Lecture 04 - FEM III

PETSc, preconditioning, TS, AMR, ...

Lambert Theisen

1 Objectives

Lecture 4 is a PETSc and preconditioning showcase. The numerical story is:

1. Start from a familiar elliptic FEM problem and see why the linear solver matters.
2. Use PETSc `KSP` and `PC` options to move from basic CG to multigrid-preconditioned CG.
3. Move to Stokes as a saddle-point problem and use PETSc field-split / Schur-complement preconditioning.
4. Optional: Revisit time-dependent problems, Boltzmann/BGK, and convection-dominated problems

```python
import inspect
import math

import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np
from mpi4py import MPI
from petsc4py import PETSc

import ufl
from basix.ufl import element, mixed_element
from dolfinx import fem, mesh
from dolfinx.fem.petsc import LinearProblem
```

2 PETSc: Portable, Extensible Toolkit for Scientific Computation

FEniCSx is excellent at describing and assembling finite element forms. PETSc is the layer that solves the resulting algebraic systems. The useful mental split is:

- UFL/FEniCSx: write the weak form and assemble matrices/vectors.
- PETSc `KSP`: choose the Krylov method (`cg`, `gmres`, `minres`, ...).
- PETSc `PC`: choose the preconditioner (`jacobi`, `lu`, `gamg`, `fieldsplit`, ...).
- PETSc options: change solver behavior without rewriting the weak form.

Preconditioning in a nutshell

After discretization, a PDE solve becomes a linear system

$$Ax = b.$$

A Krylov method such as CG or GMRES only uses matrix-vector products with A. If A is badly conditioned, those products point the iteration through a long, slow path to the solution. A preconditioner is an operator M^{-1} that is cheap to apply and approximates A^{-1} well enough that the transformed system is easier for the Krylov method:

$$M^{-1}Ax = M^{-1}b \qquad \text{or} \qquad AM^{-1}y = b, \quad x = M^{-1}y.$$

These formulas are the mathematical model. In an implementation, PETSc usually does **not** form the product $M^{-1}A$ or the inverse matrix M^{-1} explicitly. Each Krylov iteration applies A as a matrix-vector product and applies the preconditioner as an operation: given a residual-like vector r, approximately solve $Mz = r$, then use $z \approx M^{-1}r$ in the iteration.
The preconditioner should not be exact; if we could apply A^{-1} cheaply, the problem would already be solved. The point is to spend a little work per iteration on an approximate inverse, such as Jacobi, ILU, multigrid, or a block factorization, so that the outer iteration needs far fewer steps. In PETSc, `KSP` is the outer iteration and `PC` is this approximate inverse action.

We first solve a symmetric positive definite elliptic problem with CG and multigrid. Then we solve Stokes, where the matrix is indefinite and block-structured, so scalar multigrid alone is no longer the right abstraction.

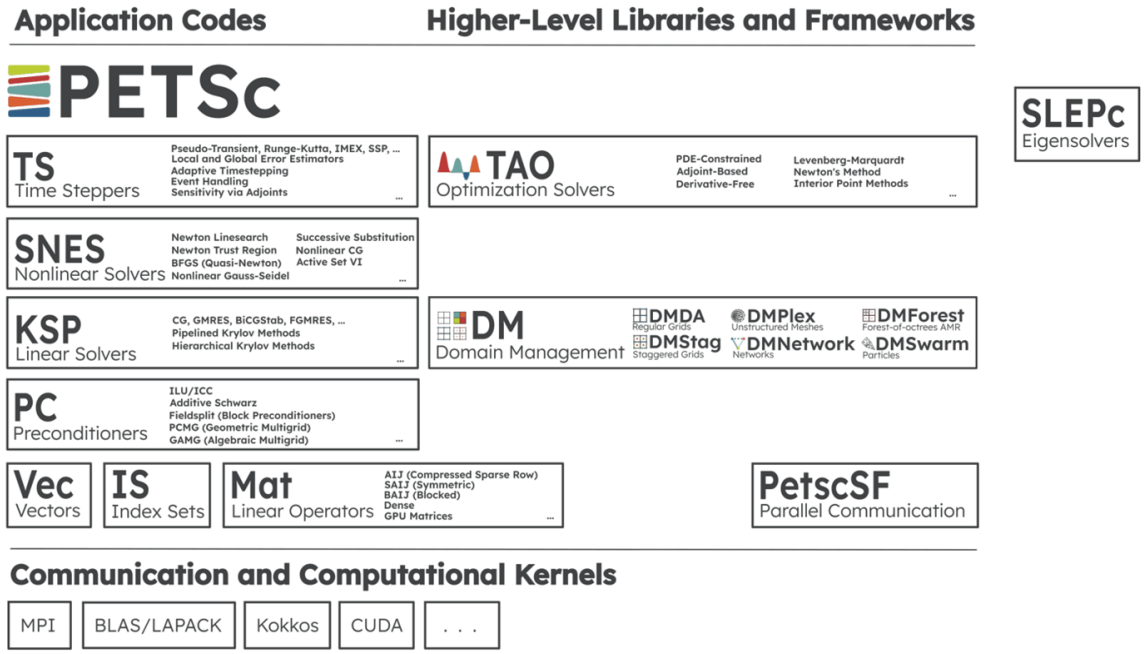


Figure 1: PETSc overview [1],[2]

PETSc habit

When a PDE solve is slow, do not first change the finite element form. First ask: what algebraic problem did I assemble, and does the used solver setup make sense for that algebraic problem?

PETSc Reference:

1. Solvers list 1, Solvers list 2, Solvers list 3
2. Preconditioners (partial list)

> **PETSc SNES**
>
> For nonlinear problems, PETSc has `SNES` (Scalable Nonlinear Equations Solvers) which is a similar abstraction to `KSP` but for nonlinear problems. We will not cover `SNES` in full detail but you can easily checkout its documentation and change, e.g., line search or trust region options in the examples of the previous lecture.

2.1 A tiny PETSc options dictionary

PETSc solvers are configured by options. In scripts these can come from the command line, for example

```
-ksp_type cg -pc_type gamg -ksp_rtol 1e-8
```

Inside a notebook it is often clearer to pass the same options as a Python dictionary. The keys are PETSc option names without the leading dash.

> **Important distinction**
>
> `ksp_type` decides the outer iteration. `pc_type` decides what approximation is used to make each iteration effective. The preconditioner is usually the more important choice.

```
elliptic_solver_cases = {
    "CG / no PC": {
        "ksp_type": "cg",
        "pc_type": "none",
        "ksp_rtol": "1.0e-8",
        "ksp_atol": "1.0e-12",
        "ksp_max_it": "2000",
    },
    "CG / Jacobi": {
        "ksp_type": "cg",
        "pc_type": "jacobi",
        "ksp_rtol": "1.0e-8",
        "ksp_atol": "1.0e-12",
        "ksp_max_it": "2000",
    },
    "CG / SOR": {
        "ksp_type": "cg",
        "pc_type": "sor",
        "ksp_rtol": "1.0e-8",
        "ksp_atol": "1.0e-12",
        "ksp_max_it": "2000",
    },
    "CG / GAMG": {
```

```
        "ksp_type": "cg",
        "pc_type": "gamg",
        "pc_gamg_type": "agg",
        "pc_gamg_threshold": "0.02",
        "ksp_rtol": "1.0e-8",
        "ksp_atol": "1.0e-12",
        "ksp_max_it": "2000",
    },
    "CG / hypre": {
        "ksp_type": "cg",
        "pc_type": "hypre",
        "ksp_rtol": "1.0e-8",
        "ksp_atol": "1.0e-12",
        "ksp_max_it": "2000",
    },
    "direct LU": {
        "ksp_type": "preonly",
        "pc_type": "lu",
    },
}
```

3 Elliptic problem (same FEM, different solver)

Consider a diffusion problem on the unit square:

$$-\nabla \cdot (k(x)\nabla u) = 1, \qquad u = 0 \text{ on } \partial\Omega.$$

The coefficient jumps at $x = 1/2$:

$$k(x) = \begin{cases} 1, & x < 1/2, \\ 100, & x \geq 1/2. \end{cases}$$

This is still the classic symmetric positive definite elliptic problem, so CG is the natural Krylov method. But the coefficient jump makes the conditioning worse. The lesson is not a new weak form; the lesson is that the preconditioner should know something about elliptic operators.

```
def boundary_facets(domain):
    fdim = domain.topology.dim - 1
    return mesh.locate_entities_boundary(
        domain, fdim, lambda X: np.full(X.shape[1], True)
    )


def solve_diffusion(nx, petsc_options, prefix):
    domain = mesh.create_unit_square(MPI.COMM_WORLD, nx, nx)
    V = fem.functionspace(domain, ("Lagrange", 1))

    u = ufl.TrialFunction(V)
```

```
    v = ufl.TestFunction(V)
    x = ufl.SpatialCoordinate(domain)

    kappa = 1.0 + 99.0 * ufl.conditional(ufl.gt(x[0], 0.5), 1.0, 0.0)
    a = kappa * ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
    L = fem.Constant(domain, PETSc.ScalarType(1.0)) * v * ufl.dx

    facets = boundary_facets(domain)
    dofs = fem.locate_dofs_topological(V, domain.topology.dim - 1, facets)
    bc = fem.dirichletbc(PETSc.ScalarType(0), dofs, V)

    problem = LinearProblem(
        a,
        L,
        bcs=[bc],
        petsc_options_prefix=prefix,
        petsc_options=petsc_options,
    )
    problem.solver.setConvergenceHistory()
    uh = problem.solve()

    ksp = problem.solver
    info = problem.A.getInfo()
    rows, cols = problem.A.getSize()
    return {
        "solution": uh,
        "iterations": ksp.getIterationNumber(),
        "reason": ksp.getConvergedReason(),
        "history": np.asarray(ksp.getConvergenceHistory(), dtype=float),
        "nnz": int(info.get("nz_used", 0)),
        "shape": (rows, cols),
    }
```

3.1 Compare Krylov methods, preconditioners, and a direct solve

All rows below use the same assembled finite element matrix. The only change is the PETSc solver configuration.

- `none`, `jacobi`, `sor`: useful baselines, but not scalable elliptic preconditioners.
- `gamg`, `hypre`: algebraic multigrid choices; these are the classic elliptic tools.
- `direct LU`: a robust reference solve, but factorization memory grows quickly in 2D and much worse in 3D.

> **PETSc options for direct LU**
>
> To use a direct LU factorization, set `-ksp_type preonly -pc_type lu`. The `preonly` KSP type tells PETSc to only apply the preconditioner, which in this case is a direct factorization. This is a common PETSc idiom for using a direct solver as a preconditioner.

> **Note**
>
> PETSc convergence reasons are listed here.

```
elliptic_results = {}
for label, options in elliptic_solver_cases.items():
    prefix = "diffusion_" + label.lower().replace(" / ", "_").replace(" ", "_") + "_"
    try:
        elliptic_results[label] = solve_diffusion(24, options, prefix)
    except Exception as err:
        elliptic_results[label] = {"error": str(err)}

for label, result in elliptic_results.items():
    if "error" in result:
        print(f"{label:14s} failed: {result['error']}")
        continue
    print(
        f"{label:14s} iterations={result['iterations']:4d} "
        f"reason={result['reason']:3d} nnz={result['nnz']:6d}"
    )
```

```
CG / no PC     iterations= 326 reason=  2 nnz=  4177
CG / Jacobi    iterations=  58 reason=  2 nnz=  4177
CG / SOR       iterations=  29 reason=  2 nnz=  4177
CG / GAMG      iterations=   8 reason=  2 nnz=  4177
CG / hypre     iterations=   5 reason=  2 nnz=  4177
direct LU      iterations=   1 reason=  4 nnz=  4177
```

3.2 PETSc `-log_view_memory`: setup memory is the point

`-log_view_memory` must be enabled when PETSc starts, so the compact measurement below runs two small solves in fresh Python processes:

- `CG / GAMG`, representing scalable elliptic preconditioning,
- `direct LU`, representing a robust direct reference solve.

The useful numbers are total PETSc-tracked runtime/memory plus the event times for `PCSetUp` and `KSPSolve`. For memory, the setup/factorization events matter most. This is closer to what we actually care about than raw assembled-matrix storage.

```
import os
import re
import subprocess
import sys


def run_diffusion_log_view(label, solver_options, nx=96):
    script = f"""
import sys, petsc4py
petsc4py.init(sys.argv)
```

```
from mpi4py import MPI
from petsc4py import PETSc
import numpy as np
import ufl
from dolfinx import fem, mesh
from dolfinx.fem.petsc import LinearProblem

domain = mesh.create_unit_square(MPI.COMM_WORLD, {nx}, {nx})
V = fem.functionspace(domain, ("Lagrange", 1))
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
x = ufl.SpatialCoordinate(domain)
kappa = 1.0 + 99.0 * ufl.conditional(ufl.gt(x[0], 0.5), 1.0, 0.0)
a = kappa * ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
L = fem.Constant(domain, PETSc.ScalarType(1.0)) * v * ufl.dx
facets = mesh.locate_entities_boundary(
    domain, domain.topology.dim - 1, lambda X: np.full(X.shape[1], True)
)
dofs = fem.locate_dofs_topological(V, domain.topology.dim - 1, facets)
bc = fem.dirichletbc(PETSc.ScalarType(0), dofs, V)
problem = LinearProblem(
    a,
    L,
    bcs=[bc],
    petsc_options_prefix="logview_diffusion_",
    petsc_options={solver_options!r},
)
problem.solve()
print("SOLVER_RESULT", problem.solver.getIterationNumber(), problem.solver.getConvergedReason(
"""
    run = subprocess.run(
        [sys.executable, "-", "-malloc_log", "-log_view", "-log_view_memory"],
        input=script,
        text=True,
        capture_output=True,
        env={**os.environ, "XDG_CACHE_HOME": os.environ.get("XDG_CACHE_HOME", "/tmp")},
    )
    if run.returncode != 0:
        raise RuntimeError(run.stderr)

    output = run.stdout + "\n" + run.stderr
    memory_match = re.search(r"Memory \(bytes\):\s+([0-9.eE+-]+)", output)
    time_match = re.search(r"Time \(sec\):\s+([0-9.eE+-]+)", output)
    result_match = re.search(r"SOLVER_RESULT\s+(\d+)\s+(-?\d+)", output)

    def event_columns(event_name):
        for line in output.splitlines():
            if line.lstrip().startswith(event_name):
                fields = line.split()
                # PETSc event table columns start with: Event Count Max Ratio TimeMax TimeRati
                # Last columns are: Total Mflop/s, Malloc, EMalloc, MMalloc, RMI.
```

```
                return {
                    "time_sec": float(fields[3]),
                    "memory_mb": tuple(float(x) for x in fields[-4:-1]),
                }
        return {"time_sec": 0.0, "memory_mb": (0.0, 0.0, 0.0)}

    return {
        "label": label,
        "iterations": int(result_match.group(1)) if result_match else None,
        "reason": int(result_match.group(2)) if result_match else None,
        "total_time_sec": float(time_match.group(1)) if time_match else float("nan"),
        "total_memory_mb": float(memory_match.group(1)) / 1024**2 if memory_match else float("
        "pcsetup": event_columns("PCSetUp"),
        "kspsolve": event_columns("KSPSolve"),
        "lusym": event_columns("MatLUFactorSym"),
        "lunum": event_columns("MatLUFactorNum"),
    }


logview_cases = [
    run_diffusion_log_view(
        "CG / GAMG",
        {"ksp_type": "cg", "pc_type": "gamg", "ksp_rtol": "1.0e-8"},
    ),
    run_diffusion_log_view(
        "direct LU",
        {"ksp_type": "preonly", "pc_type": "lu"},
    ),
]

print(
    "case        iterations  total time  PCSetUp time  KSPSolve time  "
    "total PETSc memory  PCSetUp Malloc/EMalloc/MMalloc  LU symbolic Malloc/EMalloc/MMalloc"
)
for row in logview_cases:
    print(
        f"{row['label']:10s} {row['iterations']:10d} "
        f"{row['total_time_sec']:9.3e}s  "
        f"{row['pcsetup']['time_sec']:9.3e}s  "
        f"{row['kspsolve']['time_sec']:9.3e}s  "
        f"{row['total_memory_mb']:9.2f} MiB        "
        f"{row['pcsetup']['memory_mb']}        {row['lusym']['memory_mb']}"
    )
```

```
case        iterations  total time  PCSetUp time  KSPSolve time  total PETSc memory  PCSetUp M
CG / GAMG           13 2.039e-01s  7.499e-03s  3.795e-03s       4.57 MiB        (1.0, 2.0, 3.0)
direct LU            1 1.629e-01s  1.135e-02s  4.060e-04s      10.25 MiB        (7.0, 3.0, 9.0)
```

3.3 Convergence history

The residual history makes the preconditioner visible. Good preconditioners do not merely reduce the final iteration count; they change the residual curve from a slow crawl into a steep drop.

```
fig, ax = plt.subplots(figsize=(6.5, 4.0))
for label, result in elliptic_results.items():
    history = result.get("history")
    if history is None or len(history) == 0:
        continue
    ax.semilogy(history / history[0], marker="o", markersize=3, label=label)
ax.set_xlabel("KSP iteration")
ax.set_ylabel("relative residual norm")
ax.set_title("Same FEM matrix, different PETSc solvers")
ax.grid(True, which="both", alpha=0.3)
ax.legend()
plt.show()
```

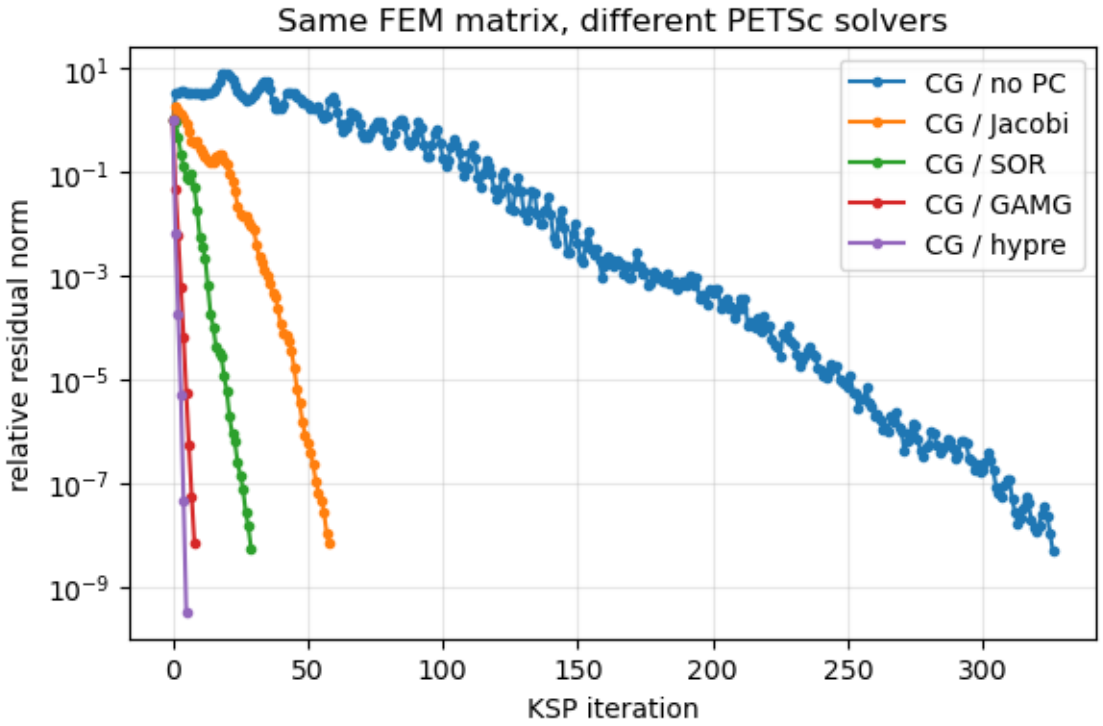


Figure 2: PETSc residual histories for the high-contrast elliptic solve.

```
preferred = "CG / GAMG" if "CG / GAMG" in elliptic_results else next(iter(elliptic_results))
uh = elliptic_results[preferred]["solution"]
domain = uh.function_space.mesh

def triangulation(domain):
    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]
    return coords, mtri.Triangulation(coords[:, 0], coords[:, 1], cells)

coords, tri = triangulation(domain)
fig, ax = plt.subplots(figsize=(5.5, 4.2))
plot = ax.tricontourf(tri, uh.x.array[: coords.shape[0]], levels=30)
fig.colorbar(plot, ax=ax, label="u")
ax.axvline(0.5, color="white", linewidth=1.5, linestyle="--")
ax.set_aspect("equal")
ax.set_title("High-contrast diffusion")
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.show()
```

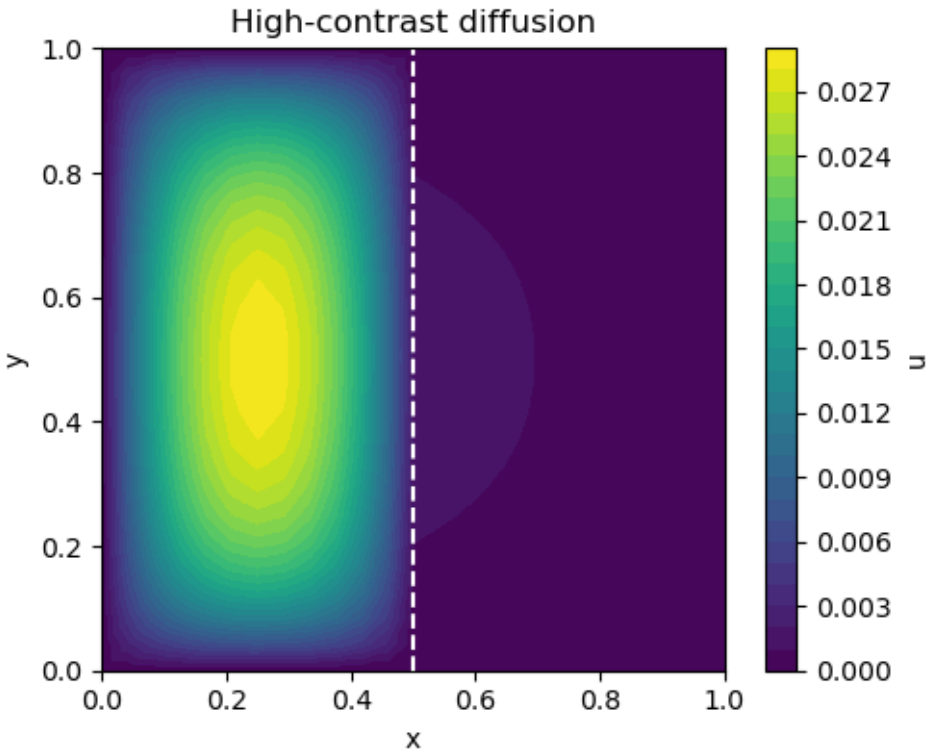


Figure 3: Diffusion solution computed with CG and PETSc GAMG preconditioning.

3.4 Mesh convergence study: Robustness of preconditioner

We use a *mesh convergence study*: solve the same problem on a sequence of refined meshes and watch whether the outputs of interest stop changing.

For the high-contrast elliptic problem above, we monitor three things:

1. degrees of freedom and solver iterations,
2. scalar quantities of interest such as the mean value and maximum value of u_h,

3. a centerline profile $u_h(x, 1/2)$ compared against the finest available mesh.

The finest mesh is only a numerical reference, not truth. If the conclusion changes when the reference mesh is refined again, the study was not fine enough yet.

```python
from dolfinx import geometry


def solve_diffusion_for_mesh_study(nx):
    domain = mesh.create_unit_square(MPI.COMM_WORLD, nx, nx)
    V = fem.functionspace(domain, ("Lagrange", 1))

    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)
    x = ufl.SpatialCoordinate(domain)

    kappa = 1.0 + 99.0 * ufl.conditional(ufl.gt(x[0], 0.5), 1.0, 0.0)
    a = kappa * ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
    L = fem.Constant(domain, PETSc.ScalarType(1.0)) * v * ufl.dx

    facets = boundary_facets(domain)
    dofs = fem.locate_dofs_topological(V, domain.topology.dim - 1, facets)
    bc = fem.dirichletbc(PETSc.ScalarType(0), dofs, V)

    linear_problem_kwargs = {
        "bcs": [bc],
        "petsc_options": {
            "ksp_type": "cg",
            "pc_type": "gamg",
            "pc_gamg_type": "agg",
            "pc_gamg_threshold": "0.02",
            "ksp_rtol": "1.0e-10",
            "ksp_atol": "1.0e-12",
            "ksp_max_it": "1000",
        },
    }
    if "petsc_options_prefix" in inspect.signature(LinearProblem).parameters:
        linear_problem_kwargs["petsc_options_prefix"] = f"mesh_study_{nx}_"

    problem = LinearProblem(a, L, **linear_problem_kwargs)
    uh = problem.solve()

    mean_local = fem.assemble_scalar(fem.form(uh * ufl.dx))
    mean_u = domain.comm.allreduce(mean_local, op=MPI.SUM)
    max_local = float(np.max(uh.x.array.real)) if uh.x.array.size else -np.inf
    max_u = domain.comm.allreduce(max_local, op=MPI.MAX)
    ndofs = V.dofmap.index_map.size_global * V.dofmap.index_map_bs

    return {
        "nx": nx,
        "h": 1.0 / nx,
        "ndofs": ndofs,
```

```
        "iterations": problem.solver.getIterationNumber(),
        "solution": uh,
        "mean_u": float(np.real(mean_u)),
        "max_u": max_u,
    }


def sample_on_centerline(uh, npoints=241):
    domain = uh.function_space.mesh
    xline = np.linspace(0.02, 0.98, npoints)
    points = np.column_stack([xline, np.full_like(xline, 0.5), np.zeros_like(xline)])

    tree = geometry.bb_tree(domain, domain.topology.dim)
    candidates = geometry.compute_collisions_points(tree, points)
    colliding_cells = geometry.compute_colliding_cells(domain, candidates, points)

    cells = []
    valid = []
    for i in range(points.shape[0]):
        links = colliding_cells.links(i)
        if len(links) > 0:
            valid.append(i)
            cells.append(links[0])

    values = np.full(points.shape[0], np.nan)
    if valid:
        values[np.array(valid)] = uh.eval(
            points[np.array(valid)], np.array(cells, dtype=np.int32)
        ).reshape(-1).real
    return xline, values


mesh_study_n_values = [8, 16, 32, 64, 128, 256]
mesh_study = [solve_diffusion_for_mesh_study(nx) for nx in mesh_study_n_values]

for row in mesh_study:
    row["line_x"], row["line_u"] = sample_on_centerline(row["solution"])

reference_line = mesh_study[-1]["line_u"]
for row in mesh_study[:-1]:
    diff = row["line_u"] - reference_line
    row["line_diff_to_reference"] = float(np.sqrt(np.nanmean(diff**2)))
mesh_study[-1]["line_diff_to_reference"] = 0.0

mean_values = np.array([row["mean_u"] for row in mesh_study])
max_values = np.array([row["max_u"] for row in mesh_study])
mean_changes = np.abs(np.diff(mean_values))
max_changes = np.abs(np.diff(max_values))
line_diffs = np.array([row["line_diff_to_reference"] for row in mesh_study[:-1]])
line_diff_orders = np.log(line_diffs[:-1] / line_diffs[1:]) / np.log(2.0)

```

```python
if MPI.COMM_WORLD.rank == 0:
    print("mesh   dofs    KSP it   mean(u)      change       max(u)       change       line di
    for i, row in enumerate(mesh_study):
        mean_change = "---" if i == 0 else f"{mean_changes[i - 1]:.3e}"
        max_change = "---" if i == 0 else f"{max_changes[i - 1]:.3e}"
        line_diff = "---" if i == len(mesh_study) - 1 else f"{row['line_diff_to_reference']:.3
        print(
            f"{row['nx']:4d} {row['ndofs']:7d} {row['iterations']:7d} "
            f"{row['mean_u']:.8e} {mean_change:>11s} "
            f"{row['max_u']:.8e} {max_change:>11s} {line_diff:>18s}"
        )
```

```
mesh   dofs    KSP it   mean(u)      change       max(u)       change       line diff to fines
   8      81       9 6.84850898e-03          --- 2.85931442e-02          ---          1.131e-
03
  16     289      11 7.42622245e-03   5.777e-04 2.89269051e-02   3.338e-04          2.829e-
04
  32    1089      11 7.57884274e-03   1.526e-04 2.90113550e-02   8.445e-05          6.948e-
05
  64    4225      13 7.61765824e-03   3.882e-05 2.90325210e-02   2.117e-05          1.670e-
05
 128   16641      13 7.62741257e-03   9.754e-06 2.90378156e-02   5.295e-06          3.425e-
06
 256   66049      14 7.62985488e-03   2.442e-06 2.90429344e-02   5.119e-06                 ---
```

We see that for CG+GAMG, the solver iteration count is quite stable across meshes in contrast to the case of CG+(noPC):

```
mesh   dofs    KSP it   mean(u)      change       max(u)       change       line diff to fines
   8      81      57 6.84850898e-03          --- 2.85931442e-02          ---          7.336e-
04
  16     289     196 7.42622245e-03   5.777e-04 2.89269051e-02   3.338e-04          1.290e-
03
  32    1089     539 7.57884274e-03   1.526e-04 2.90113550e-02   8.445e-05          1.472e-
03
  64    4225    1000 7.61765824e-03   3.882e-05 2.90325212e-02   2.117e-05          1.519e-
03
 128   16641    1000 7.62737317e-03   9.715e-06 2.90247447e-02   7.776e-06          1.527e-
03
 256   66049    1000 7.52459773e-03   1.028e-04 2.66045939e-02   2.420e-03                 ---
```

```python
if MPI.COMM_WORLD.rank == 0:
    hs = np.array([row["h"] for row in mesh_study])
    fig, axs = plt.subplots(1, 2, figsize=(9.4, 3.8), constrained_layout=True)

    axs[0].plot(hs, mean_values, "o-", linewidth=2, label=r"$\int_\Omega u_h\,dx$")
    axs[0].plot(hs, max_values, "s-", linewidth=2, label=r"$\max u_h$")
    axs[0].invert_xaxis()
    axs[0].set_xlabel("mesh size h")
    axs[0].set_ylabel("quantity of interest")
```

```
    axs[0].set_title("QoIs stabilize")
    axs[0].grid(True, alpha=0.3)
    axs[0].legend()

    axs[1].loglog(hs[1:], mean_changes, "o-", linewidth=2, label="mean change")
    axs[1].loglog(hs[1:], max_changes, "s-", linewidth=2, label="max change")
    axs[1].invert_xaxis()
    axs[1].set_xlabel("finer mesh size h")
    axs[1].set_ylabel(r"$|Q_h-Q_{2h}|$")
    axs[1].set_title("successive differences")
    axs[1].grid(True, which="both", alpha=0.3)
    axs[1].legend()
    plt.show()
```

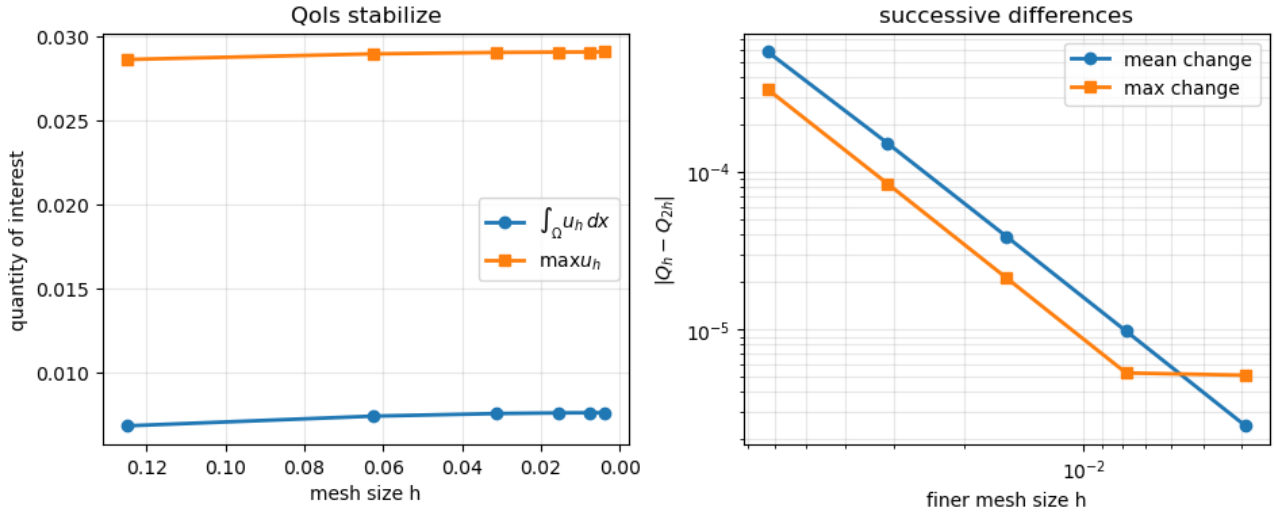


Figure 4: Scalar quantities of interest stabilize under refinement, and successive changes estimate the remaining discretization effect.

```
if MPI.COMM_WORLD.rank == 0:
    fig, axs = plt.subplots(1, 2, figsize=(10.4, 3.9), constrained_layout=True)

    for row in mesh_study:
        axs[0].plot(
            row["line_x"],
            row["line_u"],
            linewidth=1.8,
            label=rf"$h=1/{row['nx']}$",
        )
    axs[0].axvline(0.5, color="black", linestyle=":", linewidth=1.0)
    axs[0].set_xlabel(r"$x$ on the line $y=1/2$")
    axs[0].set_ylabel(r"$u_h(x, 1/2)$")
    axs[0].set_title("solution on the centerline")
    axs[0].grid(True, alpha=0.3)
    axs[0].legend()

    profile_h = np.array([row["h"] for row in mesh_study[:-1]])
```

```
    axs[1].loglog(profile_h, line_diffs, "o-", linewidth=2, label="difference to finest")
    for i, order in enumerate(line_diff_orders, start=1):
        axs[1].annotate(f"{order:.2f}", (profile_h[i], line_diffs[i]), textcoords="offset poin
    axs[1].invert_xaxis()
    axs[1].set_xlabel("mesh size h")
    axs[1].set_ylabel("RMS profile difference")
    axs[1].set_title("profile difference to finest mesh")
    axs[1].grid(True, which="both", alpha=0.3)
    axs[1].legend()
    plt.show()
```

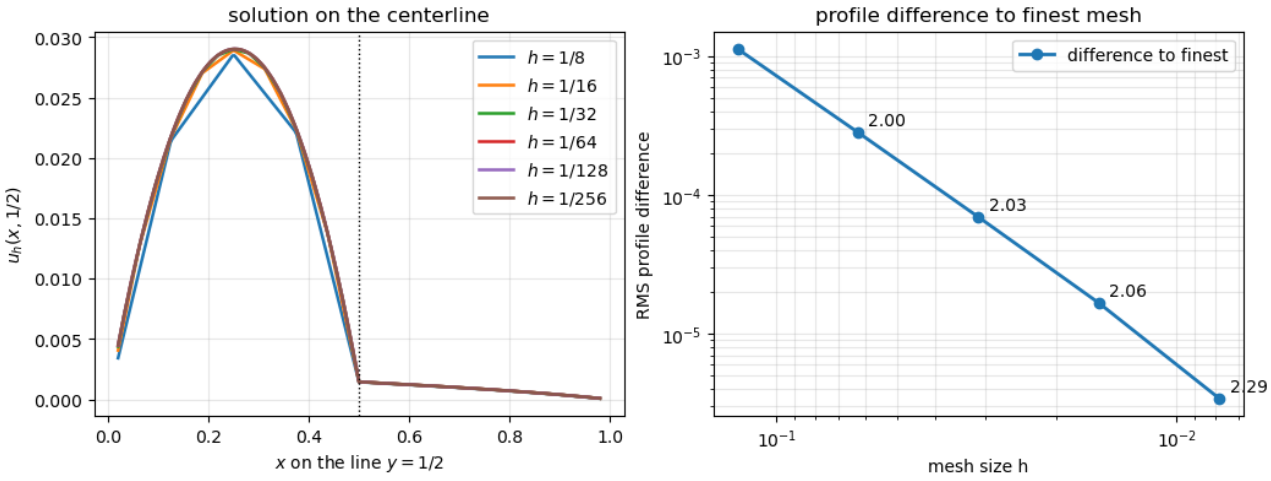


Figure 5: Solution restricted to the centerline $y = 1/2$ for different mesh resolutions. The right panel compares each centerline profile against the finest available mesh, which is only a numerical reference.

```
if MPI.COMM_WORLD.rank == 0:
    field_rows = [mesh_study[0], mesh_study[-1]]
    fig, axs = plt.subplots(1, 2, figsize=(8.6, 3.8), constrained_layout=True)

    for ax, row in zip(axs, field_rows):
        domain = row["solution"].function_space.mesh
        coords, tri = triangulation(domain)
        values = row["solution"].x.array[: coords.shape[0]].real
        plot = ax.tricontourf(tri, values, levels=30)
        fig.colorbar(plot, ax=ax, label="u")
        ax.axvline(0.5, color="white", linewidth=1.2, linestyle="--", label="coefficient jump"
        ax.axhline(0.5, color="black", linewidth=1.2, linestyle=":", label="centerline")
        ax.set_aspect("equal")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.set_title(f"{row['nx']} x {row['nx']} mesh")
    plt.show()
```

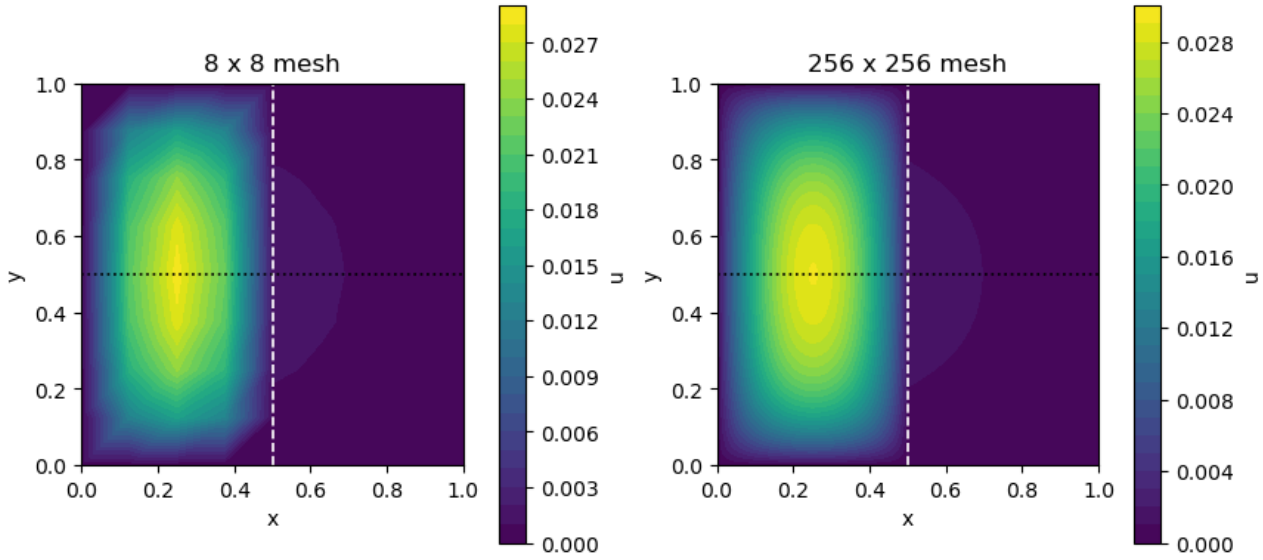


Figure 6: Coarse and finest mesh solutions for the same high-contrast elliptic problem. The white vertical line marks the coefficient jump, and the black horizontal line marks the sampled centerline $y = 1/2$.

> **Reading the PETSc reason code**
>
> A positive convergence reason means success. For example, 2 means the relative tolerance was reached. A negative reason means divergence or breakdown.

3.5 Random high-contrast media

The previous coefficient jump was deliberately simple. Real elliptic problems often have rough coefficients: porous media, composites, fractures, inclusions. Here we make a synthetic high-contrast coefficient by assigning random values cell-by-cell and smoothing nothing. The solver story is the same, but the picture is more representative of heterogeneous media.

```
rng = np.random.default_rng(12)
random_domain = mesh.create_unit_square(MPI.COMM_WORLD, 48, 48)
V_random = fem.functionspace(random_domain, ("Lagrange", 1))
DG0_random = fem.functionspace(random_domain, ("DG", 0))

cell_values = rng.lognormal(mean=0.0, sigma=1.4, size=DG0_random.dofmap.index_map.size_local)
cell_values = np.clip(cell_values, 0.05, 50.0)
kappa_random = fem.Function(DG0_random, name="kappa")
kappa_random.x.array[: cell_values.size] = cell_values
kappa_random.x.scatter_forward()

u = ufl.TrialFunction(V_random)
v = ufl.TestFunction(V_random)
a_random = kappa_random * ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
L_random = fem.Constant(random_domain, PETSc.ScalarType(1.0)) * v * ufl.dx
facets = boundary_facets(random_domain)
dofs = fem.locate_dofs_topological(V_random, random_domain.topology.dim - 1, facets)
```

```
bc = fem.dirichletbc(PETSc.ScalarType(0), dofs, V_random)

random_problem = LinearProblem(
    a_random,
    L_random,
    bcs=[bc],
    petsc_options_prefix="random_diffusion_",
    petsc_options={
        "ksp_type": "cg",
        "pc_type": "gamg",
        "ksp_rtol": "1.0e-8",
        "ksp_max_it": "1000",
    },
)
random_problem.solver.setConvergenceHistory()
u_random = random_problem.solve()
print("Random coefficient CG/GAMG iterations:", random_problem.solver.getIterationNumber())

coords, tri = triangulation(random_domain)
cell_kappa = kappa_random.x.array[: tri.triangles.shape[0]]
fig, axs = plt.subplots(1, 2, figsize=(9.0, 3.8), constrained_layout=True)

kplot = axs[0].tripcolor(tri, facecolors=cell_kappa, shading="flat")
fig.colorbar(kplot, ax=axs[0], label="kappa")
axs[0].set_title("random coefficient")
axs[0].set_aspect("equal")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")

uplot = axs[1].tricontourf(tri, u_random.x.array[: coords.shape[0]], levels=30)
fig.colorbar(uplot, ax=axs[1], label="u")
axs[1].set_title("solution")
axs[1].set_aspect("equal")
axs[1].set_xlabel("x")
axs[1].set_ylabel("y")
plt.show()
```

```
Random coefficient CG/GAMG iterations: 16
```

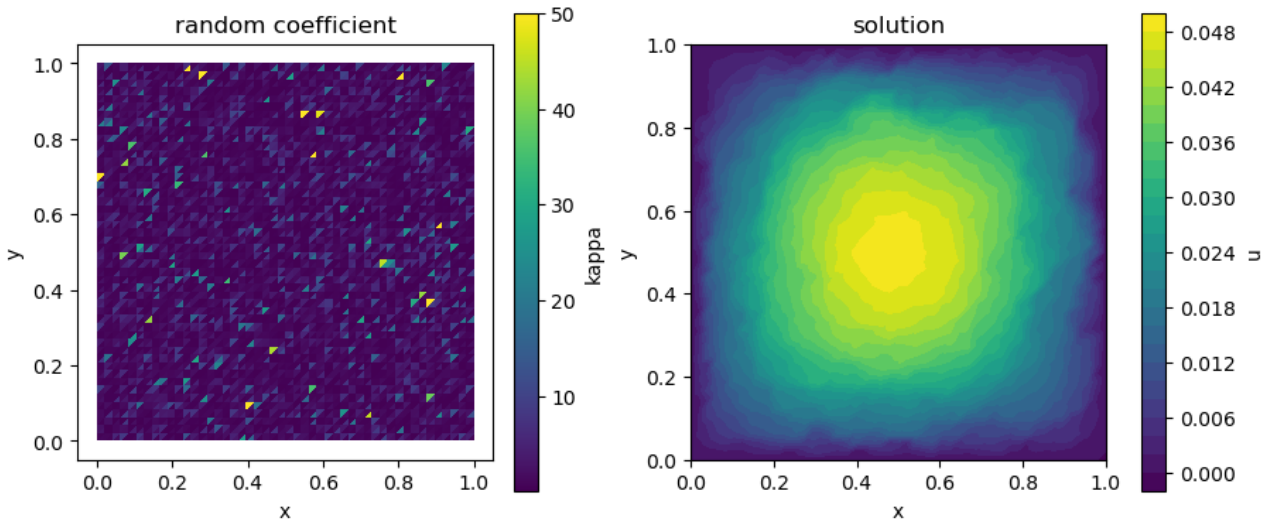


Figure 7: Random high-contrast diffusion coefficient and resulting elliptic solution.

4 Saddle-point systems and Stokes-style preconditioning

For incompressible flow, Stokes has the block form

$$\begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} U \\ P \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix},$$

where A is a vector Laplacian-like velocity block and the zero pressure block encodes the incompressibility constraint. This is the same algebraic difficulty highlighted in this Firedrake saddle-point demo: block systems are not preconditioned well by treating the whole matrix as an unstructured sparse matrix.

We start with the mixed Poisson formulation from the above mentioned demo with flux σ and scalar u. It is not Stokes physically, but it has the same saddle-point matrix pattern. That makes it the cleanest place to understand Schur complement preconditioning.

4.1 What Schur preconditioning is actually doing

A saddle-point system has the block form

$$\begin{bmatrix} A & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}.$$

Think of x as the easy field and y as the constraint field. In Stokes, x is velocity and y is pressure. The first block row says

$$Ax + B^T y = f.$$

If we could invert A, then

$$x = A^{-1}(f - B^T y).$$

Insert this into the second block row $Bx = g$:

$$BA^{-1}(f - B^T y) = g.$$

Rearranging gives a reduced equation for the constraint variable only:

$$\underbrace{(-BA^{-1}B^T)}_{S}\, y = g - BA^{-1} f.$$

The matrix $S = -BA^{-1}B^T$ is the **Schur complement**. It tells us how the constraint variable reacts after the easy field has been eliminated.

For Stokes this means: pressure is not solved by the pressure block, because the pressure block is zero. Pressure is solved through the velocity equations and the divergence constraint. That hidden pressure operator is the Schur complement.

The preconditioning idea

The exact Schur complement

$$S = -BA^{-1}B^T$$

is almost never assembled in production. Applying it exactly would require a solve with A whenever the pressure/constraint block is touched, and assembling it would usually destroy sparsity. `PCFIELDSPLIT` therefore does not try to invert the coupled matrix as one unstructured sparse object. It applies an approximate block factorization.
For

$$J = \begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix}, \qquad S = -BA^{-1}B^T,$$

the exact block factorization (like $A = LDL^\top$) is

$$J = \begin{pmatrix} I & 0 \\ BA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B^T \\ 0 & I \end{pmatrix}.$$

PETSc replaces A^{-1} and S^{-1} by approximate actions. Write these approximate inverse actions as $K_A \approx A^{-1}$ and $K_S \approx S^{-1}$. The option `pc_fieldsplit_schur_fact_type` chooses which part of the factorization is used:

- `diag`: block diagonal action only,

$$z_x = K_A r_x, \qquad z_y = K_S r_y, \qquad P^{-1}_{\text{diag}} \approx \begin{pmatrix} K_A & 0 \\ 0 & K_S \end{pmatrix}.$$

- `lower`: lower triangular correction, so the constraint residual first sees the effect of the first-field correction,

$$z_x = K_A r_x, \qquad z_y = K_S(r_y - B z_x), \qquad P^{-1}_{\text{lower}} \approx \begin{pmatrix} K_A & 0 \\ -K_S B K_A & K_S \end{pmatrix}.$$

- `upper`: upper triangular correction, so the first-field residual is corrected by the Schur update,

$$z_y = K_S r_y, \qquad z_x = K_A(r_x - B^T z_y), \qquad P_{\text{upper}}^{-1} \approx \begin{pmatrix} K_A & -K_A B^T K_S \\ 0 & K_S \end{pmatrix}.$$

- `full`: applies both triangular corrections,

$$y_x = K_A r_x, \qquad y_y = K_S(r_y - B y_x), \qquad z_y = y_y, \qquad z_x = y_x - K_A B^T y_y.$$

If $K_A = A^{-1}$ and $K_S = S^{-1}$ were exact, `full` would reproduce the exact block inverse. In practice both are approximate, so PETSc uses this approximate factorization inverse as the preconditioner applied by the outer Krylov method. It is still relatively expensive because one application may contain several block solves and coupling-block multiplications, but it is usually much cheaper than a direct solve of the full coupled matrix and much more effective than a generic sparse preconditioner.
The PETSc prefixes name the two approximate inverse actions. `fieldsplit_0_*` configures K_A, the solver/preconditioner for the first block A. `fieldsplit_1_*` configures the Schur part. The option `pc_fieldsplit_schur_precondition` chooses the matrix used to precondition that Schur solve: for example `a11` uses the $(1,1)$ block, `selfp` builds a sparse approximation such as $A_{11} - A_{10}\,\mathrm{diag}(A_{00})^{-1}A_{01}$, and `user` means that the application supplies a problem-specific Schur preconditioner.
Why is this extra choice needed? Because using the Schur complement as an operator is not the same as having a cheap inverse for it. The action of $S = -BA^{-1}B^T$ already contains an A-solve. If we used the exact S itself as the preconditioner, then PETSc would still need a way to apply something like S^{-1}, which means either assembling and factorizing a dense or expensive matrix, or running another nested Krylov solve whose matvecs again require A-solves. That can cost almost as much as solving the original coupled problem. A Schur preconditioner P_S is therefore a cheaper matrix whose inverse is easy enough to apply and spectrally close enough to S^{-1} to help the outer Krylov method.

PETSc field split options

In PETSc terms, a Schur field split separates three choices:

1. The outer Krylov method for the whole saddle-point system, usually `ksp_type gmres` or `minres` depending on symmetry.
2. The block factorization shape, set by `pc_fieldsplit_schur_fact_type`. This decides which approximate inverse factorization PETSc applies as the outer preconditioner. `upper` and `lower` mimic triangular factorizations; `full` mimics the full block factorization and applies more block operations per preconditioner call.
3. The two split solvers, configured as `fieldsplit_0_*` and `fieldsplit_1_*`.

The first split is the A block. `fieldsplit_0_ksp_type` and `fieldsplit_0_pc_type` define an approximate action of A^{-1}. What is reasonable depends on A: Jacobi can be enough for a mass matrix, ILU may be a small-mesh baseline, and multigrid is the usual scalable choice for elliptic velocity blocks.
The second split is the Schur part. `fieldsplit_1_*` does **not** solve the original zero pressure block; that block has no inverse. It controls the inner Schur correction solve. Conceptually, that correction is

$$Sz = r_y,$$

inside each application of the block preconditioner. Here r_y is the current residual component in the constraint field. PETSc can represent the Schur operator S implicitly, but an inner Krylov method for this equation still needs its own preconditioner. That is what `pc_fieldsplit_schur_precondition` supplies: a cheaper Schur-preconditioning matrix P_S used by the `fieldsplit_1_*` solver.
The distinction is important:

- $S = -BA^{-1}B^T$ is the mathematically correct reduced operator, but applying it already requires an approximate solve with A.
- P_S is the matrix whose inverse the Schur split can apply cheaply, for example by Jacobi, ILU, AMG, or a problem-specific pressure solver.
- Choosing $P_S = S$ exactly would only be useful if S^{-1} were also cheap to apply. Usually it is not: assembling S can destroy sparsity, and factorizing it or solving with it accurately gives another expensive nested solve.

So `pc_fieldsplit_schur_precondition` is not a redundant copy of the Schur complement. It tells PETSc what practical matrix should stand behind the PC used for the Schur solve. If the exact Schur complement were affordable, then one could choose $P_S = S$. In practice, P_S is cheaper. For example:

- `selfp` asks PETSc to build a sparse approximation from the available matrix blocks instead of forming $-BA^{-1}B^T$ exactly.
- Other setups can supply a user-chosen pressure/constraint operator, such as a pressure mass matrix, pressure Laplacian, or pressure convection-diffusion operator.

So the mental model is: `fieldsplit_0_*` approximates solves of the form $Aw = r_x$, while `fieldsplit_1_*` approximately applies the Schur correction, preconditioned by P_S. The goal is not to reproduce S entry by entry. The goal is to supply an operator whose inverse has similar spectral behavior to S^{-1}, so GMRES sees a clustered, better-conditioned preconditioned system.

For the mixed Poisson example below, $A = M$ is a flux mass matrix, so the first split can be cheap. The Schur complement $S = -BM^{-1}B^T$ behaves like a scalar Laplacian on the u field. This is why the Schur split is the right abstraction: approximate the mass-block inverse in split 0 and use a Laplacian-like sparse operator or preconditioner in split 1.

For Stokes, A is a vector Laplacian-like velocity operator, and $S = -BA^{-1}B^T$ is the hidden pressure operator produced by eliminating velocity. Common approximations include pressure mass matrices for constant-viscosity Stokes, and pressure convection-diffusion or least-squares commutator approximations for harder Navier–Stokes variants. The correct choice depends on viscosity, boundary conditions, stabilization, and the exact PDE being solved.

```
mixed_poisson_cases = {
    "GMRES / ILU": {
        "ksp_type": "gmres",
        "ksp_gmres_restart": "100",
        "ksp_rtol": "1.0e-8",
        "pc_type": "ilu",
    },
    "Schur / selfp": {
        "ksp_type": "gmres",
        "ksp_rtol": "1.0e-8",
        "pc_type": "fieldsplit",
```

```
        "pc_fieldsplit_detect_saddle_point": None,
        "pc_fieldsplit_type": "schur",
        "pc_fieldsplit_schur_fact_type": "full",
        "pc_fieldsplit_schur_precondition": "selfp",
        "fieldsplit_0_ksp_type": "preonly",
        "fieldsplit_0_pc_type": "jacobi",
        "fieldsplit_1_ksp_type": "preonly",
        "fieldsplit_1_pc_type": "jacobi",
    },
    "Schur / hypre": {
        "ksp_type": "gmres",
        "ksp_rtol": "1.0e-8",
        "pc_type": "fieldsplit",
        "pc_fieldsplit_detect_saddle_point": None,
        "pc_fieldsplit_type": "schur",
        "pc_fieldsplit_schur_fact_type": "full",
        "pc_fieldsplit_schur_precondition": "selfp",
        "fieldsplit_0_ksp_type": "preonly",
        "fieldsplit_0_pc_type": "jacobi",
        "fieldsplit_1_ksp_type": "preonly",
        "fieldsplit_1_pc_type": "hypre",
    },
}
```

4.2 Mixed Poisson model problem

Starting from $\nabla^2 u = -f$, introduce the flux $\sigma = \nabla u$. The first-order system is

$$\sigma - \nabla u = 0, \qquad \nabla \cdot \sigma = -f.$$

We choose a Raviart–Thomas space for σ and a discontinuous piecewise constant space for u.

Test the first equation with $\tau \in \Sigma_h$ and integrate the gradient term by parts:

$$\int_\Omega (\sigma - \nabla u) \cdot \tau \, dx = \int_\Omega \sigma \cdot \tau \, dx + \int_\Omega u \, \nabla \cdot \tau \, dx - \int_{\partial\Omega} u \, \tau \cdot n \, ds.$$

For homogeneous Dirichlet data $u = 0$ on $\partial\Omega$, the boundary term vanishes. Testing the conservation equation with $v \in V_h$ gives the mixed weak problem:

$$\begin{aligned} (\sigma, \tau)_\Omega + (u, \nabla \cdot \tau)_\Omega &= 0 && \forall \tau \in \Sigma_h, \\ (\nabla \cdot \sigma, v)_\Omega &= -(f, v)_\Omega && \forall v \in V_h. \end{aligned}$$

This weak form produces a saddle-point matrix

$$\begin{bmatrix} M & B^T \\ B & 0 \end{bmatrix}.$$

Here M is a mass matrix, so the first block is easy. The hard part is again the Schur complement $S = -BM^{-1}B^T$, which behaves like a Laplacian.

```python
def solve_mixed_poisson(nx, petsc_options, prefix):
    domain = mesh.create_unit_square(MPI.COMM_WORLD, nx, nx)
    sigma_el = element("RT", domain.basix_cell(), 1)
    scalar_el = element("DG", domain.basix_cell(), 0)
    W = fem.functionspace(domain, mixed_element([sigma_el, scalar_el]))

    sigma, u = ufl.TrialFunctions(W)
    tau, v = ufl.TestFunctions(W)
    x = ufl.SpatialCoordinate(domain)
    forcing = ufl.sin(math.pi * x[0]) * ufl.sin(math.pi * x[1])

    a = (ufl.inner(sigma, tau) + ufl.div(tau) * u + ufl.div(sigma) * v) * ufl.dx
    L = -forcing * v * ufl.dx

    problem = LinearProblem(
        a,
        L,
        petsc_options_prefix=prefix,
        petsc_options=petsc_options,
    )
    problem.solver.setConvergenceHistory()
    problem.solve()

    return {
        "iterations": problem.solver.getIterationNumber(),
        "reason": problem.solver.getConvergedReason(),
        "history": np.asarray(problem.solver.getConvergenceHistory(), dtype=float),
    }
```

```python
mixed_poisson_results = {}
for label, options in mixed_poisson_cases.items():
    prefix = "mixed_poisson_" + label.lower().replace(" / ", "_").replace(" ", "_") + "_"
    try:
        mixed_poisson_results[label] = solve_mixed_poisson(8, options, prefix)
    except Exception as err:
        mixed_poisson_results[label] = {"error": str(err)}

for label, result in mixed_poisson_results.items():
    if "error" in result:
        print(f"{label:14s} failed: {result['error']}")
    else:
        print(
            f"{label:14s} iterations={result['iterations']:4d} "
            f"reason={result['reason']:3d}"
        )
```

```
GMRES / ILU    iterations=  49 reason=  2
Schur / selfp  iterations=  34 reason=  2
Schur / hypre  iterations=  10 reason=  2
```

```python
fig, ax = plt.subplots(figsize=(6.5, 4.0))
for label, result in mixed_poisson_results.items():
    history = result.get("history")
    if history is None or len(history) == 0:
        continue
    ax.semilogy(history / history[0], marker="o", markersize=3, label=label)
ax.set_xlabel("KSP iteration")
ax.set_ylabel("relative residual norm")
ax.set_title("Saddle-point preconditioning")
ax.grid(True, which="both", alpha=0.3)
ax.legend()
plt.show()
```

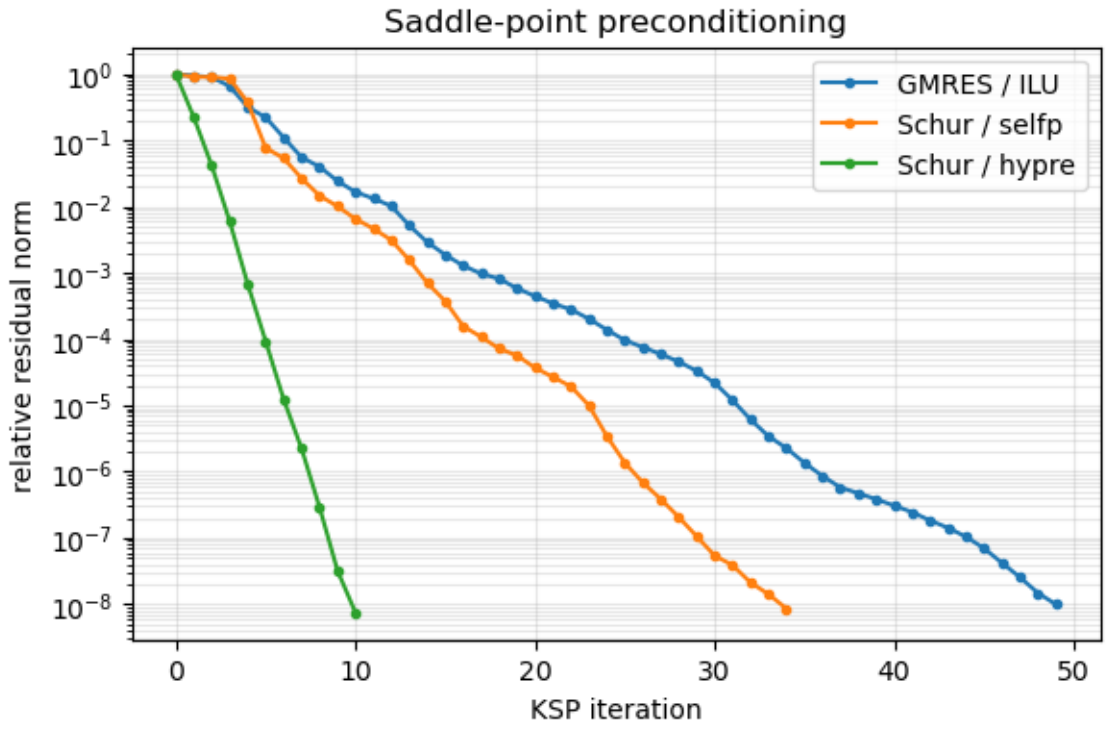


Figure 8: Residual histories for a Firedrake-style mixed Poisson saddle-point system.

4.3 Back to Stokes

For Stokes, the same Schur idea is used, but the blocks have fluid meaning:

- velocity block A: vector Laplacian or viscous operator, often preconditioned by multigrid,
- pressure Schur complement $S = -BA^{-1}B^T$: often approximated by pressure mass, pressure convection-diffusion, or least-squares commutator ideas.

The cell below solves a Taylor–Hood lid-driven cavity as a genuine FEniCSx block problem. We assemble the operator as `[[A, B^T], [B, 0]]` with `kind="nest"`, so PETSc receives a nested block matrix directly. This avoids relying on `pc_fieldsplit_detect_saddle_point`; the split is explicit in the FEniCSx problem definition. The detailed subsolver experiments are shown above for mixed Poisson, where the saddle-point structure is cleaner and the iteration-count story is easier to see.

```python
def clustered_unit_square(nx, ny):
    domain = mesh.create_unit_square(MPI.COMM_WORLD, nx, ny)
    X = domain.geometry.x
    X[:, 0] = 0.5 * (1.0 - np.cos(np.pi * X[:, 0]))
    X[:, 1] = 0.5 * (1.0 - np.cos(np.pi * X[:, 1]))
    return domain

# Finer than the solver-comparison toy mesh, with points clustered near walls
# and corners where lid-driven-cavity eddies live.
stokes_domain = clustered_unit_square(24, 24)
gdim = stokes_domain.geometry.dim

V_stokes = fem.functionspace(stokes_domain, ("Lagrange", 2, (gdim,)))
Q_stokes = fem.functionspace(stokes_domain, ("Lagrange", 1))

u = ufl.TrialFunction(V_stokes)
p = ufl.TrialFunction(Q_stokes)
v = ufl.TestFunction(V_stokes)
q = ufl.TestFunction(Q_stokes)

zero = fem.Constant(stokes_domain, PETSc.ScalarType(0))
a00 = ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
a01 = -p * ufl.div(v) * ufl.dx
a10 = q * ufl.div(u) * ufl.dx
a11 = zero * p * q * ufl.dx

L0 = ufl.inner(
    fem.Constant(stokes_domain, PETSc.ScalarType((0, 0))), v
) * ufl.dx
L1 = zero * q * ufl.dx

fdim = stokes_domain.topology.dim - 1
lid = fem.Function(V_stokes)
lid.interpolate(lambda X: np.vstack((np.ones(X.shape[1]), np.zeros(X.shape[1]))))
no_slip = fem.Function(V_stokes)
no_slip.interpolate(lambda X: np.zeros((gdim, X.shape[1])))

lid_facets = mesh.locate_entities_boundary(
    stokes_domain, fdim, lambda X: np.isclose(X[1], 1.0)
)
wall_facets = mesh.locate_entities_boundary(
    stokes_domain,
    fdim,
    lambda X: np.isclose(X[1], 0.0)
    | (
        (np.isclose(X[0], 0.0) | np.isclose(X[0], 1.0))
        & (X[1] < 1.0 - 1.0e-10)
    ),
)

stokes_bcs = [
```

```
    fem.dirichletbc(
        lid,
        fem.locate_dofs_topological(V_stokes, fdim, lid_facets),
    ),
    fem.dirichletbc(
        no_slip,
        fem.locate_dofs_topological(V_stokes, fdim, wall_facets),
    ),
]

# Pressure normalization. A pure Stokes pressure is unique only up to a constant.
p_zero = fem.Function(Q_stokes)
pressure_pin = fem.locate_dofs_geometrical(
    Q_stokes,
    lambda X: np.isclose(X[0], 0.0) & np.isclose(X[1], 0.0),
)
stokes_bcs.append(fem.dirichletbc(p_zero, pressure_pin))

velocity_h = fem.Function(V_stokes, name="velocity")
pressure_h = fem.Function(Q_stokes, name="pressure")

stokes_problem = LinearProblem(
    [[a00, a01], [a10, a11]],
    [L0, L1],
    u=[velocity_h, pressure_h],
    bcs=stokes_bcs,
    kind="nest",
    petsc_options_prefix="stokes_cavity_nest_",
    petsc_options={
        "ksp_type": "gmres",
        "ksp_rtol": "1.0e-8",
        "pc_type": "fieldsplit",
        "pc_fieldsplit_type": "schur",
        "pc_fieldsplit_schur_fact_type": "full",
        "pc_fieldsplit_schur_precondition": "selfp",
    },
)
stokes_problem.solver.setConvergenceHistory()
stokes_problem.solve()

print("Stokes GMRES + nested fieldsplit/Schur iterations:", stokes_problem.solver.getIteration
print("PETSc converged reason code:", stokes_problem.solver.getConvergedReason())
```

```
Stokes GMRES + nested fieldsplit/Schur iterations: 3
PETSc converged reason code: 2
```

```
def solve_stokes_with_options(label, options, matrix_kind="nest"):
    uh = fem.Function(V_stokes, name=f"velocity_{label}")
    ph = fem.Function(Q_stokes, name=f"pressure_{label}")
    problem = LinearProblem(
```

```
        [[a00, a01], [a10, a11]],
        [L0, L1],
        u=[uh, ph],
        bcs=stokes_bcs,
        kind=matrix_kind,
        petsc_options_prefix="stokes_" + label.lower().replace(" ", "_").replace("/", "_") + "
        petsc_options=options,
    )
    problem.solver.setConvergenceHistory()
    problem.solve()
    return {
        "iterations": problem.solver.getIterationNumber(),
        "reason": problem.solver.getConvergedReason(),
        "history": np.asarray(problem.solver.getConvergenceHistory(), dtype=float),
    }

stokes_solver_cases = {
    "GMRES / no PC": (
        {
            "ksp_type": "gmres",
            "ksp_rtol": "1.0e-8",
            "ksp_max_it": "100",
            "pc_type": "none",
        },
        "nest",
    ),
    "GMRES / ILU(2)": (
        {
            "ksp_type": "gmres",
            "ksp_rtol": "1.0e-8",
            "ksp_max_it": "100",
            "pc_type": "ilu",
            "pc_factor_levels": "2",
            "pc_factor_fill": "4.0",
        },
        "mpi",
    ),
    "Schur upper / selfp": (
        {
            "ksp_type": "gmres",
            "ksp_rtol": "1.0e-8",
            "ksp_max_it": "100",
            "pc_type": "fieldsplit",
            "pc_fieldsplit_type": "schur",
            "pc_fieldsplit_schur_fact_type": "upper",
            "pc_fieldsplit_schur_precondition": "selfp",
        },
        "nest",
    ),
    "Schur lower / selfp": (
        {
```

```
            "ksp_type": "gmres",
            "ksp_rtol": "1.0e-8",
            "ksp_max_it": "100",
            "pc_type": "fieldsplit",
            "pc_fieldsplit_type": "schur",
            "pc_fieldsplit_schur_fact_type": "lower",
            "pc_fieldsplit_schur_precondition": "selfp",
        },
        "nest",
    ),
    "Schur full / selfp": (
        {
            "ksp_type": "gmres",
            "ksp_rtol": "1.0e-8",
            "ksp_max_it": "100",
            "pc_type": "fieldsplit",
            "pc_fieldsplit_type": "schur",
            "pc_fieldsplit_schur_fact_type": "full",
            "pc_fieldsplit_schur_precondition": "selfp",
        },
        "nest",
    ),
}

stokes_histories = {}
for label, (options, matrix_kind) in stokes_solver_cases.items():
    result = solve_stokes_with_options(label, options, matrix_kind=matrix_kind)
    stokes_histories[label] = result
    print(
        f"{label:24s} iterations={result['iterations']:4d} "
        f"reason={result['reason']:3d}"
    )

fig, ax = plt.subplots(figsize=(6.6, 4.1))
for label, result in stokes_histories.items():
    history = result["history"]
    ax.semilogy(history / history[0], marker="o", markersize=3, label=label)
ax.set_xlabel("GMRES iteration")
ax.set_ylabel("relative residual norm")
ax.set_title("Stokes Schur and ILU preconditioners")
ax.grid(True, which="both", alpha=0.3)
ax.legend()
plt.show()
```

```
GMRES / no PC            iterations= 100 reason= -3
GMRES / ILU(2)           iterations=  49 reason=  2
Schur upper / selfp      iterations=   6 reason=  2
Schur lower / selfp      iterations=   4 reason=  2
Schur full / selfp       iterations=   3 reason=  2
```

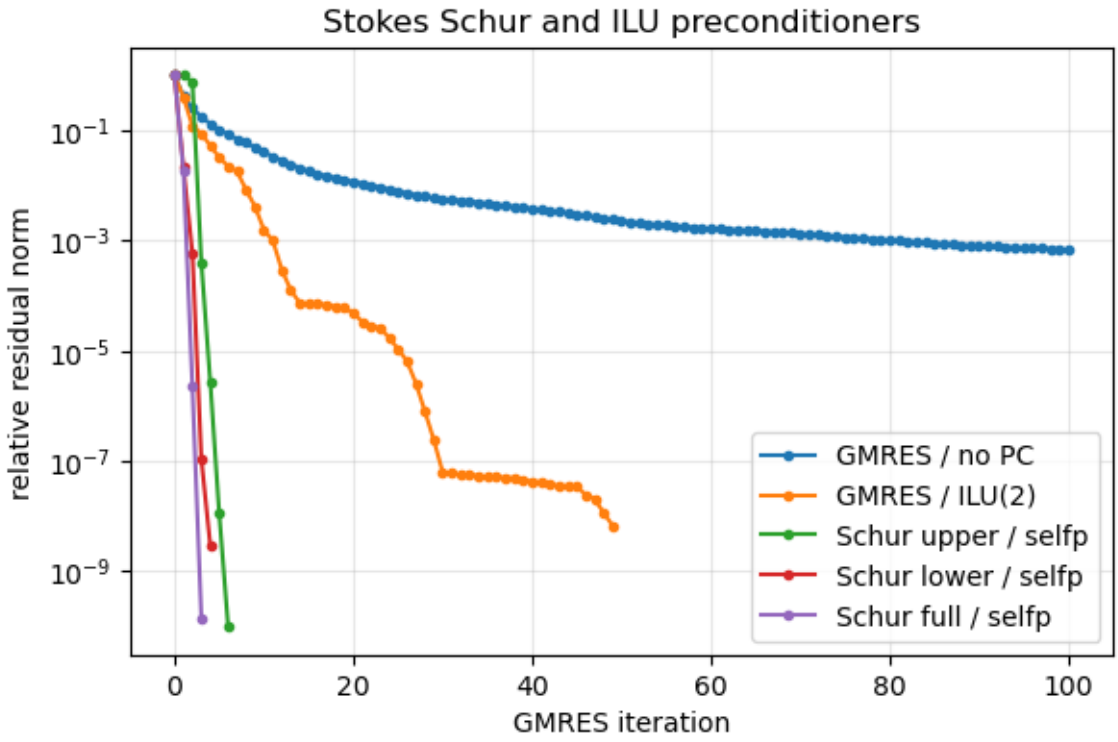


Figure 9: PETSc residual histories for nested Stokes solves and monolithic ILU(k).

The Schur factorization choice changes how much of the ideal block factorization PETSc mimics:

- `upper`/`lower`: triangular block factorizations; stronger than a generic sparse preconditioner.
- `full`: closest to the full block factorization; strongest in this small example.

The `GMRES / ILU(2)` curve is assembled as a monolithic AIJ matrix (`kind="mpi"`) because PETSc's ILU factorization does not apply directly to `MatNest`. PETSc's built-in option here is level-of-fill ILU(k), not threshold ILUT. It is still a useful "generic sparse preconditioner" comparison between no preconditioner and a Stokes-aware Schur split.

```
V_plot = fem.functionspace(stokes_domain, ("Lagrange", 1, (gdim,)))
Q_plot = fem.functionspace(stokes_domain, ("Lagrange", 1))
velocity_plot = fem.Function(V_plot)
pressure_plot_fn = fem.Function(Q_plot)
velocity_plot.interpolate(velocity_h)
pressure_plot_fn.interpolate(pressure_h)

velocity_coords = V_plot.tabulate_dof_coordinates()[:, :2]
velocity_values = velocity_plot.x.array.reshape((-1, gdim))
speed = np.linalg.norm(velocity_values, axis=1)
velocity_tri = mtri.Triangulation(velocity_coords[:, 0], velocity_coords[:, 1])

pressure_coords = Q_plot.tabulate_dof_coordinates()[:, :2]
pressure_values = pressure_plot_fn.x.array[: pressure_coords.shape[0]]
pressure_tri = mtri.Triangulation(pressure_coords[:, 0], pressure_coords[:, 1])

# Interpolate the scattered P1 values to a regular grid for streamplot.
```

```
xi = np.linspace(0.0, 1.0, 80)
yi = np.linspace(0.0, 1.0, 80)
Xi, Yi = np.meshgrid(xi, yi)
ux_interp = mtri.LinearTriInterpolator(velocity_tri, velocity_values[:, 0])
uy_interp = mtri.LinearTriInterpolator(velocity_tri, velocity_values[:, 1])
Ux = ux_interp(Xi, Yi).filled(0.0)
Uy = uy_interp(Xi, Yi).filled(0.0)
Speed_grid = np.sqrt(Ux**2 + Uy**2)

fig, axs = plt.subplots(1, 2, figsize=(9.5, 4.0), constrained_layout=True)

speed_plot = axs[0].contourf(Xi, Yi, Speed_grid, levels=30)
axs[0].streamplot(
    xi,
    yi,
    Ux,
    Uy,
    color="white",
    density=1.4,
    linewidth=0.8,
    arrowsize=0.8,
)
fig.colorbar(speed_plot, ax=axs[0], label="|u|")
axs[0].set_title("velocity streamlines")
axs[0].set_aspect("equal")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")

p_plot = axs[1].tricontourf(pressure_tri, pressure_values, levels=30)
fig.colorbar(p_plot, ax=axs[1], label="p")
axs[1].set_title("pressure")
axs[1].set_aspect("equal")
axs[1].set_xlabel("x")
axs[1].set_ylabel("y")
plt.show()
```

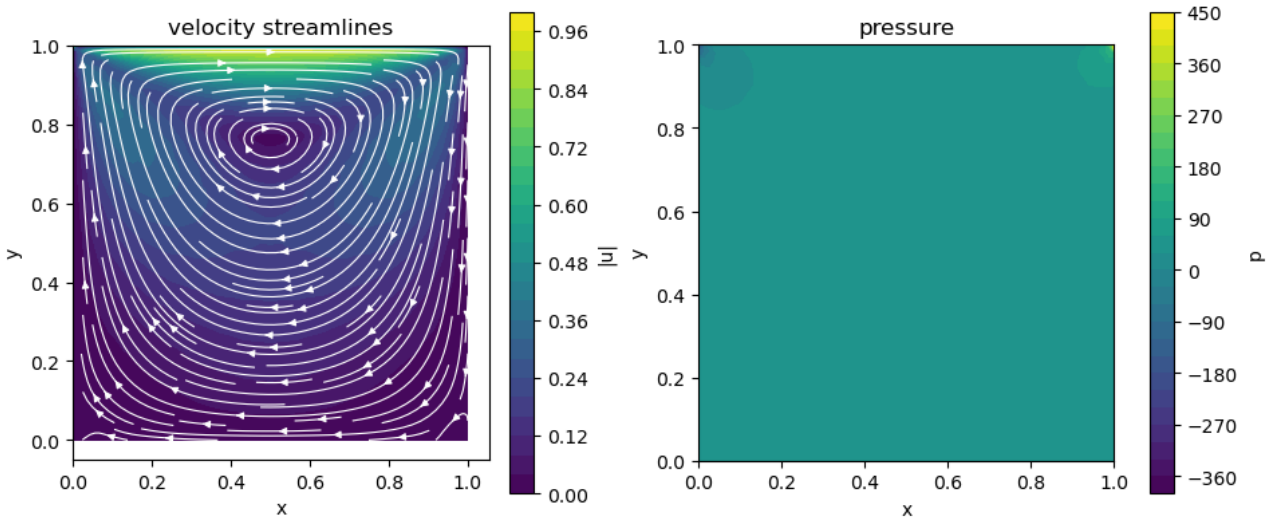


Figure 10: Taylor–Hood Stokes lid-driven cavity: streamlines, velocity magnitude, and pressure.

Warning

We would need finer mesh for bottom vortices (similar to, e.g., this demo) to appear.

What carries over from the Firedrake demo?

The important move is not a Firedrake-specific API. It is the block-factorization idea: use a preconditioner that sees the two fields, approximate the easy block directly, and approximate the Schur complement with an operator that has the right spectral behavior.

Pressure nullspace

Pinning one pressure degree of freedom is convenient for a small notebook. It's also possible to provide and explicitly attach the pressure nullspace.

5 Some further examples...

5.1 Time-dependent problems and PETSc TS

```
"""
Compare three mathematical viewpoints for the heat equation in FEniCSx:

    u_t - u_xx = 0 on x in (0, 1), t in (0, T)
    natural Neumann boundary conditions
    u(x, 0) = cos(pi x)

Exact solution:

    u(x, t) = exp(-pi^2 t) cos(pi x)
```

```
Methods:
  1. Method of lines: spatial FEM first, PETSc TS solves M Udot + K U = 0
  2. Rothe method: time discretization first, backward Euler FEM step loop
  3. Global space-time FEM: solve in (x,t) as one 2D weak problem

Run examples:

    mpirun -n 1 python compare_heat_fenicsx_methods.py --method rothe
    mpirun -n 1 python compare_heat_fenicsx_methods.py --method ts -ts_type beuler
    mpirun -n 1 python compare_heat_fenicsx_methods.py --method spacetime
    mpirun -n 1 python compare_heat_fenicsx_methods.py --method all

You can pass PETSc TS options directly, e.g.

    mpirun -n 1 python compare_heat_fenicsx_methods.py --method ts -ts_type bdf -ts_adapt_type
"""

from __future__ import annotations

import argparse
import math

import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from matplotlib import animation
import numpy as np
from IPython.display import HTML, display
from mpi4py import MPI
from petsc4py import PETSc

import ufl
from dolfinx import fem, mesh
from dolfinx.fem.petsc import assemble_matrix, assemble_vector, LinearProblem


def exact_1d(x: np.ndarray, t: float) -> np.ndarray:
    return np.exp(-math.pi**2 * t) * np.cos(math.pi * x[0])


def l2_error_1d(u_h: fem.Function, T: float) -> float:
    V = u_h.function_space
    x = ufl.SpatialCoordinate(V.mesh)
    u_ex = ufl.exp(-(math.pi**2) * T) * ufl.cos(math.pi * x[0])
    err_form = fem.form((u_h - u_ex) ** 2 * ufl.dx)
    local = fem.assemble_scalar(err_form)
    global_err = V.mesh.comm.allreduce(local, op=MPI.SUM)
    return math.sqrt(global_err)


def solve_rothe(nx: int, dt: float, T: float, degree: int = 1) -> tuple[fem.Function, float]:
    """Backward Euler / Rothe: solve one elliptic FEM problem per time step."""
```

```
    domain = mesh.create_interval(MPI.COMM_WORLD, nx, [0.0, 1.0])
    V = fem.functionspace(domain, ("Lagrange", degree))

    u_n = fem.Function(V, name="u_n")
    u_n.interpolate(lambda x: np.cos(math.pi * x[0]))

    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)

    a = (u * v / dt + ufl.dot(ufl.grad(u), ufl.grad(v))) * ufl.dx
    L = (u_n * v / dt) * ufl.dx

    problem = LinearProblem(
        a,
        L,
        bcs=[],
        petsc_options={
            "ksp_type": "preonly",
            "pc_type": "lu",
        },
        petsc_options_prefix="rothe_",
    )

    t = 0.0
    nsteps = int(round(T / dt))
    for _ in range(nsteps):
        uh = problem.solve()
        u_n.x.array[:] = uh.x.array
        t += dt

    u_n.name = "u_rothe"
    return u_n, l2_error_1d(u_n, t)


def solve_ts(nx: int, dt: float, T: float, degree: int = 1) -> tuple[fem.Function, float]:
    """Method of lines: assemble M,K and let PETSc TS solve M Udot + K U = 0."""
    comm = MPI.COMM_WORLD
    domain = mesh.create_interval(comm, nx, [0.0, 1.0])
    V = fem.functionspace(domain, ("Lagrange", degree))

    u_trial = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)

    M = assemble_matrix(fem.form(u_trial * v * ufl.dx), bcs=[])
    M.assemble()
    K = assemble_matrix(fem.form(ufl.dot(ufl.grad(u_trial), ufl.grad(v)) * ufl.dx), bcs=[])
    K.assemble()

    u = fem.Function(V, name="u_ts")
    u.interpolate(lambda x: np.cos(math.pi * x[0]))
    u.x.scatter_forward()
```

```
    # Work vectors/matrix for callbacks
    F_work = M.createVecLeft()
    J = M.copy()
    J.setOption(PETSc.Mat.Option.NEW_NONZERO_ALLOCATION_ERR, False)

    def ifunction(ts, t, U, Udot, F):
        # F(U,Udot) = M Udot + K U
        M.mult(Udot, F)
        K.multAdd(U, F, F)

    def ijacobian(ts, t, U, Udot, shift, Jmat, Pmat):
        # dF/dU + shift dF/dUdot = K + shift M
        Pmat.zeroEntries()
        Pmat.axpy(shift, M, structure=PETSc.Mat.Structure.SAME_NONZERO_PATTERN)
        Pmat.axpy(1.0, K, structure=PETSc.Mat.Structure.SAME_NONZERO_PATTERN)
        Pmat.assemble()
        if Jmat.handle != Pmat.handle:
            Jmat.zeroEntries()
            Jmat.axpy(1.0, Pmat, structure=PETSc.Mat.Structure.SAME_NONZERO_PATTERN)
            Jmat.assemble()

    ts = PETSc.TS().create(comm)
    ts.setProblemType(PETSc.TS.ProblemType.NONLINEAR)
    ts.setType(PETSc.TS.Type.BEULER)
    ts.setIFunction(ifunction, F_work)
    ts.setIJacobian(ijacobian, J, J)
    ts.setTime(0.0)
    ts.setTimeStep(dt)
    ts.setMaxTime(T)
    ts.setExactFinalTime(PETSc.TS.ExactFinalTime.MATCHSTEP)
    ts.setFromOptions()

    ts.solve(u.x.petsc_vec)
    u.x.scatter_forward()
    return u, l2_error_1d(u, float(ts.getTime()))


def solve_spacetime(nx: int, nt: int, T: float, degree: int = 1) -> tuple[fem.Function, float]:
    """
    Global continuous Galerkin space-time solve on Q = (0,1)x(0,T):

        int_Q (u_t v + u_x v_x) dx dt = 0,
        u(x,0) = cos(pi x).

    This is a compact demonstration of the fully coupled space-time idea.
    It is not meant as the most stable/general parabolic space-time method.
    """
    comm = MPI.COMM_WORLD
    domain = mesh.create_rectangle(
        comm,
```

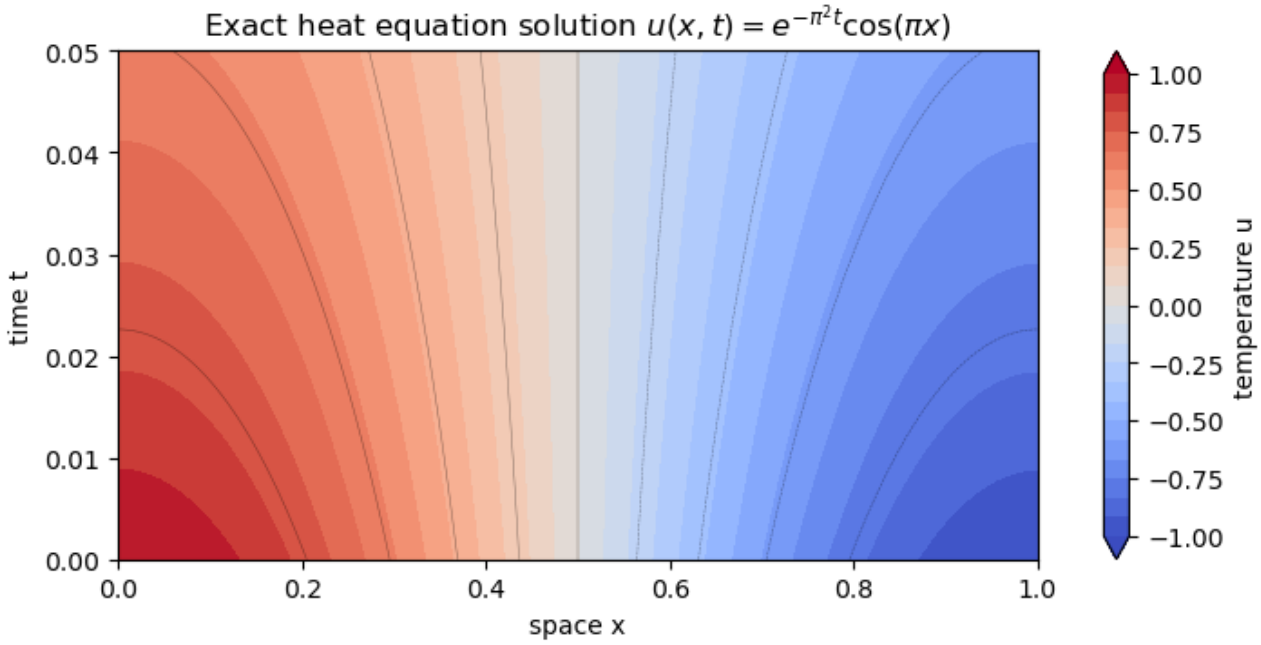
```
        points=((0.0, 0.0), (1.0, T)),
        n=(nx, nt),
        cell_type=mesh.CellType.triangle,
    )
    V = fem.functionspace(domain, ("Lagrange", degree))

    u = ufl.TrialFunction(V)
    v = ufl.TestFunction(V)
    x = ufl.SpatialCoordinate(domain)

    # x[0] is physical space, x[1] is time.
    a = (u.dx(1) * v + u.dx(0) * v.dx(0)) * ufl.dx
    L = fem.Constant(domain, PETSc.ScalarType(0.0)) * v * ufl.dx

    def initial_boundary(X):
        return np.isclose(X[1], 0.0)

    def initial_value(X):
        return np.cos(math.pi * X[0])

    facets = mesh.locate_entities_boundary(domain, domain.topology.dim - 1, initial_boundary)
    dofs = fem.locate_dofs_topological(V, domain.topology.dim - 1, facets)
    u0 = fem.Function(V)
    u0.interpolate(initial_value)
    bc = fem.dirichletbc(u0, dofs)

    problem = LinearProblem(
        a,
        L,
        bcs=[bc],
        petsc_options={
            "ksp_type": "preonly",
            "pc_type": "lu",
        },
        petsc_options_prefix="spacetime_",
    )
    Uxt = problem.solve()
    Uxt.name = "u_spacetime_on_xt"

    # Measure error on the final-time boundary approximately by evaluating at mesh vertices
    # with t == T. This is a simple diagnostic, not a proper trace L2 norm.
    coords = domain.geometry.x
    final_vertices = np.where(np.isclose(coords[:, 1], T))[0]
    # Evaluation at arbitrary points needs cell lookup; for this simple structured mesh,
    # report a nodal max error by interpolating exact values at DOF coordinates instead.
    dof_coords = V.tabulate_dof_coordinates()
    owned = slice(0, V.dofmap.index_map.size_local * V.dofmap.index_map_bs)
    vals = Uxt.x.array[owned]
    dofs_final = np.where(np.isclose(dof_coords[owned, 1], T))[0]
    if len(dofs_final) > 0:
        err_local = np.max(
```

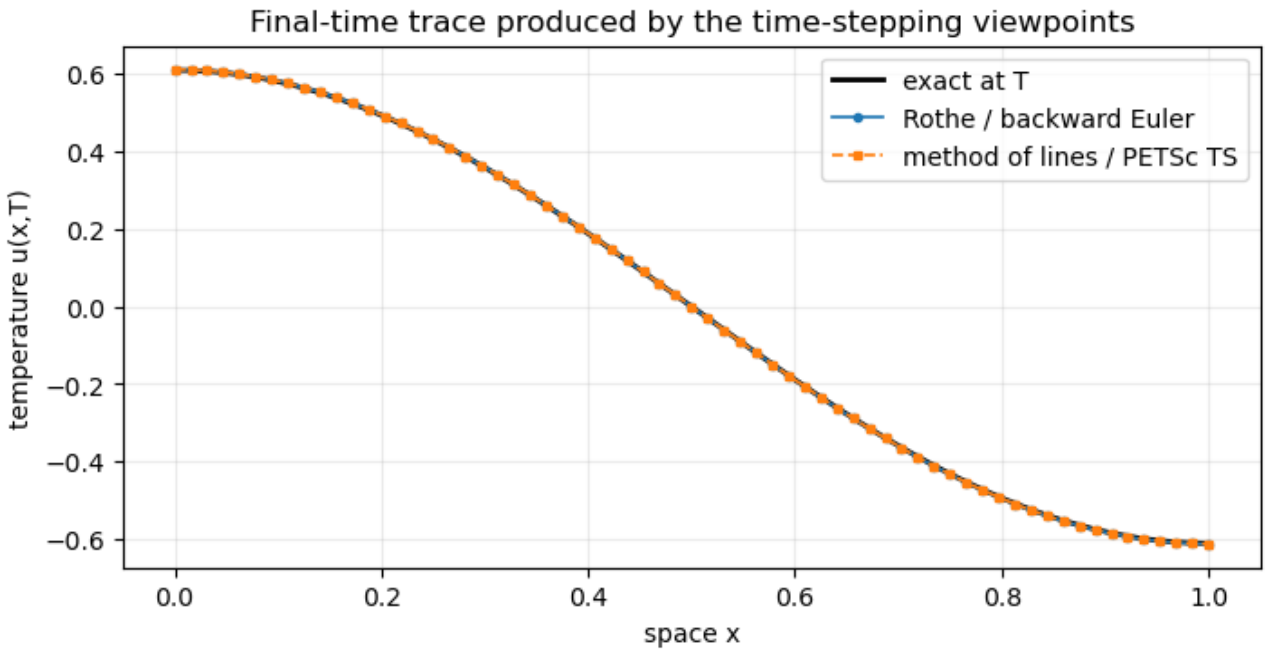
```
            np.abs(
                vals[dofs_final]
                - np.exp(-math.pi**2 * T) * np.cos(math.pi * dof_coords[owned][dofs_final, 0])
            )
        )
    else:
        err_local = 0.0
    err_global = comm.allreduce(err_local, op=MPI.MAX)
    return Uxt, float(err_global)


def dof_coordinates_and_values(u_h: fem.Function) -> tuple[np.ndarray, np.ndarray]:
    """Return local scalar dof coordinates and values sorted by x-coordinate."""
    V = u_h.function_space
    n_local = V.dofmap.index_map.size_local * V.dofmap.index_map_bs
    coords = V.tabulate_dof_coordinates()[:n_local]
    values = np.asarray(u_h.x.array[:n_local].real)
    order = np.argsort(coords[:, 0])
    return coords[order], values[order]


def plot_exact_solution_over_time(T: float) -> None:
    xs = np.linspace(0.0, 1.0, 240)
    ts = np.linspace(0.0, T, 180)
    X, Tau = np.meshgrid(xs, ts)
    U = np.exp(-math.pi**2 * Tau) * np.cos(math.pi * X)

    fig, ax = plt.subplots(figsize=(7.0, 3.6), constrained_layout=True)
    levels = np.linspace(-1.0, 1.0, 25)
    cf = ax.contourf(X, Tau, U, levels=levels, cmap="coolwarm", extend="both")
    ax.contour(X, Tau, U, levels=np.linspace(-0.8, 0.8, 9), colors="black", linewidths=0.35, a
    ax.set_title(r"Exact heat equation solution $u(x,t)=e^{-\pi^2t}\cos(\pi x)$")
    ax.set_xlabel("space x")
    ax.set_ylabel("time t")
    fig.colorbar(cf, ax=ax, label="temperature u")
    plt.show()


def plot_final_trace(rothe: fem.Function | None, ts_solution: fem.Function | None, T: float) -
    fig, ax = plt.subplots(figsize=(7.0, 3.6), constrained_layout=True)
    xs = np.linspace(0.0, 1.0, 300)
    ax.plot(xs, np.exp(-math.pi**2 * T) * np.cos(math.pi * xs), color="black", lw=2.0, label="

    if rothe is not None:
        coords, vals = dof_coordinates_and_values(rothe)
        ax.plot(coords[:, 0], vals, "o-", ms=3.0, lw=1.2, label="Rothe / backward Euler")
    if ts_solution is not None:
        coords, vals = dof_coordinates_and_values(ts_solution)
        ax.plot(coords[:, 0], vals, "s--", ms=3.0, lw=1.2, label="method of lines / PETSc TS")

    ax.set_title("Final-time trace produced by the time-stepping viewpoints")
```

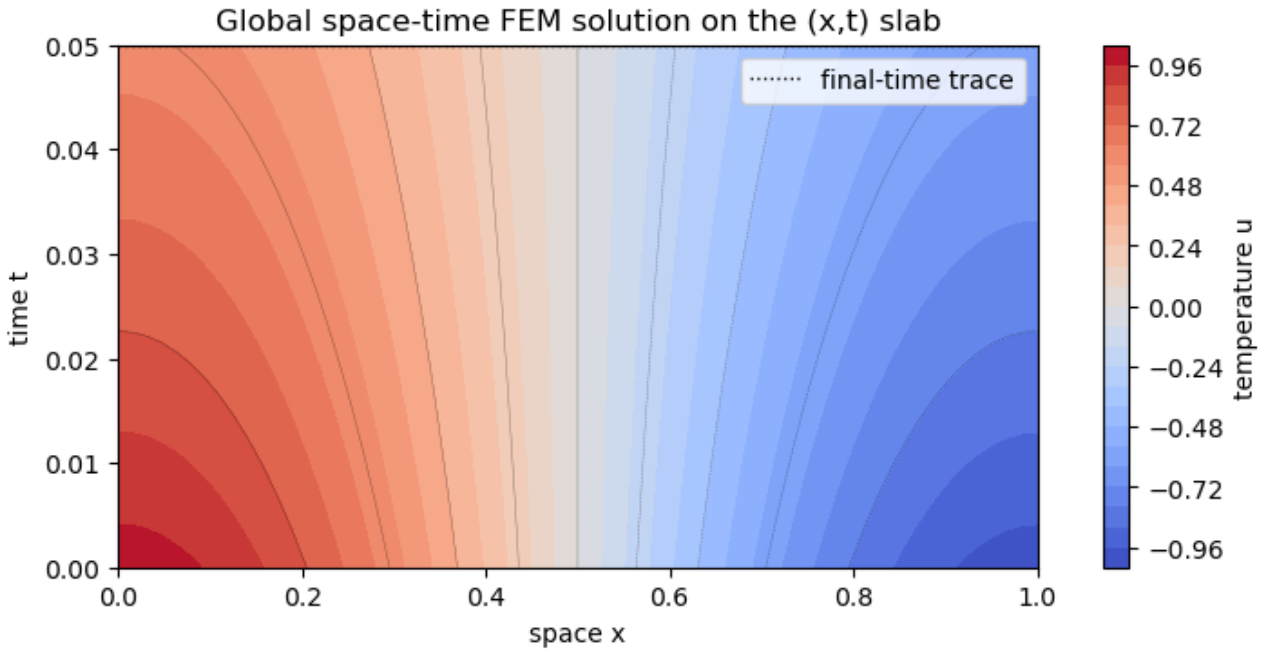
```
    ax.set_xlabel("space x")
    ax.set_ylabel("temperature u(x,T)")
    ax.grid(True, alpha=0.25)
    ax.legend(loc="best")
    plt.show()


def plot_spacetime_solution(Uxt: fem.Function | None, T: float) -> None:
    if Uxt is None:
        return

    coords, vals = dof_coordinates_and_values(Uxt)
    triangulation = mtri.Triangulation(coords[:, 0], coords[:, 1])

    fig, ax = plt.subplots(figsize=(7.0, 3.6), constrained_layout=True)
    cf = ax.tricontourf(triangulation, vals, levels=25, cmap="coolwarm")
    ax.tricontour(triangulation, vals, levels=10, colors="black", linewidths=0.3, alpha=0.45)
    ax.axhline(T, color="black", lw=1.0, ls=":", label="final-time trace")
    ax.set_title("Global space-time FEM solution on the (x,t) slab")
    ax.set_xlabel("space x")
    ax.set_ylabel("time t")
    ax.legend(loc="upper right")
    fig.colorbar(cf, ax=ax, label="temperature u")
    plt.show()


def main() -> None:
    parser = argparse.ArgumentParser()
    parser.add_argument("--method", choices=["ts", "rothe", "spacetime", "all"], default="all"
    parser.add_argument("--nx", type=int, default=64)
    parser.add_argument("--nt", type=int, default=64, help="time slabs for space-time method")
    parser.add_argument("--dt", type=float, default=1.0e-3)
    parser.add_argument("--T", type=float, default=0.05)
    parser.add_argument("--degree", type=int, default=1)
    args, _ = parser.parse_known_args()

    comm = MPI.COMM_WORLD
    rothe_solution = None
    ts_solution = None
    spacetime_solution = None

    if args.method in {"rothe", "all"}:
        rothe_solution, err = solve_rothe(args.nx, args.dt, args.T, args.degree)
        if comm.rank == 0:
            print(f"Rothe / backward Euler:     L2 error at T = {err:.6e}")

    if args.method in {"ts", "all"}:
        ts_solution, err = solve_ts(args.nx, args.dt, args.T, args.degree)
        if comm.rank == 0:
            print(f"MoL / PETSc TS:             L2 error at T = {err:.6e}")
```

```
    if args.method in {"spacetime", "all"}:
        spacetime_solution, err = solve_spacetime(args.nx, args.nt, args.T, args.degree)
        if comm.rank == 0:
            print(f"Global space-time FEM:      nodal max error at final trace = {err:.6e}")

    if comm.rank == 0:
        plot_exact_solution_over_time(args.T)
        plot_final_trace(rothe_solution, ts_solution, args.T)
        plot_spacetime_solution(spacetime_solution, args.T)


if __name__ == "__main__":
    main()
```

```
Rothe / backward Euler:     L2 error at T = 9.171406e-04
MoL / PETSc TS:             L2 error at T = 9.171406e-04
Global space-time FEM:      nodal max error at final trace = 4.034253e-04
```







5.2 A Boltzmann-type kinetic equation

The Boltzmann equation evolves a particle distribution function $f(x, v, t)$ in phase space. A common reduced model is the BGK relaxation equation

$$\partial_t f + v\,\partial_x f = \frac{1}{\tau}(f^{\mathrm{eq}} - f),$$

where x is position, v is molecular velocity, τ is a relaxation time, and f^{eq} is a Maxwellian-like equilibrium. This keeps the transport-collision structure of the Boltzmann equation but avoids the full nonlinear collision integral.

For a finite element method on the phase-space domain $Q = (0, L_x) \times (v_{\min}, v_{\max})$, use a trial function $f_h^{n+1} \in V_h$ and a test function $g_h \in V_h$. Backward Euler in time gives

$$\int_Q \frac{f_h^{n+1} - f_h^n}{\Delta t} g_h \, dx \, dv + \int_Q v \, \partial_x f_h^{n+1} g_h \, dx \, dv + \int_Q \frac{1}{\tau} f_h^{n+1} g_h \, dx \, dv = \int_Q \frac{1}{\tau} f^{\mathrm{eq}} g_h \, dx \, dv.$$

Since the transport term is hyperbolic, the demo below adds a small streamline-upwind stabilization term and applies inflow boundary data: at $x = 0$ for $v > 0$ and at $x = L_x$ for $v < 0$. The final plot shows both the microscopic phase-space distribution and two macroscopic moments extracted from it.

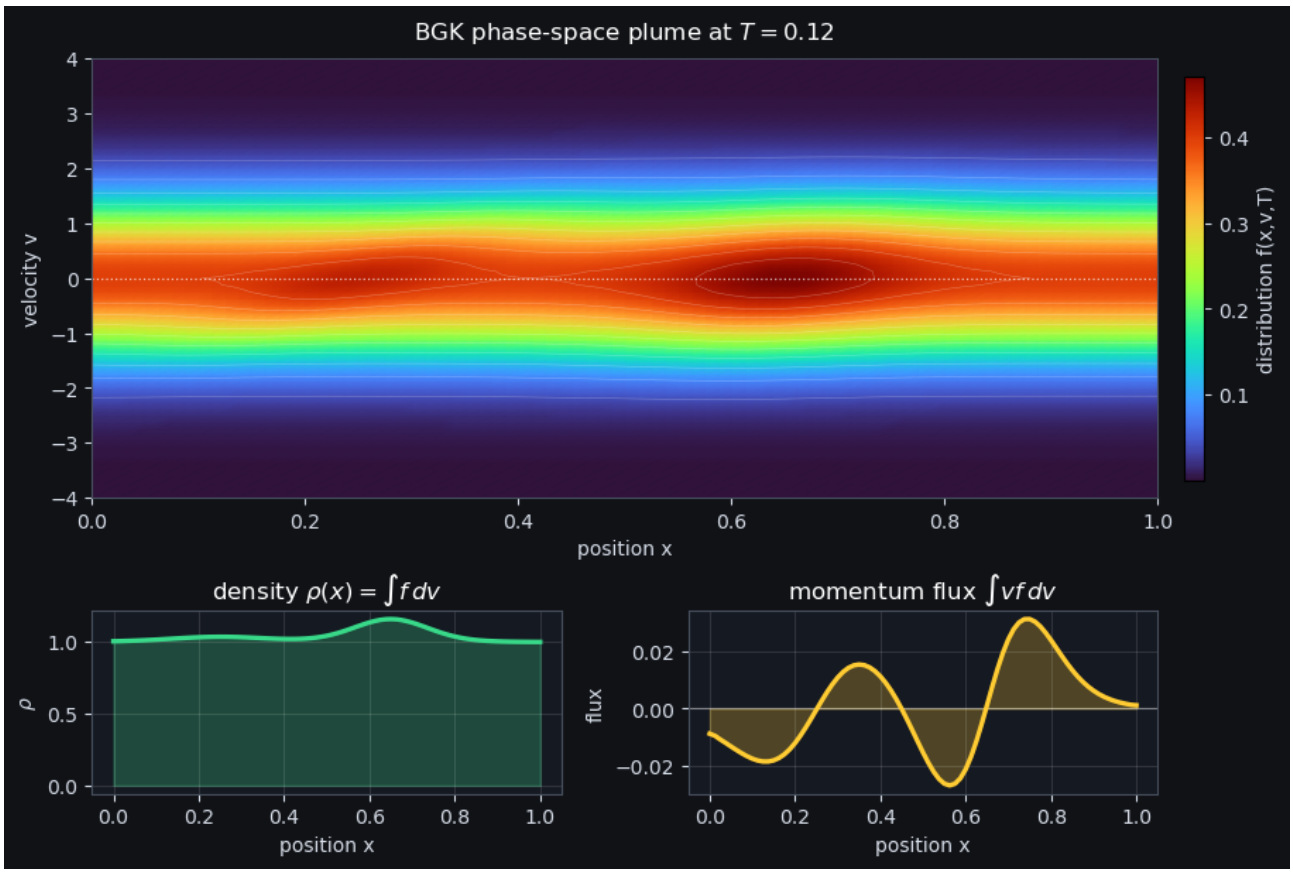
```
import math

import matplotlib.pyplot as plt
import matplotlib.tri as mtri
from matplotlib import animation
import numpy as np
from IPython.display import HTML, display
from mpi4py import MPI
from petsc4py import PETSc

import ufl
from dolfinx import fem, mesh
from dolfinx.fem.petsc import LinearProblem


def maxwellian(v):
    return np.exp(-0.5 * v**2) / math.sqrt(2.0 * math.pi)


comm = MPI.COMM_WORLD
Lx = 1.0
vmin, vmax = -4.0, 4.0
nx, nv = 80, 64
dt, T = 2.0e-3, 0.12
tau = 5.0e-2
supg_delta = 2.5e-3

# Mesh coordinates are (x, v), so the FEM problem lives in phase space.
phase = mesh.create_rectangle(
    comm,
    points=((0.0, vmin), (Lx, vmax)),
    n=(nx, nv),
    cell_type=mesh.CellType.triangle,
)
V = fem.functionspace(phase, ("Lagrange", 1))

f_n = fem.Function(V, name="f_n")
f_eq = fem.Function(V, name="f_eq")
f_in = fem.Function(V, name="f_in")
```

```
def equilibrium(X):
    rho = 1.0 + 0.20 * np.exp(-80.0 * (X[0] - 0.65) ** 2)
    return rho * maxwellian(X[1])


def initial_distribution(X):
    rho = 1.0 + 0.85 * np.exp(-160.0 * (X[0] - 0.25) ** 2)
    return rho * maxwellian(X[1])


def inflow_distribution(X):
    return maxwellian(X[1])


f_n.interpolate(initial_distribution)
f_eq.interpolate(equilibrium)
f_in.interpolate(inflow_distribution)

xv = ufl.SpatialCoordinate(phase)
v_coord = xv[1]
f = ufl.TrialFunction(V)
g = ufl.TestFunction(V)

a = (
    f * g / dt
    + v_coord * f.dx(0) * g
    + f * g / tau
    + supg_delta * v_coord * f.dx(0) * v_coord * g.dx(0)
) * ufl.dx
L = (f_n * g / dt + f_eq * g / tau) * ufl.dx

def inflow_boundary(X):
    left_in = np.isclose(X[0], 0.0) & (X[1] > 0.0)
    right_in = np.isclose(X[0], Lx) & (X[1] < 0.0)
    return left_in | right_in


facets = mesh.locate_entities_boundary(phase, phase.topology.dim - 1, inflow_boundary)
dofs = fem.locate_dofs_topological(V, phase.topology.dim - 1, facets)
bc = fem.dirichletbc(f_in, dofs)

problem = LinearProblem(
    a,
    L,
    bcs=[bc],
    petsc_options={"ksp_type": "preonly", "pc_type": "lu"},
    petsc_options_prefix="bgk_",
)

if comm.rank == 0:
    phase.topology.create_connectivity(phase.topology.dim, 0)
```

```
    cells = phase.topology.connectivity(phase.topology.dim, 0).array.reshape(-1, 3)
    nverts = phase.topology.index_map(0).size_local + phase.topology.index_map(0).num_ghosts
    coords = phase.geometry.x[:nverts, :2]
    vertex_dofs = fem.locate_dofs_topological(V, 0, np.arange(nverts, dtype=np.int32))
    tri = mtri.Triangulation(coords[:, 0], coords[:, 1], triangles=cells)

    xs = np.unique(np.round(coords[:, 0], 12))
    vs = np.unique(np.round(coords[:, 1], 12))
    x_lookup = {x: i for i, x in enumerate(xs)}
    v_lookup = {v: i for i, v in enumerate(vs)}

    def snapshot(u_h, time):
        values = u_h.x.array[vertex_dofs].real.copy()
        F = np.full((len(vs), len(xs)), np.nan)
        for (x, v), val in zip(np.round(coords, 12), values):
            F[v_lookup[v], x_lookup[x]] = val
        rho = np.trapezoid(F, vs, axis=0)
        flux = np.trapezoid(F * vs[:, None], vs, axis=0)
        return {"time": time, "values": values, "F": F, "rho": rho, "flux": flux}

    snapshots = [snapshot(f_n, 0.0)]

nsteps = int(round(T / dt))
frame_stride = max(1, nsteps // 24)
for step in range(1, nsteps + 1):
    f_h = problem.solve()
    f_n.x.array[:] = f_h.x.array
    f_n.x.scatter_forward()
    if comm.rank == 0 and (step % frame_stride == 0 or step == nsteps):
        snapshots.append(snapshot(f_n, step * dt))

if comm.rank == 0:
    final = snapshots[-1]
    values = final["values"]
    F = final["F"]
    rho = final["rho"]
    flux = final["flux"]

    fig = plt.figure(figsize=(9.0, 6.0), constrained_layout=True)
    gs = fig.add_gridspec(2, 2, height_ratios=(3.0, 1.25), width_ratios=(1.0, 1.0))
    ax_phase = fig.add_subplot(gs[0, :])
    ax_rho = fig.add_subplot(gs[1, 0])
    ax_flux = fig.add_subplot(gs[1, 1])

    fig.patch.set_facecolor("#101216")
    for ax in (ax_phase, ax_rho, ax_flux):
        ax.set_facecolor("#151922")
        ax.tick_params(colors="#d7dde8")
        for spine in ax.spines.values():
            spine.set_color("#49515f")
```

```
    pcm = ax_phase.tripcolor(tri, values, shading="gouraud", cmap="turbo")
    ax_phase.tricontour(tri, values, levels=12, colors="white", linewidths=0.35, alpha=0.35)
    ax_phase.axhline(0.0, color="white", lw=0.9, ls=":", alpha=0.7)
    ax_phase.set_title(rf"BGK phase-space plume at $T={T}$", color="white", pad=10)
    ax_phase.set_xlabel("position x", color="#d7dde8")
    ax_phase.set_ylabel("velocity v", color="#d7dde8")
    cbar = fig.colorbar(pcm, ax=ax_phase, pad=0.01, shrink=0.92)
    cbar.set_label("distribution f(x,v,T)", color="#d7dde8")
    cbar.ax.yaxis.set_tick_params(color="#d7dde8")
    plt.setp(cbar.ax.get_yticklabels(), color="#d7dde8")

    ax_rho.fill_between(xs, rho, color="#39d98a", alpha=0.25)
    ax_rho.plot(xs, rho, color="#39d98a", lw=2.4)
    ax_rho.set_title(r"density $\rho(x)=\int f\,dv$", color="white")
    ax_rho.set_xlabel("position x", color="#d7dde8")
    ax_rho.set_ylabel(r"$\rho$", color="#d7dde8")
    ax_rho.grid(True, color="white", alpha=0.12)

    ax_flux.axhline(0.0, color="white", lw=0.8, alpha=0.4)
    ax_flux.fill_between(xs, flux, color="#ffcc33", alpha=0.25)
    ax_flux.plot(xs, flux, color="#ffcc33", lw=2.4)
    ax_flux.set_title(r"momentum flux $\int vf\,dv$", color="white")
    ax_flux.set_xlabel("position x", color="#d7dde8")
    ax_flux.set_ylabel("flux", color="#d7dde8")
    ax_flux.grid(True, color="white", alpha=0.12)

    plt.show()

    fig_anim = plt.figure(figsize=(8.8, 5.6), constrained_layout=True)
    gs_anim = fig_anim.add_gridspec(2, 1, height_ratios=(3.0, 1.1))
    ax_anim = fig_anim.add_subplot(gs_anim[0, 0])
    ax_moment = fig_anim.add_subplot(gs_anim[1, 0])
    fig_anim.patch.set_facecolor("#101216")
    for ax in (ax_anim, ax_moment):
        ax.set_facecolor("#151922")
        ax.tick_params(colors="#d7dde8")
        for spine in ax.spines.values():
            spine.set_color("#49515f")

    vmin_anim = min(np.nanmin(s["F"]) for s in snapshots)
    vmax_anim = max(np.nanmax(s["F"]) for s in snapshots)
    rho_max = 1.05 * max(np.nanmax(s["rho"]) for s in snapshots)

    image = ax_anim.imshow(
        snapshots[0]["F"],
        origin="lower",
        extent=(xs[0], xs[-1], vs[0], vs[-1]),
        aspect="auto",
        cmap="turbo",
        vmin=vmin_anim,
        vmax=vmax_anim,
```

```
        interpolation="bilinear",
    )
    ax_anim.axhline(0.0, color="white", lw=0.8, ls=":", alpha=0.65)
    ax_anim.set_xlabel("position x", color="#d7dde8")
    ax_anim.set_ylabel("velocity v", color="#d7dde8")
    title = ax_anim.set_title("", color="white", pad=8)
    cbar_anim = fig_anim.colorbar(image, ax=ax_anim, pad=0.01, shrink=0.9)
    cbar_anim.set_label("distribution f", color="#d7dde8")
    cbar_anim.ax.yaxis.set_tick_params(color="#d7dde8")
    plt.setp(cbar_anim.ax.get_yticklabels(), color="#d7dde8")

    (rho_line,) = ax_moment.plot(xs, snapshots[0]["rho"], color="#39d98a", lw=2.4)
    rho_fill = [ax_moment.fill_between(xs, snapshots[0]["rho"], color="#39d98a", alpha=0.25)]
    ax_moment.set_xlim(xs[0], xs[-1])
    ax_moment.set_ylim(0.0, rho_max)
    ax_moment.set_xlabel("position x", color="#d7dde8")
    ax_moment.set_ylabel(r"$\rho(x)$", color="#d7dde8")
    ax_moment.grid(True, color="white", alpha=0.12)

    def update_frame(i):
        snap = snapshots[i]
        image.set_data(snap["F"])
        rho_line.set_ydata(snap["rho"])
        rho_fill[0].remove()
        rho_fill[0] = ax_moment.fill_between(xs, snap["rho"], color="#39d98a", alpha=0.25)
        title.set_text(rf"BGK phase-space evolution, $t={snap['time']:.3f}$")
        return image, rho_line, rho_fill[0], title

    anim = animation.FuncAnimation(
        fig_anim,
        update_frame,
        frames=len(snapshots),
        interval=120,
        blit=False,
    )
    display(HTML(anim.to_jshtml()))
    plt.close(fig_anim)
```



(a) Phase-space FEM solution of a BGK-type kinetic equation and its density/flux moments.

```
<IPython.core.display.HTML object>
```

(b)

Figure 11

5.3 Hemker problem (convection-dominated)

As a final solver/stabilization example, we compare three discretizations for the Hemker convection-diffusion problem, see, e.g. [3] or [4], around a circular obstacle:

$$-\varepsilon \Delta u + \beta \cdot \nabla u = 0, \qquad \beta = (1, 0)^\top.$$

The inflow boundary is set to $u = 0$, the cylinder is set to $u = 1$, and the outflow is left natural. We reuse the helper functions from `hemker_fenicsx_convection_dominated.py` without modifying that file.

The point of this comparison is qualitative: as ε decreases, the problem becomes increasingly convection dominated. Plain continuous Galerkin develops strong oscillations, while SUPG and DG-upwind add stabilization.

```
import importlib.util
import sys
from pathlib import Path

```

```
hemker_path_candidates = [
    Path("lectures/hemker_fenicsx_convection_dominated.py"),
    Path("hemker_fenicsx_convection_dominated.py"),
]
hemker_path = next(path for path in hemker_path_candidates if path.exists())

spec = importlib.util.spec_from_file_location("hemker_convection_dominated", hemker_path)
hemker = importlib.util.module_from_spec(spec)
sys.modules[spec.name] = hemker
spec.loader.exec_module(hemker)
```

```
hemker_eps_values = [1.0e-2, 1.0e-3, 1.0e-4, 1.0e-5]

# Practical lecture mesh: fine enough to show the trend, coarse enough to render quickly.
hemker_domain, hemker_facet_tags = hemker.create_hemker_mesh(lc_bulk=0.45/2, lc_near=0.10/2)
fdim = hemker_domain.topology.dim - 1
hemker_domain.topology.create_connectivity(fdim, hemker_domain.topology.dim)

hemker_samples = []
hemker_diagnostics = []
for eps_value in hemker_eps_values:
    results = [
        hemker.summarize(
            "CG plain",
            hemker.solve_cg(hemker_domain, hemker_facet_tags, eps_value, degree=1, supg=False)
        ),
        hemker.summarize(
            "CG + SUPG",
            hemker.solve_cg(hemker_domain, hemker_facet_tags, eps_value, degree=1, supg=True),
        ),
        hemker.summarize(
            "DG upwind",
            hemker.solve_dg_upwind(
                hemker_domain, hemker_facet_tags, eps_value, degree=1, penalty=20.0
            ),
        ),
    ]

    for result in results:
        hemker_diagnostics.append((eps_value, result))

    sample = hemker.sample_field_grid(results, nx=260, ny=130)
    if MPI.COMM_WORLD.rank == 0:
        hemker_samples.append((eps_value, sample))

if MPI.COMM_WORLD.rank == 0:
    print(f"{'eps':>9} {'method':<12} {'min(u)':>11} {'max(u)':>11} {'L2 violation':>14}")
    print("-" * 64)
    for eps_value, result in hemker_diagnostics:
        print(
            f"{eps_value:9.0e} {result.name:<12} "
```

```
            f"{result.min_u:11.3e} {result.max_u:11.3e} {result.l2_violation:14.3e}"
        )
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 5 (Ellipse)
Info    : [ 30%] Meshing curve 6 (Line)
Info    : [ 50%] Meshing curve 7 (Line)
Info    : [ 70%] Meshing curve 8 (Line)
Info    : [ 90%] Meshing curve 9 (Line)
Info    : Done meshing 1D (Wall 0.00767071s, CPU 0.006085s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.0717747s, CPU 0.070261s)
Info    : 4961 nodes 9927 elements
      eps method            min(u)      max(u)   L2 violation
-----------------------------------------------------------------
    1e-02 CG plain      -3.491e-01   1.001e+00      4.342e-02
    1e-02 CG + SUPG     -1.387e-03   1.001e+00      1.151e-03
    1e-02 DG upwind     -3.455e-01   1.006e+00      1.759e-02
    1e-03 CG plain      -1.756e+00   1.874e+00      3.543e-01
    1e-03 CG + SUPG     -3.643e-01   1.053e+00      9.071e-02
    1e-03 DG upwind     -4.443e-01   1.089e+00      5.397e-02
    1e-04 CG plain      -4.314e+00   5.062e+00      1.838e+00
    1e-04 CG + SUPG     -6.046e-01   1.083e+00      1.435e-01
    1e-04 DG upwind     -2.536e-01   1.167e+00      8.689e-02
    1e-05 CG plain      -7.108e+00   8.319e+00      3.260e+00
    1e-05 CG + SUPG     -6.508e-01   1.087e+00      1.507e-01
    1e-05 DG upwind     -2.425e-01   1.179e+00      9.248e-02
```

```
if MPI.COMM_WORLD.rank == 0:
    import matplotlib.patches as patches

    method_names = hemker_samples[0][1][0]
    nrows = len(hemker_samples)
    ncols = len(method_names)
    fig, axes = plt.subplots(
        nrows,
        ncols,
        figsize=(12.8, 8.4),
        constrained_layout=True,
        sharex=True,
        sharey=True,
    )

    for row, (eps_value, sample) in enumerate(hemker_samples):
        names, _X, _Y, values = sample
        for col, (name, vals) in enumerate(zip(names, values)):
            ax = axes[row, col]
            im = ax.imshow(
                vals,
```

```
                origin="lower",
                extent=(-3.0, 9.0, -3.0, 3.0),
                cmap="viridis",
                vmin=-0.2,
                vmax=1.2,
                interpolation="nearest",
            )
            ax.add_patch(patches.Circle((0.0, 0.0), 1.0, color="white", zorder=5))
            ax.add_patch(
                patches.Circle((0.0, 0.0), 1.0, fill=False, edgecolor="black", linewidth=0.8,
            )
            if row == 0:
                ax.set_title(name)
            if col == 0:
                ax.set_ylabel(f"eps={eps_value:.0e}\ny")
            if row == nrows - 1:
                ax.set_xlabel("x")
            ax.set_aspect("equal")

    fig.colorbar(im, ax=axes, shrink=0.72, label="u_h, clipped color scale [-0.2, 1.2]")
    plt.show()
```

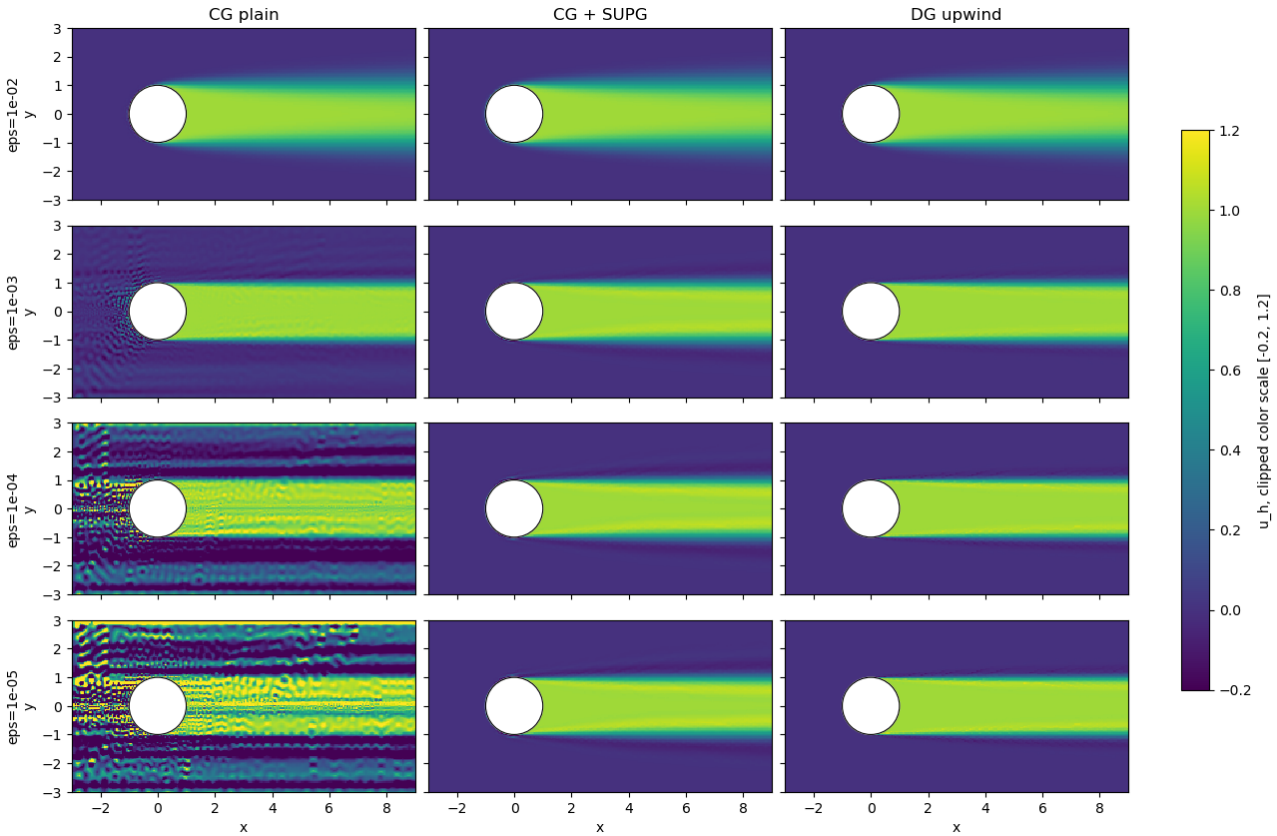


Figure 12: Hemker convection-dominated benchmark for decreasing diffusion.

> **Reading the Hemker plots**
>
> The color scale is intentionally clipped to `[-0.2, 1.2]`. This keeps the physical range visible while still exposing undershoots and overshoots. The diagnostic table gives the actual extrema, which can be much larger for unstabilized CG when ε is small.

5.4 Mesh refinement and mesh-based diffusion

The previous figure varied the physical diffusion parameter on one practical lecture mesh. Now we keep the hard case fixed and only refine the mesh: the baseline Hemker mesh, half the element size, and quarter the element size.

This is the simple reason why the wake can still look diffusive even for tiny ε: the discretization only sees structures at the scale of the mesh. A useful local number is the cell Peclet number

$$\mathrm{Pe}_h = \frac{\|\beta\| h}{2\varepsilon}.$$

When $\mathrm{Pe}_h \gg 1$, the physical layer is thinner than what the mesh can represent. Stable methods then add numerical diffusion, explicitly as in SUPG or implicitly through upwinding/finite resolution, typically of size comparable to $\|\beta\| h$. Halving h roughly halves this mesh-based diffusion; quartering h reduces it again. So making ε smaller without also reducing h does not necessarily make the plotted layer sharper.

```
hemker_mesh_eps = 1.0e-5
hemker_mesh_levels = [
    ("base h", 1.0, 0.45, 0.10),
    ("h/2", 0.5, 0.45 / 2.0, 0.10 / 2.0),
    ("h/4", 0.25, 0.45 / 4.0, 0.10 / 4.0),
]

hemker_mesh_samples = []
hemker_mesh_diagnostics = []
for level_name, h_factor, lc_bulk, lc_near in hemker_mesh_levels:
    domain_h, facet_tags_h = hemker.create_hemker_mesh(lc_bulk=lc_bulk, lc_near=lc_near)
    fdim = domain_h.topology.dim - 1
    domain_h.topology.create_connectivity(fdim, domain_h.topology.dim)

    n_cells = domain_h.topology.index_map(domain_h.topology.dim).size_local
    results_h = [
        hemker.summarize(
            "CG plain",
            hemker.solve_cg(domain_h, facet_tags_h, hemker_mesh_eps, degree=1, supg=False),
        ),
        hemker.summarize(
            "CG + SUPG",
            hemker.solve_cg(domain_h, facet_tags_h, hemker_mesh_eps, degree=1, supg=True),
        ),
        hemker.summarize(
            "DG upwind",
            hemker.solve_dg_upwind(
```

```
                domain_h, facet_tags_h, hemker_mesh_eps, degree=1, penalty=20.0
            ),
        ),
    ]

    for result in results_h:
        hemker_mesh_diagnostics.append((level_name, h_factor, n_cells, result))

    sample_h = hemker.sample_field_grid(results_h, nx=260, ny=130)
    if MPI.COMM_WORLD.rank == 0:
        hemker_mesh_samples.append((level_name, h_factor, n_cells, sample_h))

if MPI.COMM_WORLD.rank == 0:
    print(f"eps = {hemker_mesh_eps:.0e}")
    print(f"{'mesh':<8} {'cells':>8} {'Pe_h near':>10} {'method':<12} {'min(u)':>11} {'max(u)'
    print("-" * 86)
    for level_name, h_factor, n_cells, result in hemker_mesh_diagnostics:
        pe_near = h_factor * 0.10 / (2.0 * hemker_mesh_eps)
        print(
            f"{level_name:<8} {n_cells:8d} {pe_near:10.1f} {result.name:<12} "
            f"{result.min_u:11.3e} {result.max_u:11.3e} {result.l2_violation:14.3e}"
        )
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 5 (Ellipse)
Info    : [ 30%] Meshing curve 6 (Line)
Info    : [ 50%] Meshing curve 7 (Line)
Info    : [ 70%] Meshing curve 8 (Line)
Info    : [ 90%] Meshing curve 9 (Line)
Info    : Done meshing 1D (Wall 0.00733163s, CPU 0.006847s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.0239842s, CPU 0.022573s)
Info    : 1379 nodes 2763 elements
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 5 (Ellipse)
Info    : [ 30%] Meshing curve 6 (Line)
Info    : [ 50%] Meshing curve 7 (Line)
Info    : [ 70%] Meshing curve 8 (Line)
Info    : [ 90%] Meshing curve 9 (Line)
Info    : Done meshing 1D (Wall 0.00529167s, CPU 0.005286s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.0664959s, CPU 0.065841s)
Info    : 4961 nodes 9927 elements
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 5 (Ellipse)
Info    : [ 30%] Meshing curve 6 (Line)
Info    : [ 50%] Meshing curve 7 (Line)
Info    : [ 70%] Meshing curve 8 (Line)
Info    : [ 90%] Meshing curve 9 (Line)
```

```
Info    : Done meshing 1D (Wall 0.00698617s, CPU 0.006971s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.317238s, CPU 0.310865s)
Info    : 18815 nodes 37635 elements
eps = 1e-05
mesh        cells  Pe_h near method            min(u)      max(u)   L2 violation
-------------------------------------------------------------------------------------
base h       2603     5000.0 CG plain      -2.598e+01   2.431e+01      9.329e+00
base h       2603     5000.0 CG + SUPG     -5.498e-01   1.146e+00      2.042e-01
base h       2603     5000.0 DG upwind     -2.403e-01   1.270e+00      1.248e-01
h/2          9622     2500.0 CG plain      -7.108e+00   8.319e+00      3.260e+00
h/2          9622     2500.0 CG + SUPG     -6.508e-01   1.087e+00      1.507e-01
h/2          9622     2500.0 DG upwind     -2.425e-01   1.179e+00      9.248e-02
h/4         37032     1250.0 CG plain      -1.201e+01   1.227e+01      3.345e+00
h/4         37032     1250.0 CG + SUPG     -5.916e-01   1.076e+00      1.052e-01
h/4         37032     1250.0 DG upwind     -2.187e-01   1.189e+00      7.120e-02
```

```python
if MPI.COMM_WORLD.rank == 0:
    import matplotlib.patches as patches

    method_names = hemker_mesh_samples[0][3][0]
    nrows = len(hemker_mesh_samples)
    ncols = len(method_names)
    fig, axes = plt.subplots(
        nrows,
        ncols,
        figsize=(12.8, 6.6),
        constrained_layout=True,
        sharex=True,
        sharey=True,
    )

    for row, (level_name, h_factor, n_cells, sample) in enumerate(hemker_mesh_samples):
        names, _X, _Y, values = sample
        for col, (name, vals) in enumerate(zip(names, values)):
            ax = axes[row, col]
            im = ax.imshow(
                vals,
                origin="lower",
                extent=(-3.0, 9.0, -3.0, 3.0),
                cmap="viridis",
                vmin=-0.2,
                vmax=1.2,
                interpolation="nearest",
            )
            ax.add_patch(patches.Circle((0.0, 0.0), 1.0, color="white", zorder=5))
            ax.add_patch(
                patches.Circle((0.0, 0.0), 1.0, fill=False, edgecolor="black", linewidth=0.8,
            )
            if row == 0:
```

```
                ax.set_title(name)
            if col == 0:
                ax.set_ylabel(f"{level_name}\n{n_cells} cells\ny")
            if row == nrows - 1:
                ax.set_xlabel("x")
            ax.set_aspect("equal")

    fig.colorbar(im, ax=axes, shrink=0.9, label="u")
    plt.show()
```

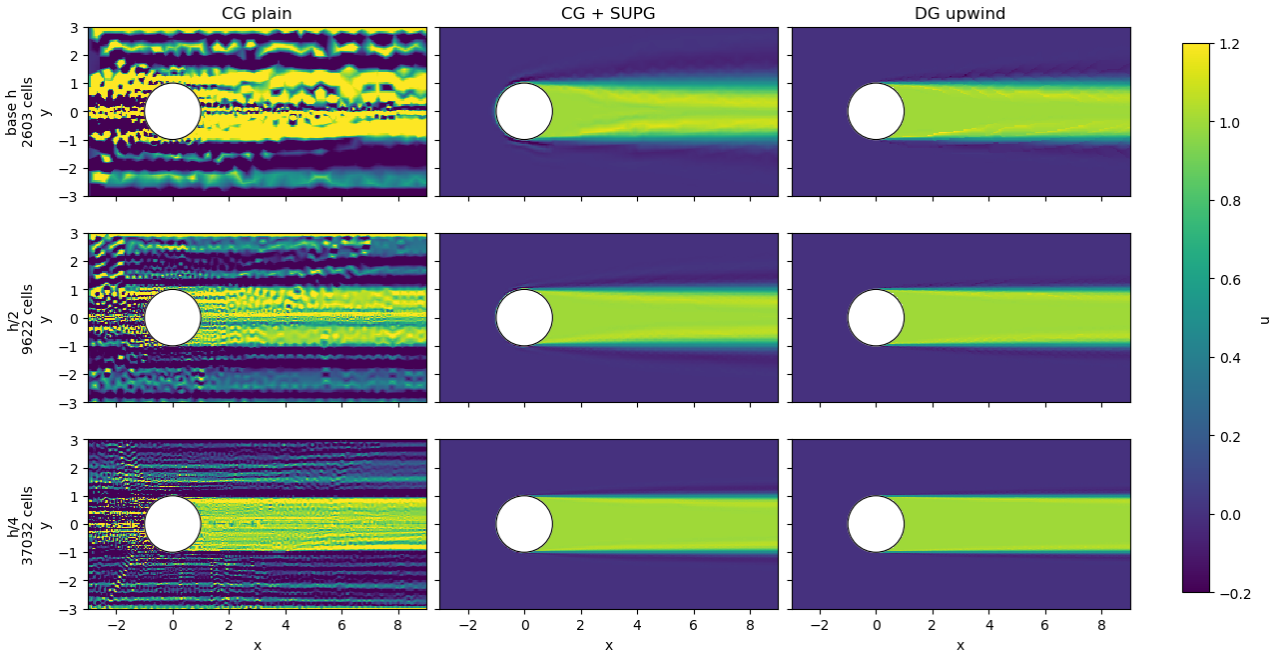


Figure 13: Hemker benchmark under mesh refinement.

5.5 Other interesting examples

- Navier-Stokes: flow past a cylinder
- Many examples for computational mechanics (Jeremy Bleyer)
- Adaptive mesh refinining (Jørgen S. Dokken)

6 References

- PETSc manual page: PCFIELDSPLIT
- PETSc manual page: Schur preconditioners for field splits
- PETSc guide to the Stokes equations
- Firedrake saddle-point systems demo
- Firedrake Stokes equations demo
- Firedrake solver interface and field-split discussion

[1] R. Farber, "PETSc/TAO: How to create, maintain, and modernize a numerical toolkit throughout decades of supercomputer innovations." Exascale Computing Project, Nov. 18, 2022. Accessed: May 06, 2026. [Online]. Available: https://www.exascaleproject.org/highlight/petsc-tao-how-to-create-maintain-and-modernize-a-numerical-toolkit-throughout-decades-of-supercomputer-innovations/

[2] P. Jolivet, "Efficient preconditioning with PETSc and petsc4py." Mar. 2026. Accessed: May 06, 2026. [Online]. Available: https://joliv.et/petsc-tutorial/main.pdf

[3] P. W. Hemker, "A singularly perturbed model problem for numerical computation," *Journal of Computational and Applied Mathematics*, vol. 76, no. 1–2, pp. 277–285, Dec. 1996, doi: 10.1016/S0377-0427(96)00113-6.

[4] A. Jha, "Numerical Algorithms for Algebraic Stabilizations of Scalar Convection-Dominated Problems," PhD thesis, 2020.