linear solver iterations=6
  total number of function evaluations=7
```

```
  norm schedule ALWAYS
  SNESLineSearch Object: (nlpoisson_) 1 MPI process
    type: bt
      interpolation: cubic
      alpha=1.000000e-04
    maxlambda=1.000000e+00, minlambda=1.000000e-12
    tolerances: relative=1.000000e-08, absolute=1.000000e-15, lambda=1.000000e-08
    maximum iterations=40
  KSP Object: (nlpoisson_) 1 MPI process
    type: preonly
    maximum iterations=10000, initial guess is zero
    tolerances: relative=1e-05, absolute=1e-50, divergence=10000.
    left preconditioning
    using NONE norm type for convergence test
  PC Object: (nlpoisson_) 1 MPI process
    type: lu
      out-of-place factorization
      tolerance for zero pivot 2.22045e-14
      matrix ordering: nd
      factor fill ratio given 5., needed 5.21682
        Factored matrix follows:
          Mat Object: (nlpoisson_) 1 MPI process
            type: seqaij
            rows=1089, cols=1089
            package used to perform factorization: petsc
            total: nonzeros=38401, allocated nonzeros=38401
              not using I-node routines
    linear system matrix = precond matrix:
    Mat Object: (nlpoisson_A_) 1 MPI process
      type: seqaij
      rows=1089, cols=1089
      total: nonzeros=7361, allocated nonzeros=7361
      total number of mallocs used during MatSetValues calls=0
        not using I-node routines
Converged in 6 iterations.
```

2.12 Plot solution

The first result plot shows the numerical solution u_h on the mesh. The surface plot is mainly a visual aid; the error check below is the actual verification.

```python
# plot
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(9, 4), constrained_layout=True)
import matplotlib.tri as mtri

coords, tri = mesh_triangulation(domain)
nverts = coords.shape[0]

vertex_ids = np.arange(nverts, dtype=np.int32)
dofs = fem.locate_dofs_topological(V, 0, vertex_ids)
```

```
uh_vertex = uh.x.array[dofs].real

ax0 = fig.add_subplot(1, 2, 1)
p0 = ax0.tripcolor(tri, uh_vertex, shading="gouraud", cmap="viridis")
fig.colorbar(p0, ax=ax0, shrink=0.85)
ax0.set_title(r"Field plot of $u_h$")
ax0.set_xlabel("x")
ax0.set_ylabel("y")
ax0.set_aspect("equal")

ax1 = fig.add_subplot(1, 2, 2, projection="3d")
p1 = ax1.plot_trisurf(tri, uh_vertex, cmap="viridis", linewidth=0.1, edgecolor="black")
fig.colorbar(p1, ax=ax1, shrink=0.65, pad=0.08)
ax1.set_title(r"Surface plot of $u_h$")
ax1.set_xlabel("x")
ax1.set_ylabel("y")
ax1.set_zlabel(r"$u_h$")
ax1.view_init(elev=28, azim=-135)

plt.show()
```

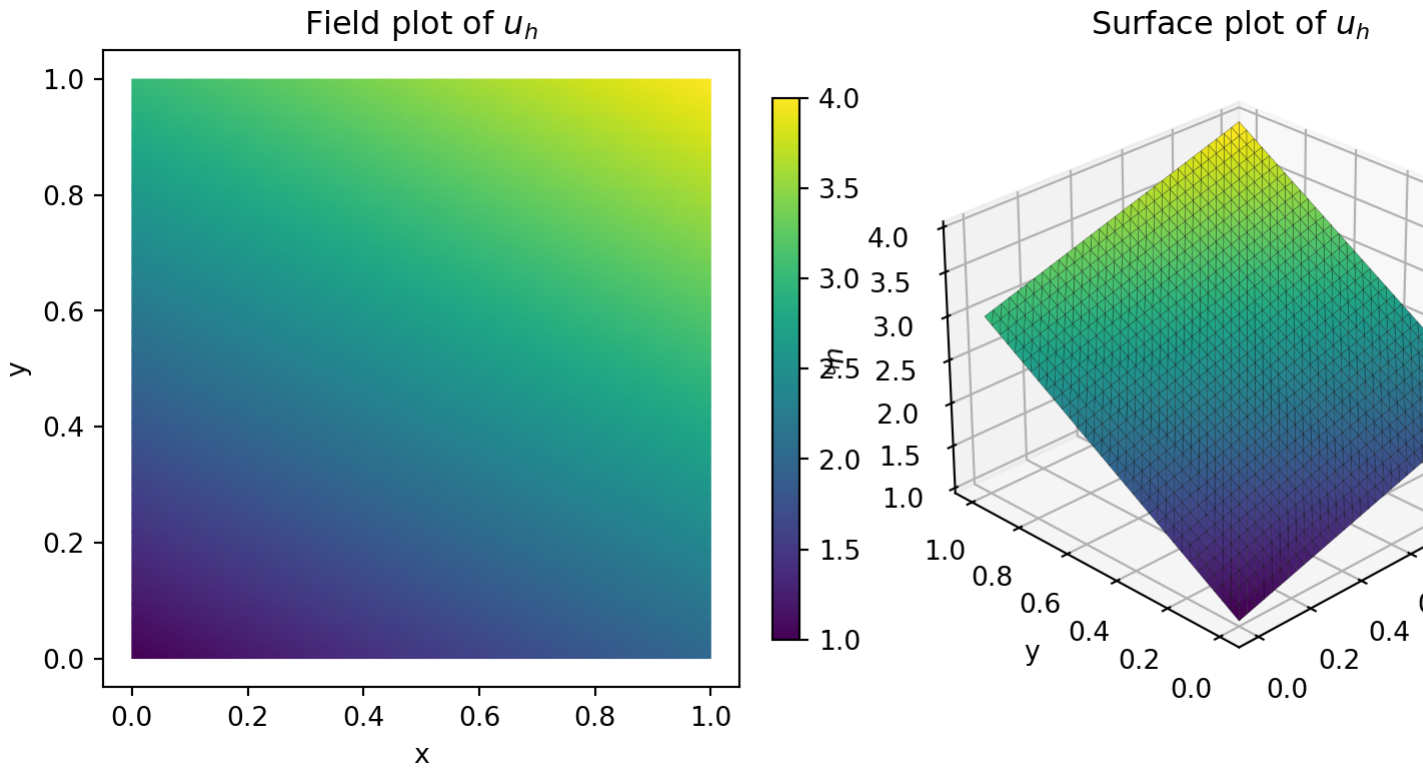


Figure 8: Computed nonlinear Poisson solution u_h.

2.13 Error check

Since the exact solution is known from Section 2.5, we can compute error norms:

What to expect

For a linear manufactured solution and a `P1` space, the solution is represented very accurately. Remaining error is mainly due to solver tolerances and quadrature/assembly details.

For the error $e_h = u - u_h$, we calculated $\|e_h\|_{L^2}$, $|e|_{H^1} := \|\nabla e_h\|_{L^2}$, and $\|e_h\|_{H^1} := \sqrt{\|e_h\|_{L^2}^2 + |e_h|_{H^1}^2}$, i.e.

$$\|e_h\|_{L^2} = \left(\int_\Omega (e_h)^2 \, dx \right)^{1/2},$$

$$\|e_h\|_{H^1} = \left(\int_\Omega (e_h)^2 + \|\nabla e_h\|^2 \, dx \right)^{1/2}$$

2.13.1 Error calculation

```
L2_error_form = fem.form((uh - u_exact)**2 * ufl.dx)
L2_error_local = fem.assemble_scalar(L2_error_form)
L2_error = np.sqrt(domain.comm.allreduce(L2_error_local, op=MPI.SUM))

H1_semi_error_form = fem.form(
    ufl.inner(ufl.grad(uh - u_exact), ufl.grad(uh - u_exact)) * ufl.dx
)
H1_semi_error_local = fem.assemble_scalar(H1_semi_error_form)
H1_semi_error = np.sqrt(domain.comm.allreduce(H1_semi_error_local, op=MPI.SUM))

H1_full_error = np.sqrt(L2_error**2 + H1_semi_error**2)

if domain.comm.rank == 0:
    print(f"L2 error      = {L2_error:.3e}")
    print(f"H1-semi error = {H1_semi_error:.3e}")
    print(f"H1-full error = {H1_full_error:.3e}")
```

```
L2 error      = 9.194e-15
H1-semi error = 1.474e-13
H1-full error = 1.477e-13
```

2.13.2 Summary

```
u_exact_vertex = 1 + coords[:, 0] + 2 * coords[:, 1]
error_vertex = uh_vertex - u_exact_vertex

fig, axes = plt.subplots(1, 3, figsize=(9, 3), constrained_layout=True)
```

```
for ax, data, title in zip(
    axes,
    [uh_vertex, u_exact_vertex, error_vertex],
    [r"$u_h$", r"$u_\mathrm{exact}$", r"$u_h-u_\mathrm{exact}$"],
):
    cmap = "coolwarm" if "-" in title else "viridis"
    p = ax.tripcolor(tri, data, shading="gouraud", cmap=cmap)
    fig.colorbar(p, ax=ax, shrink=0.8)
    ax.set_title(title)
    ax.set_aspect("equal")

plt.show()
```

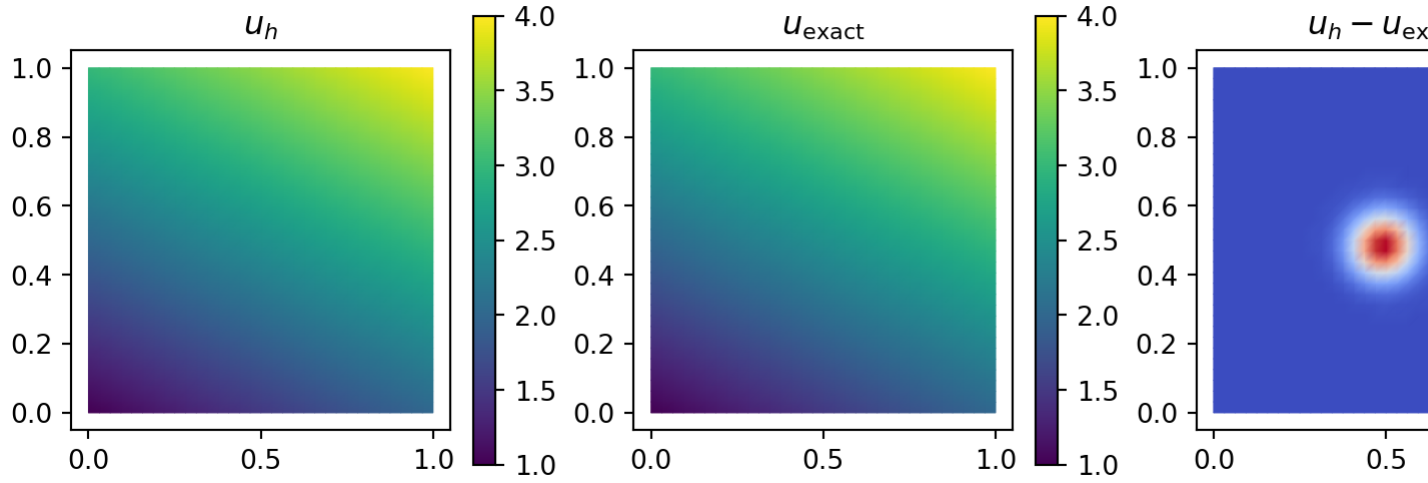


Figure 9: Numerical solution, exact solution, and pointwise error.

2.14 Optional: write solution to XDMF

> **External visualization**
>
> The XDMF output can be opened with ParaView for interactive inspection, slicing, and publication-quality rendering.

- There will be a small exercise session about how to use ParaView with FEniCSx output.

```
from dolfinx import io

with io.XDMFFile(domain.comm, "nonlinear_poisson_solution.xdmf", "w") as xdmf:
    xdmf.write_mesh(domain)
    xdmf.write_function(uh)
```

```
from pathlib import Path

Path("nonlinear_poisson_solution.xdmf").exists()
```

```
True
```

2.15 Questions / Exercises

1. Derive the Jacobian of a nonlinear reaction diffusion equation $-\Delta u + r(u) = f$ homogenous Dirichlet boundary conditions, where $r(u) = u^3$ is a nonlinear reaction term.

- Implement the above reaction diffusion equation in FEniCSx, using both a manual and an automatic Jacobian. Verify that the two Jacobians give the same solution and error norms.

3 Stokes with heat transport

We continue with the Stokes equations, which model slow, viscous, incompressible flow. We will encounter:

- **Gmsh** for more complex geometries + mesh generation (exercise)
- **Taylor–Hood elements (P2/P1)** for the Stokes problem.

3.1 Complex Part/Geometry in Gmsh

We use a plate with two cylinders and a side obstacle. This is complicated enough to demonstrate curved boundaries and tagged surfaces.

```python
import gmsh


def build_bracket_model(lc=0.01, terminal_output=1):
    gmsh.initialize()
    gmsh.model.add("bracket2d")
    occ = gmsh.model.occ

    gmsh.option.setNumber("General.Terminal", terminal_output)

    # Base plate (0.28m x 0.12m)
    plate = occ.addRectangle(0.0, 0.0, 0.0, 0.28, 0.12)

    # Two circular heat sources. The second one may intersect the side
    # cutout; the physical tags below are still recovered topologically.
    h1 = occ.addDisk(0.06, 0.06, 0.0, 0.012, 0.012)
    h2 = occ.addDisk(0.22, 0.06, 0.0, 0.012, 0.012)

    # Side cutout to imitate a clamp/jaw opening
    cut = occ.addRectangle(0.24, 0.035, 0.0, 0.04, 0.05)

    # Fragment first, then keep the plate fragments that are not part of
    # the disks or the cutout. This preserves the topology needed to tag
    # shared interfaces without guessing from coordinates.
    _, entity_map = occ.fragment([(2, plate)], [(2, h1), (2, h2), (2, cut)])
    occ.synchronize()
```

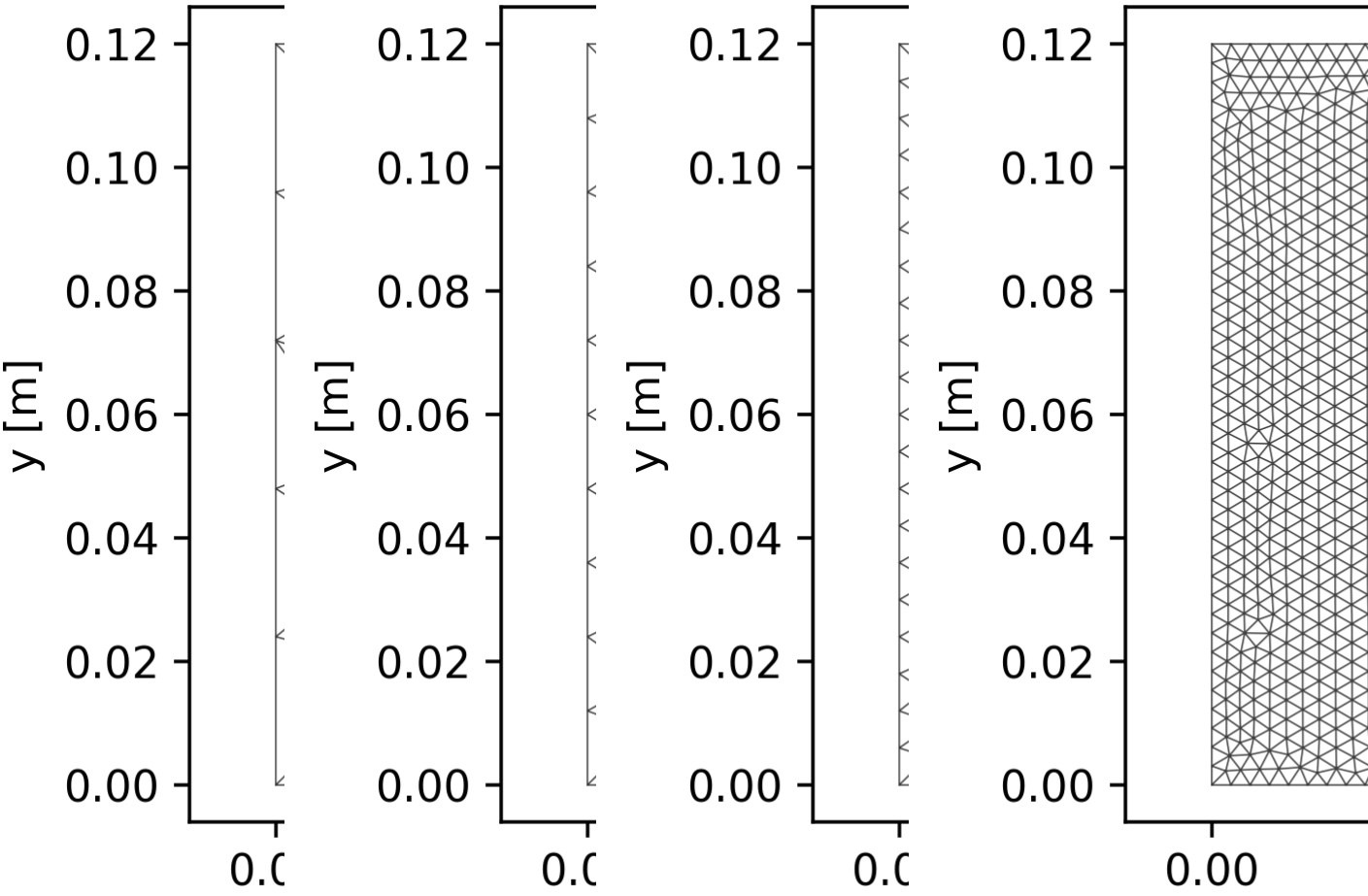
```
    hole_surfaces = set(entity_map[1]) | set(entity_map[2])
    cut_surfaces = set(entity_map[3])
    removed_surfaces = hole_surfaces | cut_surfaces
    domain_surfaces = [
        tag
        for dim, tag in entity_map[0]
        if dim == 2 and (dim, tag) not in removed_surfaces
    ]

    def boundary_curves(surface_entities):
        curves = set()
        for surface in surface_entities:
            for dim, tag in gmsh.model.getBoundary([surface], oriented=False):
                if dim == 1:
                    curves.add(tag)
        return curves

    domain_entities = [(2, tag) for tag in domain_surfaces]
    domain_curves = boundary_curves(domain_entities)
    hole_curves = boundary_curves(hole_surfaces)

    # The heated boundary is the part of the domain boundary that is shared
    # with a disk surface. This also works if a disk is split by the cutout.
    hot_wall = sorted(domain_curves & hole_curves)

    # The disk and cutout surfaces were only kept to identify shared curves.
    # Remove them before meshing, otherwise Gmsh also meshes the holes.
    occ.remove(list(removed_surfaces), recursive=False)
    occ.synchronize()

    remaining_curves = domain_curves - set(hot_wall)

    def curves_on_vertical_boundary(curves, x_value, tol=1e-5):
        selected = []
        for tag in curves:
            xmin, _, _, xmax, _, _ = gmsh.model.getBoundingBox(1, tag)
            if abs(xmin - x_value) < tol and abs(xmax - x_value) < tol:
                selected.append(tag)
        return sorted(selected)

    # Inlet and outlet are semantic geometric labels: the left and right
    # parts of the exterior boundary of the final fluid domain. The heated
    # hole boundary above is tagged purely topologically.
    left = curves_on_vertical_boundary(remaining_curves, 0.0)
    right = curves_on_vertical_boundary(remaining_curves, 0.28)
    wall = sorted(remaining_curves - set(left) - set(right))

    gmsh.model.addPhysicalGroup(2, domain_surfaces, 1)
    gmsh.model.setPhysicalName(2, 1, "solid")
    gmsh.model.addPhysicalGroup(1, left, 11)
```

```
    gmsh.model.setPhysicalName(1, 11, "left")
    gmsh.model.addPhysicalGroup(1, right, 12)
    gmsh.model.setPhysicalName(1, 12, "right")
    gmsh.model.addPhysicalGroup(1, wall, 13)
    gmsh.model.setPhysicalName(1, 13, "cold_walls")
    gmsh.model.addPhysicalGroup(1, hot_wall, 14)
    gmsh.model.setPhysicalName(1, 14, "heated_holes")

    gmsh.option.setNumber("Mesh.CharacteristicLengthMin", lc)
    gmsh.option.setNumber("Mesh.CharacteristicLengthMax", 2.5 * lc)

    gmsh.model.mesh.generate(2)

    return gmsh.model
```

3.1.1 Plot

```
# plot mesh
import matplotlib.pyplot as plt
import matplotlib.tri as mtri

for lcc in [0.01, 0.01/2, 0.01/4, 0.01/8]:

    model = build_bracket_model(lc=lcc, terminal_output=0)

    node_tags, node_coords, _ = model.mesh.getNodes()
    points = node_coords.reshape(-1, 3)[:, :2]
    node_to_index = {tag: i for i, tag in enumerate(node_tags)}

    element_types, _, element_node_tags = model.mesh.getElements(2)
    triangles = []
    for element_type, node_tags_for_type in zip(
        element_types,
        element_node_tags
    ):
        (
            name,
            dim,
            order,
            num_nodes,
            *_,
        ) = model.mesh.getElementProperties(element_type)
        if dim == 2 and num_nodes == 3:
            triangles.extend(
                [node_to_index[tag] for tag in triangle]
                for triangle in node_tags_for_type.reshape(-1, num_nodes)
            )
```

```
trian = mtri.Triangulation(points[:, 0], points[:, 1], triangles=triangles)

fig, ax = plt.subplots(figsize=(7, 3.5), dpi=160)
ax.triplot(trian, color="0.25", linewidth=0.35)
ax.set_aspect("equal", adjustable="box")
ax.set_xlabel("x [m]")
ax.set_ylabel("y [m]")
ax.set_title(f"mesh: {len(triangles)} triangles, {len(points)} vertices")
fig.tight_layout()
plt.show()

# gmsh.finalize()
```



(a) $l_c = 0.01$ (b) $l_c = 0.005$ (c) $l_c = 0.0025$ (d) $l_c = 0.00125$

Figure 10: Gmsh mesh for the bracket geometry before importing it into DOLFINx.

3.2 Import Mesh to FEniCSx

- `facet_tags` is essential for applying boundary conditions and boundary integrals.
- TODO

```
from mpi4py import MPI
import importlib

gmshio = None
for module_name in ("dolfinx.io.gmshio", "dolfinx.io.gmsh"):
    try:
        gmshio = importlib.import_module(module_name)
        break
    except ImportError:
        pass

if gmshio is None:
    raise ImportError("Could not import a DOLFINx Gmsh I/O module")
```

```
model = build_bracket_model(lc=0.004/4)
mesh_data = gmshio.model_to_mesh(model, MPI.COMM_WORLD, 0, gdim=2)
if isinstance(mesh_data, tuple):
    if len(mesh_data) < 3:
        raise ValueError("Unexpected model_to_mesh return format")
    # Newer DOLFINx may return extra mesh metadata after the first 3 values.
    msh, cell_tags, facet_tags, *_ = mesh_data
else:
    msh = mesh_data.mesh
    cell_tags = mesh_data.cell_tags
    facet_tags = mesh_data.facet_tags
gmsh.finalize()

# Physical tags from gmsh
SOLID = 1
LEFT = 11
RIGHT = 12
WALLS = 13
HOT_WALLS = 14
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 10%] Meshing curve 2 (Line)
Info    : [ 20%] Meshing curve 3 (Line)
Info    : [ 30%] Meshing curve 4 (Line)
Info    : [ 40%] Meshing curve 5 (Line)
Info    : [ 50%] Meshing curve 6 (Line)
Info    : [ 60%] Meshing curve 7 (Line)
Info    : [ 70%] Meshing curve 8 (Line)
Info    : [ 80%] Meshing curve 9 (Ellipse)
Info    : [ 90%] Meshing curve 10 (Ellipse)
Info    : [100%] Meshing curve 11 (Line)
```

```
Info    : Done meshing 1D (Wall 0.0013615s, CPU 0.001338s)
Info    : Meshing 2D...
Info    : Meshing surface 5 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.221435s, CPU 0.211938s)
Info    : 6087 nodes 12168 elements
```

3.3 Shared variational setup

We first define a small compatibility helper for `LinearProblem` and the integration measures on the imported Gmsh mesh. The flow solve comes first; the temperature solution below will then use the computed velocity field as an advective transport field.

```python
from petsc4py import PETSc
import inspect
import ufl
from dolfinx import fem
from dolfinx.fem.petsc import LinearProblem

def make_linear_problem(a, L, bcs, petsc_options, prefix):
    kwargs = {"petsc_options": petsc_options}
    if "petsc_options_prefix" in inspect.signature(LinearProblem).parameters:
        kwargs["petsc_options_prefix"] = prefix
    return LinearProblem(a, L, bcs=bcs, **kwargs)

dx = ufl.Measure("dx", domain=msh, subdomain_data=cell_tags)
ds = ufl.Measure("ds", domain=msh, subdomain_data=facet_tags)
fdim = msh.topology.dim - 1
```

3.4 One-way coupling workflow

The coupled example is solved in sequence:

1. solve Stokes for the velocity field u_h,
2. use u_h in an advection-diffusion equation for temperature,
3. heat the circular hole boundaries and keep the remaining exterior boundary cold.

3.5 Stokes Flow with Taylor–Hood Elements

We solve steady incompressible Stokes equations in stress form:

$$-\nabla \cdot \sigma(u, p) = f, \qquad \nabla \cdot u = 0,$$

with no-slip boundary conditions on solid walls and a prescribed inflow profile on the left boundary. For a Newtonian fluid,

$$\sigma(u, p) = 2\mu\varepsilon(u) - pI, \qquad \varepsilon(u) = \mathrm{sym}(\nabla u) = \tfrac{1}{2}(\nabla u + \nabla u^T).$$

The symmetric part is the strain-rate tensor. It measures stretching and shearing, while the anti-symmetric part of ∇u is local rigid-body rotation. Viscosity should dissipate deformation, not pure rotation.

For stability, we choose **Taylor–Hood** elements, see Brezzi and Fortin [2]:

- velocity: continuous quadratic P_2,
- pressure: continuous linear P_1.

The weak formulation reads: find $u \in V$ and $p \in Q$ such that

$$\int_\Omega 2\mu\,\varepsilon(u) : \varepsilon(v)\,dx - \int_\Omega p\,\nabla\cdot v\,dx = \int_\Omega f\cdot v\,dx,$$
$$-\int_\Omega q\,\nabla\cdot u\,dx = 0,$$

for all test functions $v \in V$ and $q \in Q$.

> **i** Why not just $\nabla u : \nabla v$?
>
> For constant viscosity and exactly incompressible velocity fields, the stress operator simplifies (see, e.g., [3] on footnote of page 4) because
>
> $$-\nabla\cdot(2\mu\varepsilon(u)) = -\mu\Delta u - \mu\nabla(\nabla\cdot u) = -\mu\Delta u.$$
>
> In that special case, one often writes the simpler Laplacian form $\mu\int_\Omega \nabla u : \nabla v\,dx$. The symmetric-gradient form is the physical stress form and is safer for variable viscosity, non-Newtonian models, traction boundary conditions, and stress interpretation. A standard reference is Girault and Raviart [4] [p80ff].

3.5.1 Implementation

```
from dolfinx import fem
from basix.ufl import element, mixed_element
import numpy as np
import ufl

# Mixed Taylor-Hood space W = V2 x Q1
Ve = element("Lagrange", msh.basix_cell(), 2, shape=(msh.geometry.dim,))
Qe = element("Lagrange", msh.basix_cell(), 1)
W = fem.functionspace(msh, mixed_element([Ve, Qe]))

(u, p) = ufl.TrialFunctions(W)
(v, q) = ufl.TestFunctions(W)

mu = fem.Constant(msh, PETSc.ScalarType(1.0e-3))
f = fem.Constant(msh, np.array((0.0, 0.0), dtype=np.float64))

```

```python
# Newtonian stress form: viscosity acts on the symmetric strain rate.
aS = (
    2.0 * mu * ufl.inner(ufl.sym(ufl.grad(u)), ufl.sym(ufl.grad(v))) * ufl.dx
    - ufl.div(v) * p * ufl.dx
    - q * ufl.div(u) * ufl.dx
)
LS = ufl.dot(f, v) * ufl.dx

# Boundary conditions
W0, _ = W.sub(0).collapse()  # velocity space
fdim = msh.topology.dim - 1

wall_facets = facet_tags.find(WALLS)
hot_wall_facets = facet_tags.find(HOT_WALLS)
left_facets = facet_tags.find(LEFT)
right_facets = facet_tags.find(RIGHT)

# No-slip at solid walls, including the heated circular holes.
no_slip_facets = np.concatenate([wall_facets, hot_wall_facets])
wall_dofs = fem.locate_dofs_topological((W.sub(0), W0), fdim, no_slip_facets)
zero = fem.Function(W0)
zero.x.array[:] = 0.0
bc_wall = fem.dirichletbc(zero, wall_dofs, W.sub(0))

# Parabolic inflow profile at LEFT
inlet = fem.Function(W0)
inlet.interpolate(
    lambda x: np.vstack(
        (1.0e-2 * 4.0 * x[1] * (0.12 - x[1]) / 0.12**2, np.zeros_like(x[1]))
        )
)
left_dofs_u = fem.locate_dofs_topological((W.sub(0), W0), fdim, left_facets)
bc_inlet = fem.dirichletbc(inlet, left_dofs_u, W.sub(0))

# Pressure reference at RIGHT: p=0
right_dofs_p = fem.locate_dofs_topological(W.sub(1), fdim, right_facets)
bc_outlet_p = fem.dirichletbc(PETSc.ScalarType(0.0), right_dofs_p, W.sub(1))

stokes_problem = make_linear_problem(
    aS,
    LS,
    bcs=[bc_wall, bc_inlet, bc_outlet_p],
    prefix="stokes03_",
    petsc_options={"ksp_type": "minres", "pc_type": "lu"},
)
wh = stokes_problem.solve()
uh, ph = wh.split()
uh.name = "velocity"
ph.name = "pressure"
```

3.6 Temperature transported by the Stokes flow

We now solve a steady advection-diffusion equation for the temperature field. The velocity u_h is the Stokes solution computed above, so this is a **one-way coupled** thermo-fluid model:

$$u_h \cdot \nabla T - \nabla \cdot (\alpha \nabla T) = 0.$$

The two circular hole boundaries are heated, while the remaining exterior boundary is kept cold. We use a relatively small thermal diffusivity and a scaled advection term so that the cold inflow from the left visibly transports heat downstream.

> **One-way coupling**
>
> The flow affects the temperature through the advection term $u_h \cdot \nabla T$. The temperature does not feed back into the Stokes equations.

3.6.1 Implementation

```
V = fem.functionspace(msh, ("Lagrange", 1))
T = ufl.TrialFunction(V)
s = ufl.TestFunction(V)

# Thermal diffusivity. Smaller alpha makes the left-to-right transport
# by the Stokes velocity more visible in the temperature field.
alpha = fem.Constant(msh, PETSc.ScalarType(5.0e-5))

aT = (
    alpha * ufl.dot(ufl.grad(T), ufl.grad(s)) * dx(SOLID)
    + ufl.dot(uh, ufl.grad(T)) * s * dx(SOLID)
)
LT = fem.Constant(msh, PETSc.ScalarType(0.0)) * s * dx(SOLID)

hot_facets = facet_tags.find(HOT_WALLS)
cold_facets = np.concatenate([
    facet_tags.find(LEFT),
    facet_tags.find(RIGHT),
    facet_tags.find(WALLS),
])

hot_dofs = fem.locate_dofs_topological(V, fdim, hot_facets)
cold_dofs = fem.locate_dofs_topological(V, fdim, cold_facets)

bc_hot = fem.dirichletbc(PETSc.ScalarType(380.0), hot_dofs, V)
bc_cold = fem.dirichletbc(PETSc.ScalarType(300.0), cold_dofs, V)

temperature_problem = make_linear_problem(
    aT,
```

```
    LT,
    bcs=[bc_hot, bc_cold],
    prefix="advdiff03_",
    petsc_options={"ksp_type": "preonly", "pc_type": "lu"},
)
Th = temperature_problem.solve()
Th.name = "temperature"
```

3.7 Optional Output

The coupled fields can be written to XDMF together with the mesh for inspection in ParaView.

> **Website downloads**
>
> Each XDMF export creates two files: a small `.xdmf` metadata file and a paired `.h5` data file. Both files must be kept together. The course website is configured to upload `lectures/bracket_advective_heat_*.xdmf` and `lectures/bracket_advective_heat_*.h5`, so rerun this cell before publishing if the files are missing. The snippet below writes into `lectures/` when run from the repository root and into the current directory when run from inside `lectures/`.

```
from pathlib import Path
from dolfinx import io

output_dir = Path(".") if Path.cwd().name == "lectures" else Path("lectures")
output_dir.mkdir(exist_ok=True)

gdim = msh.geometry.dim
degree = msh.geometry.cmap.degree

V_out = fem.functionspace(msh, ("Lagrange", degree, (gdim,)))
uh_out = fem.Function(V_out)
uh_out.interpolate(uh)
uh_out.name = "velocity"

with io.XDMFFile(msh.comm, str(output_dir / "bracket_advective_heat_T.xdmf"), "w") as xdmf:
    xdmf.write_mesh(msh)
    xdmf.write_function(Th)
with io.XDMFFile(msh.comm, str(output_dir / "bracket_advective_heat_p.xdmf"), "w") as xdmf:
    xdmf.write_mesh(msh)
    xdmf.write_function(ph)
with io.XDMFFile(msh.comm, str(output_dir / "bracket_advective_heat_u.xdmf"), "w") as xdmf:
    xdmf.write_mesh(msh)
    xdmf.write_function(uh_out)
```

3.8 Simulation Output Plots

```
import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from mpl_toolkits.axes_grid1 import make_axes_locatable
from basix.ufl import element

tdim = msh.topology.dim
msh.topology.create_connectivity(tdim, 0)
msh.topology.create_connectivity(0, tdim)
cells = msh.topology.connectivity(tdim, 0).array.reshape(-1, 3)

nverts = (
    msh.topology.index_map(0).size_local
    + msh.topology.index_map(0).num_ghosts
)
coords = msh.geometry.x[:nverts, :2]
vertex_ids = np.arange(nverts, dtype=np.int32)
tri = mtri.Triangulation(coords[:, 0], coords[:, 1], triangles=cells)

xmin, xmax = coords[:, 0].min(), coords[:, 0].max()
ymin, ymax = coords[:, 1].min(), coords[:, 1].max()
pad = 0.01 * max(xmax - xmin, ymax - ymin)

def format_geometry_axes(ax):
    ax.set_xlim(xmin - pad, xmax + pad)
    ax.set_ylim(ymin - pad, ymax + pad)
    ax.set_aspect("equal", adjustable="box")
    ax.set_xlabel("x [m]")
    ax.set_ylabel("y [m]")


def boundary_edges_from_triangles(cells):
    edge_count = {}
    for cell in cells:
        for edge in (
            (cell[0], cell[1]), (cell[1], cell[2]), (cell[2], cell[0])
        ):
            edge = tuple(sorted(edge))
            edge_count[edge] = edge_count.get(edge, 0) + 1
    return np.array(
        [edge for edge, count in edge_count.items() if count == 1],
    dtype=np.int32)

boundary_edges = boundary_edges_from_triangles(cells)

def add_mesh_boundary(ax):
    for edge in boundary_edges:
        points = coords[edge]
        ax.plot(
            points[:, 0], points[:, 1], color="black", linewidth=0.9, zorder=5
        )
```

```
def scalar_at_vertices(f, Vh):
    dofs = fem.locate_dofs_topological(Vh, 0, vertex_ids)
    return f.x.array[dofs].real

def vector_at_vertices(f):
    Vvec = fem.functionspace(
        msh,
        element("Lagrange", msh.basix_cell(), 1, shape=(msh.geometry.dim,)),
    )
    f_p1 = fem.Function(Vvec)
    f_p1.interpolate(f)

    dofs = fem.locate_dofs_topological(Vvec, 0, vertex_ids)
    bs = Vvec.dofmap.bs
    return f_p1.x.array.reshape(-1, bs)[dofs, :].real

def add_matched_colorbar(fig, ax, mappable, label):
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.08)
    cbar = fig.colorbar(mappable, cax=cax)
    cbar.set_label(label)
    return cbar

def plot_scalar_panel(data, title, cmap, colorbar_label):
    fig, ax = plt.subplots(
        figsize=(6.0, 3.1), dpi=180, constrained_layout=False
    )
    pcm = ax.tripcolor(tri, data, shading="gouraud", cmap=cmap)
    ax.set_title(title)
    format_geometry_axes(ax)
    add_mesh_boundary(ax)
    add_matched_colorbar(fig, ax, pcm, colorbar_label)
    plt.show()

def plot_streamline_panel(u_vertex):
    fig, ax = plt.subplots(
        figsize=(6.0, 3.1), dpi=180, constrained_layout=False
    )

    interp_ux = mtri.LinearTriInterpolator(tri, u_vertex[:, 0])
    interp_uy = mtri.LinearTriInterpolator(tri, u_vertex[:, 1])

    x_grid = np.linspace(xmin, xmax, 240)
    y_grid = np.linspace(ymin, ymax, 120)
    X, Y = np.meshgrid(x_grid, y_grid)
    U = interp_ux(X, Y)
    W = interp_uy(X, Y)
    speed = np.sqrt(U**2 + W**2)

    lines = ax.streamplot(
```

```
        X,
        Y,
        U,
        W,
        color=speed,
        cmap="viridis",
        density=1.9,
        linewidth=0.9,
        arrowsize=0.75,
    )
    add_mesh_boundary(ax)
    ax.set_title(r"Velocity streamlines")
    format_geometry_axes(ax)
    add_matched_colorbar(fig, ax, lines.lines, "m/s")
    plt.show()

T_vertex = scalar_at_vertices(Th, V)
p_vertex = scalar_at_vertices(ph, ph.function_space)
u_vertex = vector_at_vertices(uh)

Vmag = fem.functionspace(msh, ("Lagrange", 1))
u_mag = fem.Function(Vmag)
interp_points = Vmag.element.interpolation_points
if callable(interp_points):
    interp_points = interp_points()
u_mag_expr = fem.Expression(ufl.sqrt(ufl.inner(uh, uh)), interp_points)
u_mag.interpolate(u_mag_expr)
u_mag_vertex = scalar_at_vertices(u_mag, Vmag)

plot_scalar_panel(T_vertex, r"Temperature $T_h$", "inferno", "K")
plot_scalar_panel(p_vertex, r"Pressure $p_h$", "coolwarm", "pressure")
plot_scalar_panel(u_mag_vertex, r"Velocity magnitude $|u_h|$", "viridis", "m/s")
plot_streamline_panel(u_vertex)
```

4 Linear elasticity

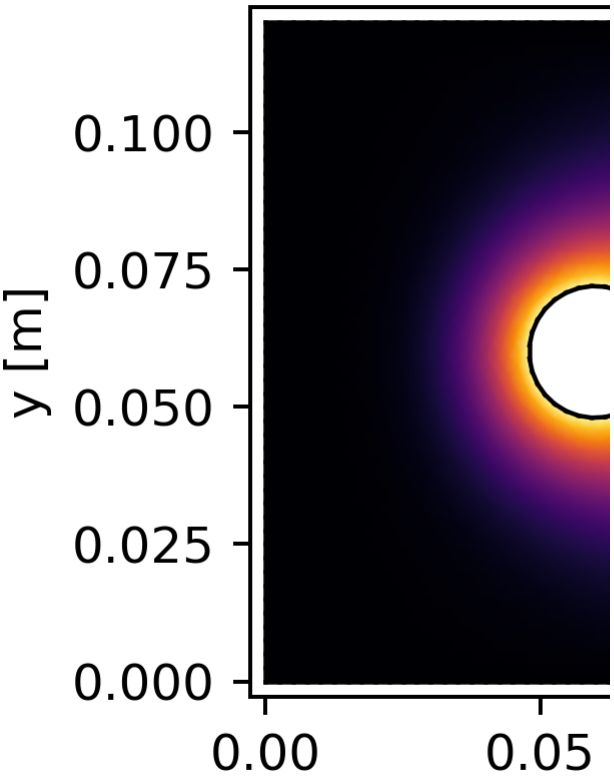
- TODO
- See very good material by Jeremy Bleyer

5 Schrödinger's equation

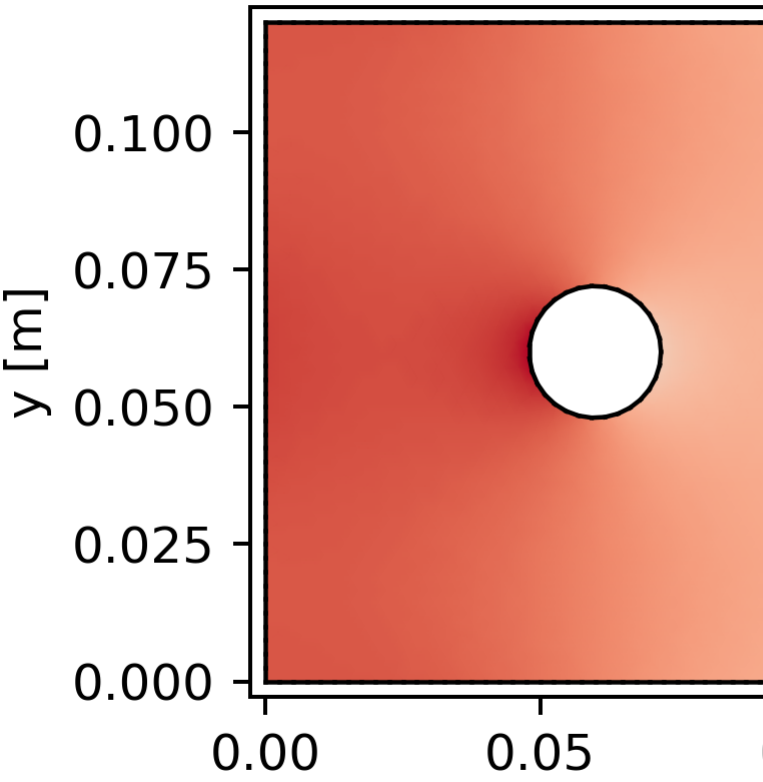
We now turn from boundary value problems to **eigenvalue problems (EVPs)**. A canonical example from quantum mechanics is the time-independent Schrödinger equation

$$\hat{H}\,\psi = E\,\psi, \qquad \hat{H} = -\tfrac{1}{2}\Delta + V(\mathbf{x}),$$

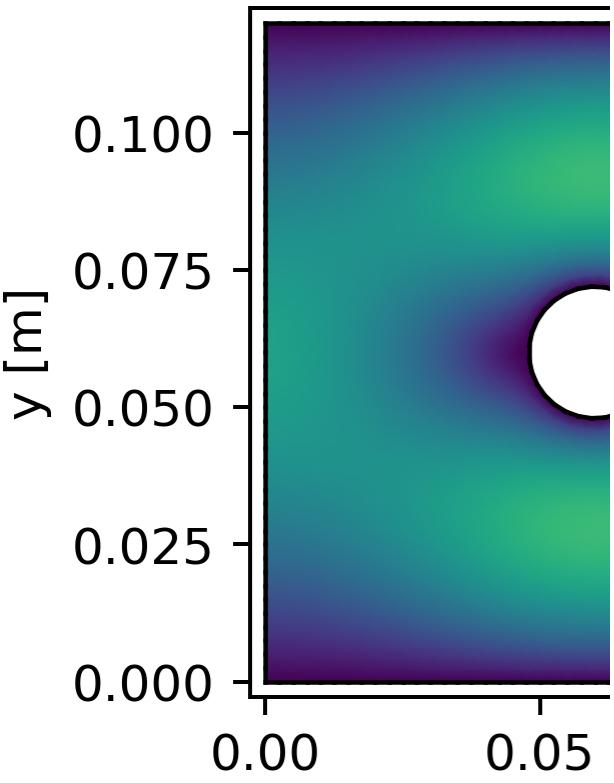
where we use atomic units ($\hbar = m = 1$). The eigenvalues E_n are the allowed energies and the eigenfunctions ψ_n are the stationary states.



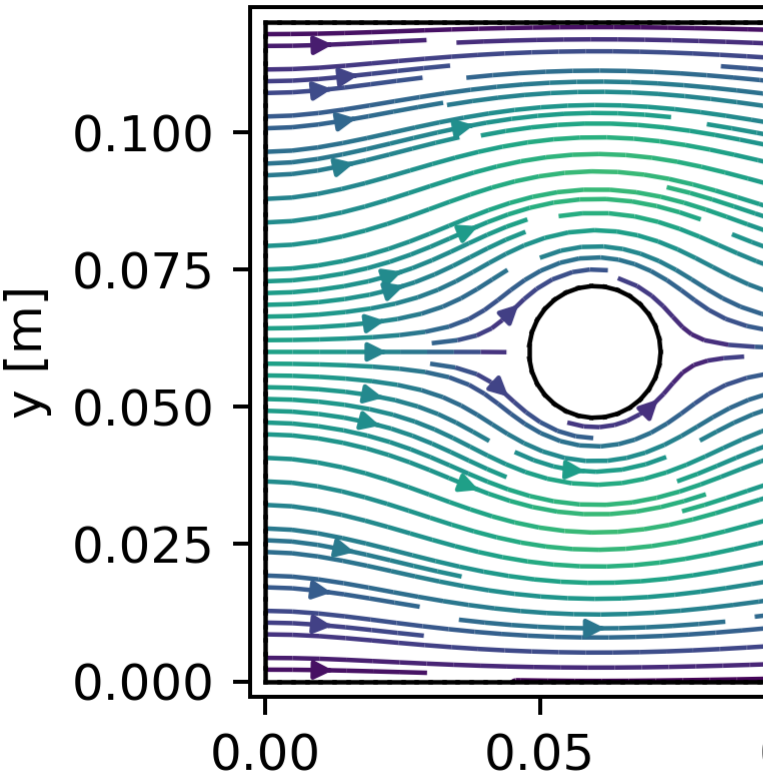
(a) Temperature



(b) Pressure



(c) Velocity magnitude



(d) Velocity streamlines

5.1 Weak form and generalized EVP

Multiplying by a test function $v \in H_0^1(\Omega)$ and integrating by parts gives: find $\psi \in H_0^1(\Omega)$ and $E \in \mathbb{R}$ such that

$$\underbrace{\int_\Omega \tfrac{1}{2}\nabla\psi \cdot \nabla v + V\,\psi\, v \,\mathrm{d}x}_{a(\psi,v)} = E \underbrace{\int_\Omega \psi\, v \,\mathrm{d}x}_{m(\psi,v)} \quad \forall\, v \in H_0^1(\Omega).$$

After discretization with FE basis $\{\varphi_i\}$ this becomes a **generalized matrix eigenvalue problem**

$$A\,\psi = E\,M\,\psi,$$

with stiffness-plus-potential matrix A and mass matrix M. Both are sparse, symmetric, and we are typically interested in only the few smallest eigenvalues — the perfect setting for **SLEPc**, the eigenvalue companion to PETSc.

More information about FEM for EVPs, numerical algorithms can be found in my two papers [5] and [6].

5.2 Test problem: 2D quantum harmonic oscillator

We pick $V(x, y) = \frac{1}{2}(x^2 + y^2)$ on a sufficiently large box $\Omega = [-L, L]^2$ with $\psi = 0$ on $\partial\Omega$. The exact eigenvalues (separation yield two 1D harmonic oscillators [7], then add them) are

$$E_{n_x,n_y} = n_x + n_y + 1, \qquad n_x, n_y \in \{0, 1, 2, \dots\},$$

i.e. 1, 2, 2, 3, 3, 3, ... (note the degeneracies). This gives us a cheap verification: the FE eigenvalues should converge to these integers as the mesh is refined and L is large enough that the truncation of $\mathbb{R}^2$ to the box is harmless.

> **Why SLEPc and not `numpy.linalg.eig`?**
>
> Assembling A and M as dense matrices and calling a dense solver scales as $\mathcal{O}(N^3)$ in storage and time. SLEPc uses **Krylov methods** (Krylov–Schur by default) that only need matrix–vector products, so we can extract the lowest few eigenpairs in $\mathcal{O}(N)$-ish work.

> **Dirichlet dofs in a generalized EVP**
>
> The discrete problem is generalized,
>
> $$A\psi = EM\psi.$$
>
> Dirichlet boundary dofs are fixed and should not become physical eigenmodes. When the boundary condition is imposed algebraically, the constrained rows and columns are zeroed and a unit diagonal is inserted. If this happens in both matrices, a vector supported only on one constrained boundary dof satisfies

$$Ae_i = e_i, \qquad Me_i = e_i,$$

and therefore gives the artificial eigenvalue $E = 1$. This is especially bad here because the true harmonic-oscillator ground-state energy is also 1.
We keep the unit diagonal in A, but set the corresponding diagonal entries of M to zero. Then these boundary-only vectors no longer define finite eigenvalues and do not pollute the low-energy spectrum.

5.2.1 Implementation

```python
import numpy as np
from mpi4py import MPI
from petsc4py import PETSc
from slepc4py import SLEPc
import ufl
from dolfinx import fem, mesh as dmesh

# Domain [-L, L]^2, large enough that the bound states decay before the wall.
L = 6.0
N = 64
msh_qho = dmesh.create_rectangle(
    MPI.COMM_WORLD,
    [np.array([-L, -L]), np.array([L, L])],
    [N, N],
    cell_type=dmesh.CellType.triangle,
)

V_qho = fem.functionspace(msh_qho, ("Lagrange", 1))
psi = ufl.TrialFunction(V_qho)
v = ufl.TestFunction(V_qho)

# Harmonic potential V(x,y) = 1/2 (x^2 + y^2).
x = ufl.SpatialCoordinate(msh_qho)
V_pot = 0.5 * (x[0]**2 + x[1]**2)

a = (0.5 * ufl.dot(ufl.grad(psi), ufl.grad(v)) + V_pot * psi * v) * ufl.dx
m = psi * v * ufl.dx

# Homogeneous Dirichlet on the box boundary: psi = 0.
tdim = msh_qho.topology.dim
msh_qho.topology.create_connectivity(tdim - 1, tdim)
boundary_facets = dmesh.exterior_facet_indices(msh_qho.topology)
boundary_dofs = fem.locate_dofs_topological(V_qho, tdim - 1, boundary_facets)
bc = fem.dirichletbc(PETSc.ScalarType(0.0), boundary_dofs, V_qho)

# Assemble A and M. Both get rows/cols of constrained DOFs zeroed with 1 on
# the diagonal. To avoid spurious eigenvalues at lambda = 1 colliding with
```

```python
# the true ground state E0 = 1, we then overwrite M's BC diagonal with 0 -
# the spurious eigenvalues are pushed to infinity and a target near 0 ignores
# them.
from dolfinx.fem.petsc import assemble_matrix
A = assemble_matrix(fem.form(a), bcs=[bc])
A.assemble()
M = assemble_matrix(fem.form(m), bcs=[bc])
M.assemble()

bc_dofs = np.asarray(boundary_dofs, dtype=PETSc.IntType)
diag = M.getDiagonal()
diag.setValues(bc_dofs, np.zeros(bc_dofs.size, dtype=PETSc.ScalarType))
diag.assemble()
M.setDiagonal(diag)
M.assemblyBegin(); M.assemblyEnd()

# Configure SLEPc: smallest eigenvalues of a generalized Hermitian EVP.
eps = SLEPc.EPS().create(MPI.COMM_WORLD)
eps.setOperators(A, M)
eps.setProblemType(SLEPc.EPS.ProblemType.GHEP)  # A = A^T, M SPD on free DOFs
eps.setWhichEigenpairs(SLEPc.EPS.Which.TARGET_REAL)
eps.setTarget(0.0)                                  # look near zero
eps.setDimensions(nev=8)                            # want 8 lowest

# Shift-and-invert spectral transform makes "smallest" tractable for Krylov.
st = eps.getST()
st.setType(SLEPc.ST.Type.SINVERT)
ksp = st.getKSP()
ksp.setType("preonly")
ksp.getPC().setType("lu")

eps.setFromOptions()
eps.solve()

nconv = eps.getConverged()
print(f"SLEPc converged eigenvalues: {nconv}")

eigvals = np.array([eps.getEigenvalue(i).real for i in range(nconv)])
eigvals.sort()
print("First 8 FE eigenvalues:   ", np.round(eigvals[:8], 4))
print("Exact QHO eigenvalues:    ", [1, 2, 2, 3, 3, 3, 4, 4])
```

```
SLEPc converged eigenvalues: 10
First 8 FE eigenvalues:    [1.0037 2.0066 2.0153 3.0117 3.0196 3.0377 4.0189 4.0261]
Exact QHO eigenvalues:     [1, 2, 2, 3, 3, 3, 4, 4]
```

5.3 Verification

The first FE eigenvalues should be close to the exact harmonic-oscillator spectrum 1, 2, 2, 3, 3, 3, 4, 4, ... Two error sources control the accuracy:

1. **Domain truncation.** We replaced $\mathbb{R}^2$ by the box $[-L, L]^2$ with $\psi = 0$ on the wall. The ground state $\psi_{0,0} \propto e^{-(x^2+y^2)/2}$ has decayed to $\sim e^{-L^2/2}$, so $L = 6$ gives $\sim 10^{-8}$ — well below discretization error.
2. **FE discretization.** With P_1 elements on a uniform mesh the eigenvalue error is $\mathcal{O}(h^2)$. Try increasing `N` (or switching to `("Lagrange", 2)`) and watch the lower eigenvalues tighten onto integers.

5.4 Plotting the lowest eigenfunctions

```
import matplotlib.pyplot as plt
import matplotlib.tri as mtri

# Sort eigenpairs by eigenvalue, then extract eigenvectors as Functions.
order = np.argsort([eps.getEigenvalue(i).real for i in range(nconv)])

vr, _ = A.getVecs()
psi_funcs = []
energies = []
for k in order[:6]:
    eps.getEigenpair(k, vr)
    f = fem.Function(V_qho)
    f.x.array[:] = vr.array[:].real
    # Normalize so max|psi| = 1 for plotting.
    f.x.array[:] /= np.max(np.abs(f.x.array)) or 1.0
    psi_funcs.append(f)
    energies.append(eps.getEigenvalue(k).real)

# Triangulation for matplotlib.
msh_qho.topology.create_connectivity(tdim, 0)
msh_qho.topology.create_connectivity(0, tdim)
cells_qho = msh_qho.topology.connectivity(tdim, 0).array.reshape(-1, 3)
nverts_qho = (
    msh_qho.topology.index_map(0).size_local
    + msh_qho.topology.index_map(0).num_ghosts
)
coords_qho = msh_qho.geometry.x[:nverts_qho, :2]
tri_qho = mtri.Triangulation(
    coords_qho[:, 0], coords_qho[:, 1], triangles=cells_qho
)
vids = np.arange(nverts_qho, dtype=np.int32)
dofs_v = fem.locate_dofs_topological(V_qho, 0, vids)

for k, (f, E) in enumerate(zip(psi_funcs, energies)):
    fig, ax = plt.subplots(figsize=(4.0, 3.6), dpi=140, constrained_layout=True)
    data = f.x.array[dofs_v].real
    vmax = np.max(np.abs(data))
    pcm = ax.tripcolor(tri_qho, data, shading="gouraud", cmap="RdBu_r",
                       vmin=-vmax, vmax=vmax)
    ax.set_title(rf"$\psi_{{{k}}}$,  $E\approx{E:.4f}$")
    ax.set_aspect("equal")
```

```
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    fig.colorbar(pcm, ax=ax, shrink=0.85)
    plt.show()
```

5.5 Remarks and extensions

- **Other potentials.** Replace `V_pot` to study e.g. a double well $V = \frac{1}{4}(x^2-1)^2 + \frac{1}{2}y^2$, a Coulomb-like $V = -1/\sqrt{x^2+y^2+\varepsilon^2}$, or a periodic potential.
- **Scaling.** SLEPc with shift-and-invert needs one LU factorization but many cheap back-solves; for very large 3D problems switch the inner KSP to an iterative solver with an algebraic multigrid preconditioner.
- **Parallelism.** Both PETSc and SLEPc are MPI-parallel out of the box — the same script runs under `mpirun -n P python ...` with no changes.
- **Linear elasticity vibrations.** The same machinery gives natural modes of an elastic body: $K\,u = \omega^2 M\,u$, with K the elasticity stiffness and M the (lumped or consistent) mass matrix.

6 Navier–Stokes (Non-Newtonian)

- TODO, see, Gemotion.jl
- Non-newtonian power-law fluid with viscosity $\mu(\dot\gamma) = \dot\gamma^{n-1}$, where $\dot\gamma = \sqrt{2\varepsilon(u) : \varepsilon(u)}$ with $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ is the strain rate tensor. The power-law index n controls the shear-thinning behavior: $n < 1$ is shear-thinning, $n = 1$ is Newtonian, and $n > 1$ is shear-thickening.

References

[1] I. A. Baratta *et al.*, "DOLFINx: The next generation FEniCS problem solving environment."

[2] F. Brezzi and M. Fortin, *Mixed and hybrid finite element methods.* in Springer series in computational mathematics, no. 15. New York: Springer, 1991. doi: 10.1007/978-1-4612-3172-1.

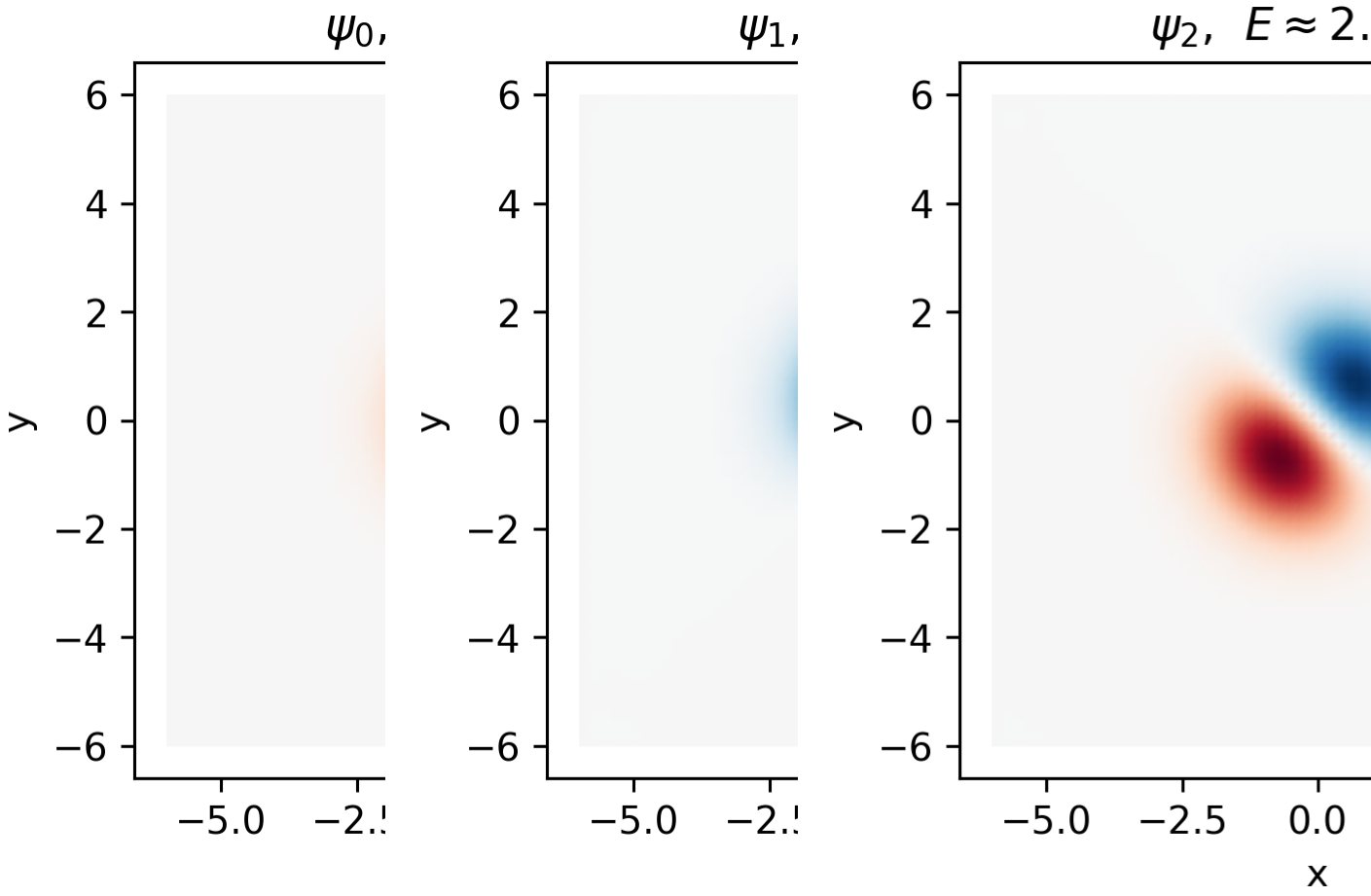
[3] P. Lewintan, L. Theisen, and M. Torrilhon, "Well-Posedness of the Linear Regularized 13-Moment Equations Using Tensor-Valued Korn Inequalities." arXiv, Apr. 2026. doi: 10.48550/arXiv.2501.14108.

[4] V. Girault and P.-A. Raviart, *Finite element methods for navier-stokes equations: Theory and algorithms.* in Springer series in computational mathematics, no. 5. Berlin: Springer, 1986. doi: 10.1007/978-3-642-61623-5.

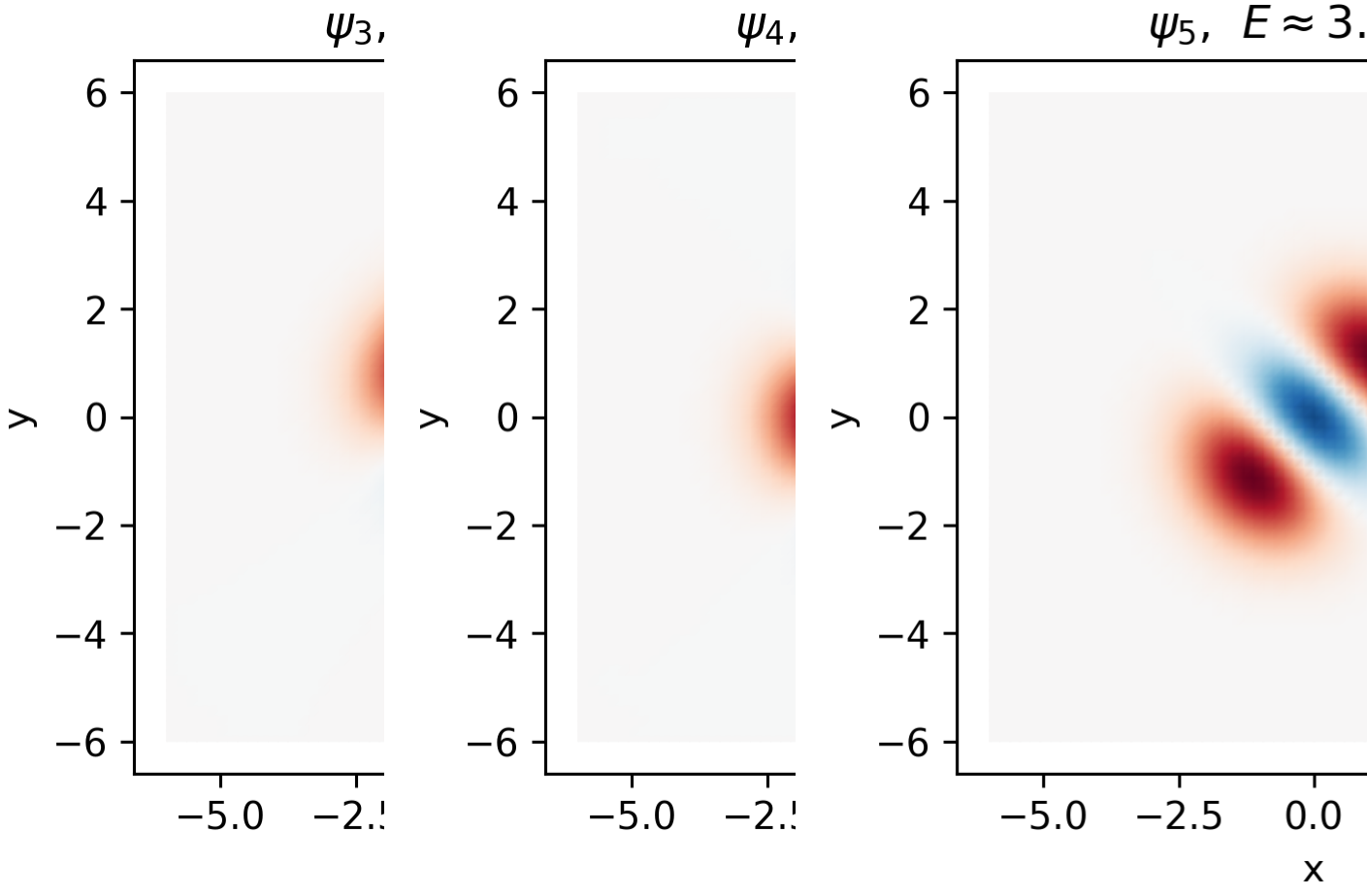
[5] L. Theisen and B. Stamm, "A Scalable Two-Level Domain Decomposition Eigensolver for Periodic Schrödinger Eigenstates in Anisotropically Expanding Domains," *SIAM J. Sci. Comput.*, pp. A3067–A3093, Oct. 2024, doi: 10.1137/23M161848X.

[6] B. Stamm and L. Theisen, "A Quasi-Optimal Factorization Preconditioner for Periodic Schrödinger Eigenstates in Anisotropically Expanding Domains," *SIAM J. Numer. Anal.*, vol. 60, no. 5, pp. 2508–2537, Oct. 2022, doi: 10.1137/21M1456005.

[7] D. J. Griffiths and D. F. Schroeter, *Introduction to Quantum Mechanics*, 3rd ed. Cambridge University Press, 2018. doi: 10.1017/9781316995433.

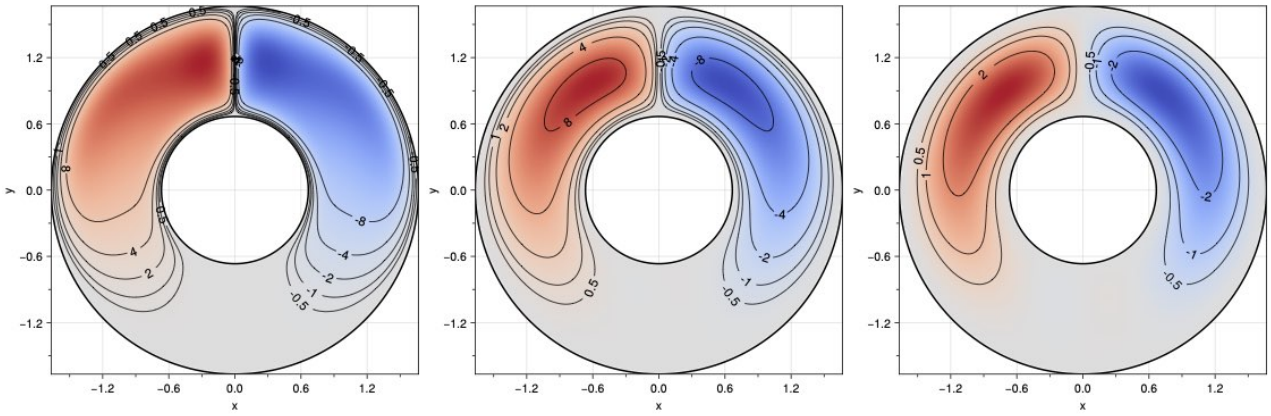


(a) n=0: E 1 (ground state) (b) n=1: E 2 (c) n=2: E 2

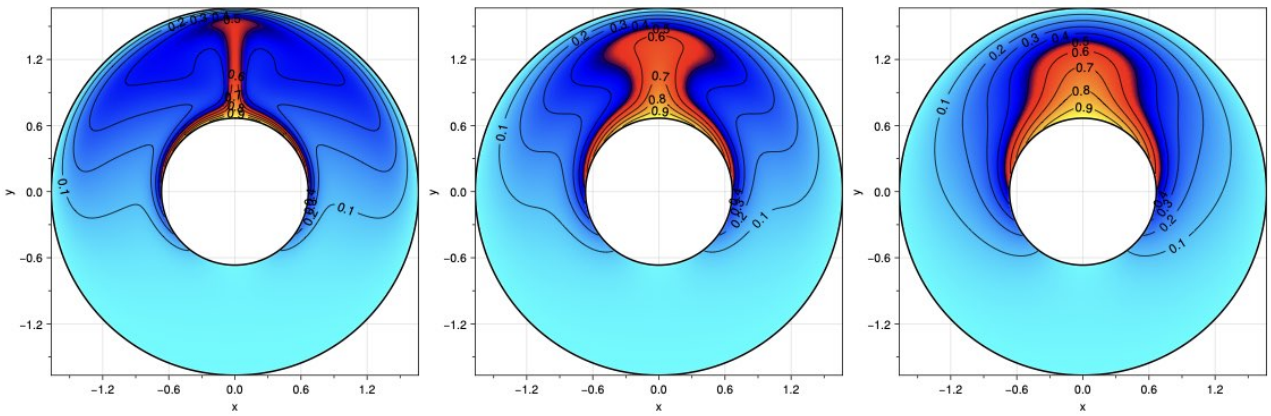


(d) n=3: E 3 (e) n=4: E 3 (f) n=5: E 3

Figure 12: Lowest six eigenfunctions of the 2D quantum harmonic oscillator computed with FEniCSx



(a) Stream function, $n = 0.6$ (a) Stream function, $n = 1.0$ (a) Stream function, $n = 1.4$



(a) Temperature, $n = 0.6$ (a) Temperature, $n = 1.0$ (a) Temperature, $n = 1.4$