Modern Simulation Software Development

Lectures, Exercises, and Skills

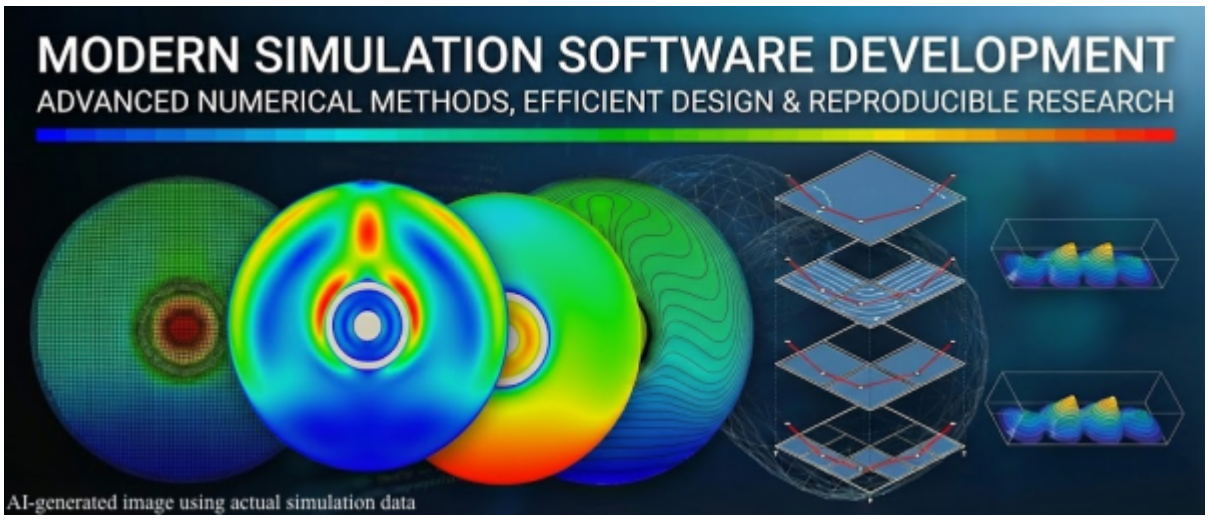
Dr. Lambert Theisen Dr. Georgii Oblapenko

2026-05-07

Table of contents

1. Modern Simulation Software Development

Summer 2026 · ACoM · RWTH Aachen University



A course for graduate students (e.g., master program in CES, SiSc, Math, Physics, etc.) on development of modern numerical software for simulation of complex engineering problems.

Lecturers: Dr. Lambert Theisen & Dr. Georgii Oblapenko

ECTS: 5 · **Format:** 2SWS lecture + 1SWS tutorial (tutorials merged to 2SWS slots every other week, 1SWS=45mins/week)

Registration: Opens 11 March 2026 via RWTHOnline (course 11.00153)

Module: *Current Topics in Computational Science and Engineering* (*Aktuelle Themen in Computational Science and Engineering*)

1.1. Goals & Topics

Advanced numerical methods for solving PDEs are presented and discussed, along with aspects of efficient numerical software design, focusing on robustness, accuracy in large-scale simulations, and reproducible research. Students gain insight into how modern simulation software is designed, implemented, and validated for real-world scientific and engineering applications.

Topics include (among others):

- Finite difference methods
- Finite element and discontinuous Galerkin methods
- Iterative solvers
- Particle-based methods

- Introduction to uncertainty quantification

1.2. Lectures

#	Topic	Notes	Slides	Date
1	Introduction	Notes	Slides	15.4.2026
2	Introduction to the Finite Element Method	Notes	Slides	22.4.2026
3	FEM II: nonlinear Poisson, Stokes with heat transport, eigenvalue problems	Notes	Slides · Notebook	29.4.2026
4	FEMI: Time, Preconditioners, PETSc	Notes	Slides	6.5.2026
5	Timestepping methods	Notes	Slides · Notebook	13.5.2026
6	Finite Volume Methods	-	-	20.5.2026
7	DG Methods: Introduction	-	-	3.6.2026
8	DG Methods: DGSEM, Entropy-stable methods	-	-	10.6.2026
9	Particle-based methods: SPH, Vortex methods	-	-	17.6.2026
10	Particle-based methods: DSMC	-	-	24.6.2026
11	Guest lecture	-	-	1.7.2026
12	Uncertainty quantification, sensitivity analysis	-	-	8.7.2026

1.3. Exercises

#	Topic	Topics	Date
1	Introduction (Shell, Git, Finite Difference, Repo Template)	Shell, Git, Finite Difference and Repo Structure	21.4.2026

#	Topic	Topics	Date
2	FEniCSx/Gmsh project preparation, post-processing	Gmsh geometry and boundary tags, FEM, Paraview	05.05.2026

1.4. Projects

#	Topic	Sheet	Deadline	Submission
0	Finite Differences	Project 00 (example)	-	21.4.2026
1	Open FEM Project with FEniCSx	Notes · Slides · PDF · Notebook	01.06.2026 (end of date)	Via Moodle as Link to repo and PDF

1.5. Skills

#	Topic	Notes
1	Bash for Simulation Workflows	Skills 01
2	Git for Simulation Software Development	Skills 02
3	Gmsh Geometry for FEM	Skills 03
4	Paraview for FEM Output	Skills 04

1.6. Local Installation and Execution

The local workflow below uses the FEniCSx/DOLFINx stack used by Lecture 03 and Project 01: `dolfinx 0.10.x`, PETSc, Gmsh, Jupyter, and Quarto.

1.6.1. 1. Create local conda environment

```
conda create -n fenicsx010 --override-channels -c conda-forge -y \
  python=3.12 fenics-dolfinx=0.10.0 ipykernel pyyaml matplotlib gmsh
```

If the conda solve does not provide the Python `gmsh` or `pyvista[jupyter]` module on your platform, install it with pip inside the environment:

```
conda run -n fenicsx010 python -m pip install --no-cache-dir gmsh pyvista[jupyter]
```

1.6.2. 2. Register Jupyter kernel for Quarto

Quarto notebooks in this repo use `jupyter: fenicsx`, so register that kernel name:

```
conda run -n fenicsx010 python -m ipykernel install --user \
  --name fenicsx --display-name "Python (fenicsx 0.10.0)"
```

1.6.3. 3. Install Quarto (if needed)

Check if Quarto is available:

```
quarto --version
```

If not installed, follow the official instructions: - https://quarto.org/docs/get-started/

1.6.4. 4. Render the site locally

```
quarto render
```

Output is written to: - `_site/index.html`

The site render includes Quarto notebooks such as:

- `lectures/03.ipynb`, producing notes, RevealJS slides, PDF, and a copied notebook resource.
- `exercises/01-fem.ipynb`, producing `01-fem-notes.html`, `01-fem-slides.html`, `01-fem.pdf`, and a copied notebook resource.

To render only the current FEM project sheet:

```
quarto render exercises/01-fem.ipynb
```

1.6.5. 5. Render the combined manuscript PDF

To build the book-like combined PDF with syllabus, lecture notes, exercises, and skills:

```
quarto render --profile manuscript --to pdf
```

Output is written to: - `manuscript/_book/mssd.pdf`

1.6.6. 6. Quick sanity checks

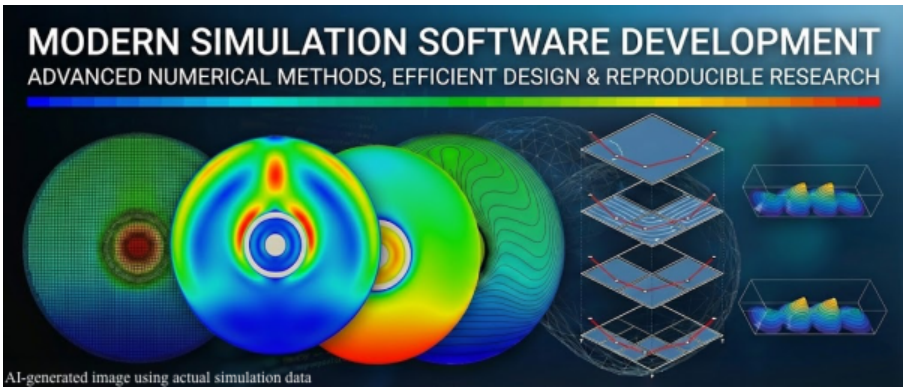
```
conda run -n fenicsx010 python -c "import dolfinx, gmsh, petsc4py, mpi4py; print(dolfinx.__ver
```

You should see DOLFINx `0.10.0`.

1.7. Links

- ACoM Website
- RWTHOnline Course Page
- Moodle Page

2. Preface



This manuscript compiles the current **Modern Simulation Software Development** course material into one book-like PDF. It includes:

- the syllabus,
- lecture notes,
- exercise sheets,
- practical skill notes.

The manuscript is generated from the same Quarto sources used for the course website so that the PDF and the web version stay aligned.

> **Note**
>
> This compilation intentionally uses the **lecture notes** rather than the slide decks. The goal is a readable manuscript, not a slide transcript.

> **Important**
>
> Some chapters contain executable code examples. The manuscript is designed as a consolidated reference document; if you want to rerun the computational material, use the repository workflows and environment described in the course documentation.

3. Syllabus

Modern Simulation Software Development · Summer 2026

3.1. Course Information

Lecturers	Dr. Lambert Theisen, Dr. Georgii Oblapenko
ECTS	5
Format	2h lecture + 1h tutorial/week (tutorials merged to 2h biweekly)
Examination	Oral exam or project-based evaluation (TBD)
Module	Current Topics in Computational Science and Engineering
Registration	RWTHOnline (course 11.00153) – opens 11 March 2026

3.2. Schedule

Week	Topic
1	Finite Difference Methods
2	Stability & Convergence
3–4	Finite Element Methods
5–6	Discontinuous Galerkin Methods
7–8	Iterative Solvers
9–10	Particle-Based Methods
11–12	Uncertainty Quantification
13	Advanced Topics & Review

Schedule is tentative and subject to change.

3.3. Background Literature

- D. A. Kopriva (2009). *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers.* Springer.
- A. Logg, K. A. Mardal & G. Wells, Eds. (2012). *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book.* Springer.

- R. C. Smith (2024). *Uncertainty Quantification: Theory, Implementation, and Applications.* SIAM.
- I. D. Boyd & T. E. Schwartzentruber (2017). *Nonequilibrium Gas Dynamics and Molecular Simulation.* Cambridge University Press.

3.4. Links

- ACoM Website
- Course Announcement
- Moodle Page

Part I.

Lectures

4. Lecture 01 · Introduction to Modern Simulation Software Development

Detailed Notes

4.1. Motivation

- **Complexity of Modern Simulation Software**: Developing industry-ready computational science and engineering (CSE) software, such as FEM-based tools, requires handling realistic geometries, multiphysics, efficient solvers, adaptive meshes, and scalability. Building such software from scratch would require approximately 100k+ lines of code, highlighting the need for collaborative and modular development approaches.
- **Challenges in Academic Software Development**: Research software in academia is often developed by graduate students and maintained by postdocs, who may lack software engineering experience and have limited time. Faculty advisors, who also often lack software expertise, provide limited oversight. This results in software that is primarily viewed as a tool for publishing rather than a sustainable, high-quality product.
- **Industry vs. Academia Workflows**: Industry prioritizes ready-made solutions, integration with existing tools, and robustness, while academia focuses on experimentation and publishing new methods. Bridging this gap requires software that is both flexible for research and reliable for industrial applications.
- **Sustainability and Collaboration**: Collaboration, modular design, and community involvement are essential to creating software that can evolve and remain relevant over decades. Collaboration is only possible when code is well-structured, testable, documented.

4.2. Course Goals

The course aims to introduce students to modern numerical methods and software development practices used in cutting-edge research. Key objectives include:

- Understanding advanced numerical methods for solving PDEs (e.g., finite differences, finite elements, discontinuous Galerkin, particle-based methods).
- Learning modern software development practices such as version control, testing, and continuous integration.
- Gaining hands-on experience with open-source frameworks like FEniCSx and Trixi.jl for setting up numerical simulations.
- Exploring applications in thermal transfer, fluid mechanics, gas dynamics, and uncertainty quantification.

4.3. Lecturers

In the summer semester of 2026, the course is taught by:

- **Dr. Lambert Theisen** (PhD 2024): Specializes in finite element methods (FEM), domain decomposition methods, linear system preconditioning.
- **Dr. Georgii Oblapenko** (PhD 2017): Focuses on models for multi-species reacting flows, finite volume methods (FVM), discontinuous Galerkin (DG) methods, particle methods for rarefied flows, and uncertainty quantification (UQ) for supersonic flows.

4.4. Organizational Details

- **Lectures**: Held every Wednesday from 10:30 to 12:00.
- **Exercises**: Schedule to be determined (TBD).
- **Projects**: Three projects will be assigned throughout the semester to apply the concepts learned in lectures.

4.5. What is "Modern" Simulation Software Development?

Modern simulation software development leverages advanced tools and methodologies to streamline the creation, testing, and deployment of simulation software. This includes:

- **Verifiable code and reproducible research**: Extensive testing and verification of code, versioning of code to ensure correctness and reproducibility of results
- **Automated workflows**: Integrating continuous integration and deployment (CI/CD) pipelines to ensure code quality and reproducibility.
- **Collaborative tools**: Using version control systems (e.g., Git) and platforms like GitHub/GitLab to facilitate teamwork and code sharing.

4.6. Capabilities of Modern Simulation Software

Modern simulation software is characterized by its ability to handle complex and diverse engineering problems. Key capabilities include:

- **Multi-physics**: Simulating interactions between different physical phenomena (e.g., fluid-structure interaction, thermo-mechanical coupling, phase transitions, etc.).
- **High-order methods**: Using high-order numerical schemes to achieve greater accuracy and efficiency.
- **Hybrid methods**: Combining different numerical methods (e.g., FEM with particle methods) to facilitate multi-physics simulations.
- **Uncertainty quantification**: Incorporating methods to assess and manage uncertainties in simulation inputs and outputs, provide safety tolerances, assess impact of parameter uncertainties on solution quality.

4.7. Software Aspects

Modern simulation software emphasizes robust and maintainable code development practices:

- **Unit tests**: Testing individual components of the code to ensure they function correctly in isolation.
- **Reproducible research**: Ensuring that simulations can be replicated and verified by others.
- **Modular design**: Structuring code into reusable and interchangeable modules.
- **Exascale-ready**: Efficiently supporting parallel computing technologies like MPI and GPU acceleration.
- **Mixed precision**: Utilizing different levels of numerical precision to optimize performance and accuracy.
- **Automated code generation**: Using tools for automatic differentiation (AD) and code generation to reduce manual coding errors.

4.8. Verification and Validation

- **Verification**: Ensures that the equations are solved correctly. This involves both mathematical correctness (e.g., using appropriate numerical methods) and programming correctness (e.g., implementing the methods accurately).
- **Validation**: Ensures that the correct equations are solved, addressing the physical and engineering relevance of the model. This requires experimental data, a ground truth model, or a combination thereof!

4.9. Code Verification

Code verification involves several levels of testing to ensure the reliability and correctness of the software:

4.9.1. Unit Testing

Tests individual units of code (e.g., functions, classes) to verify their correctness. Example:

```
E0 = energy(particles)  # assumes energy has already been tested
collide_2particles!(particles, collision_parameters, i1, i2)
@test isapprox(energy(particles), E0; atol=eps(), rtol=eps()) == true
```

4.9.2. Integration Testing

Tests the interaction between different units to ensure they work together as intended. Example:

```
E0 = energy(particles)  # assumes energy has already been tested
collide_all_particles!(particles, collision_parameters)
@test isapprox(energy(particles), E0; atol=eps(), rtol=eps()) == true
```

4.9.3. Regression Testing

Ensures that existing functionality continues to work as expected after changes or updates. Example:

```
heat_flux = simulation_result()
reference_heat_flux = parse(Float64, read("reference_q.txt", String))
@test isapprox(heat_flux, reference_heat_flux; atol=eps(), rtol=eps()) == true
```

4.10. Continuous Integration (CI): example 1 (Github Actions)

CI involves automating the build and test processes to ensure that code changes do not introduce errors. CI pipelines are typically integrated with version control systems and run automatically when code is committed. Example CI setup using Github Actions (from the Merzbild.jl code of Georgii Oblapenko):

```
name: Tests

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

# needed to allow julia-actions/cache to delete old caches that it has created
permissions:
  actions: write
  contents: read

jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        julia-version: ['lts', '1', 'pre']
        julia-arch: [x64, x86]
        os: [ubuntu-latest, windows-latest]

    steps:
      - uses: actions/checkout@v4
      - uses: julia-actions/setup-julia@v2
        with:
          version: ${{ matrix.julia-version }}
          arch: ${{ matrix.julia-arch }}
      - uses: julia-actions/cache@v2
      - uses: julia-actions/julia-buildpkg@v1
      - uses: julia-actions/julia-runtest@v1
      - uses: julia-actions/julia-processcoverage@v1
      - name: Upload results to Codecov
```

```
        uses: codecov/codecov-action@v5
        with:
            fail_ci_if_error: true
            token: ${{ secrets.CODECOV_TOKEN }}
```

This sets up to run tests on 2 operating systems (Ubuntu and Windows) using 3 Julia versions (LTS, latest stable, and pre-release) compiled for 2 architectures (x64 and x86). The tests are run and code coverage statistics are uploaded to codecov.io. The tests are run every time a pull request to the `main` branch receives a commit, or the `main` branch receives a commit directly. So a pull request where tests fail (code produces errors or new code is not covered by tests) will display a warning.

4.11. Continuous Deployment (CD)

CD automates the release process, ensuring that new features and bug fixes are deployed reliably and frequently. CD can include:

- Automatic deployment of documentation websites (documentation a mix of docstrings in code and additional documentation).
- Analysis of code coverage to ensure comprehensive testing (see above).
- Archiving releases to platforms like Zenodo for persistent DOIs.

The effectiveness of CD depends on the quality and coverage of the tests.

Example of documentation CD from Merzbild.jl:

```
name: Documentation

on:
  push:
    branches:
      - main # update to match your development branch (master, main, dev, trunk, ...)
    tags: '*'
  pull_request:
    branches:
      - main

jobs:
  build:
    permissions:
      actions: write
      contents: write
      pull-requests: read
      statuses: write
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: julia-actions/setup-julia@v2
        with:
          version: '1'
      - uses: julia-actions/cache@v2
```

```
      - name: Install dependencies
        run: julia --project=docs/ -e 'using Pkg; Pkg.develop(PackageSpec(path=pwd())); Pkg.in
      - name: Build and deploy
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # If authenticating with GitHub Actions to
        run: julia --project=docs/ docs/make.jl
```

4.12. Continuous Integration (CI/CD): example 2 (Gitlab)

Example CI/CD setup using Gitlab (adapted from the fenicsR13 code of Lambert Theisen), this both tests the code and builds and deploys documentation:

```
variables:
  APP_DIRECTORY: .
  DOCS_DIRECTORY: ${APP_DIRECTORY}/docs
  DOCS_LATEX_NAME: fenicsr13
stages:
- prepare
- build
- test
- deploy

# ****************************************************************************** #
# prepare
# ****************************************************************************** #

prepare:docker:
  stage: prepare
  image: docker:26.1.4
  services:
    - docker:26.1.4-dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker pull $CI_REGISTRY_IMAGE:latest || true
    - docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA || true
    - docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME || true
    - docker build
      --cache-from $CI_REGISTRY_IMAGE:latest
      --tag $CI_REGISTRY_IMAGE:latest
      --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME .
    - docker push $CI_REGISTRY_IMAGE:latest
  only:
    - master
    - tags
  tags:
    - dockerindocker
```

```
# ****************************************************************************** #
# build
# ****************************************************************************** #

build:docs:
  stage: build
  dependencies:
    - prepare:docker
  image:
    name: $CI_REGISTRY_IMAGE:latest
    entrypoint: [""]
  script:
    - cd ${DOCS_DIRECTORY}
    - sphinx-apidoc -f -o source/fenicsR13 ../fenicsR13
    - sphinx-apidoc -f -o source/tests/2d_heat ../tests/2d_heat
    - sphinx-apidoc -f -o source/tests/2d_stress ../tests/2d_stress
    - sphinx-apidoc -f -o source/tests/2d_r13 ../tests/2d_r13
    - sphinx-apidoc -f -o source/tests/3d_heat ../tests/3d_heat
    - sphinx-apidoc -f -o source/tests/3d_stress ../tests/3d_stress
    - sphinx-apidoc -f -o source/tests/3d_r13 ../tests/3d_r13
    - sphinx-apidoc -f -o source/examples ../examples
    - make html
    - make latex
  artifacts:
    paths:
      - ${DOCS_DIRECTORY}/_build/html/
      - ${DOCS_DIRECTORY}/_build/latex/
    expire_in: 6 month
  tags:
    - docker

# ****************************************************************************** #
# test
# ****************************************************************************** #

.test: # dot means "hidden", acts as base class
  stage: test
  dependencies:
    - prepare:docker
  before_script:
    - pip install -e . # local install to have right coverage
  image:
    name: $CI_REGISTRY_IMAGE:latest
    entrypoint: [""]
  tags:
    - docker

test:flake8:
  extends: .test
  script:
```

```
    - cd ${APP_DIRECTORY}
    - flake8 fenicsR13 examples tests

test:pytest:
  extends: .test
  script:
    - cd ${APP_DIRECTORY}
    - pytest -v --durations=10
  after_script:
    - mkdir convergence_plots
    - cp tests/*/*/convergence_plot* convergence_plots/
    - mkdir errors
    - for file in tests/*/*/errors.csv; do foldername=${file%/errors.csv}; casename=$(basename
    # - for file in tests/*/*/errors.csv; do foldername=${file%/errors.csv}; casename=$(basena
  coverage: '/TOTAL.*? (100(?:\.0+)?\%|[1-9]?\d(?:\.\d+)?\%)$/'
  artifacts:
    when: always
    paths:
      - ${APP_DIRECTORY}/convergence_plots
      - ${APP_DIRECTORY}/errors
      - ${APP_DIRECTORY}/htmlcov
    expire_in: 6 month

# ************************************************************************** #
# deploy
# ************************************************************************** #

deploy:latex:
  stage: deploy
  image:
    # name: blang/latex:ctanfull # only TexLive 2017
    name: aergus/latex:latest
  dependencies:
    - build:docs
  script:
    - cd ${DOCS_DIRECTORY}/_build/latex/
    - make
  artifacts:
    paths:
      - ${DOCS_DIRECTORY}/_build/latex/${DOCS_LATEX_NAME}.pdf
    expire_in: 6 month
  only:
    - master
    - tags
  tags:
    - docker

pages:
  stage: deploy
  dependencies:
    - build:docs
```

```
script:
  - mv ${DOCS_DIRECTORY}/_build/html/ ${CI_PROJECT_DIR}/public/
artifacts:
  paths:
    - public
only:
  - master
  - tags
tags:
  - shell
```

4.13. Examples of Open-Source Codes by course lecturers

- **fenicsR13** (Lambert Theisen): A FEM code based on FEniCSx for solving the linear R13 equations. Uses GitLab CI for continuous integration.
- **Merzbild.jl** (Georgii Oblapenko): A DSMC code for variable-weight particles, implemented in Julia. Uses GitHub CI for continuous integration.

4.14. Examples of other codes that do it "right"

- Deal.II - highly scalable FEM code
- AMReX - block-structured meshes with adaptive mesh refinement
- t8code - hybrid meshes with adaptive mesh refinement
- DifferentialEquations.jl - library for ODE solvers
- Trixi.jl - DGSEM code for solving hyperbolic PDEs

4.15. Sustainable Computational Engineering

The course aligns with broader initiatives in sustainable computational engineering, such as the course "Sustainable Computational Engineering" (42.00019) from MBD. This course focuses on:

- Continuous Integration (CI)
- Research Data Management (RDM)
- Licensing and legal aspects of software development

4.16. Key Take-Aways

1. **Learning from Mistakes**: The second code you write is often the worst because you aim to avoid all the mistakes from your first attempt. Embrace iterative improvement.
2. **Testing and Documentation**: These should not be afterthoughts but integral parts of the development process.
3. **Modularization**: Avoid monolithic functions (e.g., a `solve!()` function with 10,000 lines). Break down the code into manageable, reusable modules.

4.17. References

- How do we make large opensource projects sustainable? (lessons learned from 25 years of deal.II)

5. Lecture 02 – Introduction to the Finite Element Method

… and some FEniCSx basics

5.1. Partial Differential Equations (PDEs)

Some PDE examples in a table:

Example Equation	Remark
$-\nabla \cdot (k\nabla u) = f$, temperature $u : \Omega \to \mathbb{R}$	Steady-state heat conduction, elliptic
$\frac{\partial u}{\partial t} - \nabla \cdot (k\nabla u) = f$	Transient heat conduction, parabolic
$\frac{\partial^2 u}{\partial t^2} - \mathrm{c}^2\nabla^2 u = 0$	Wave propagation, hyperbolic
$-\nabla \cdot (k\nabla u) + R(u) = f$, concentration $c : \Omega \to \mathbb{R}$, $R(u) = u(1-u)$ (e.g.)	Reaction-diffusion

5.1.1. Notation

For simplicity, we wrote PDEs in vector form, but they can also be expressed in component, e.g.

$$-\nabla \cdot (k\nabla u) = -\partial_x(k\partial_x u) - \partial_y(k\partial_y u) - \partial_z(k\partial_z u).$$

Other helpful differential operators include:

- Gradient: $\nabla u = \left(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}\right)$
- Divergence: $\nabla \cdot \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}$ with $\mathbf{v} = (v_x, v_y, v_z)$
- Laplacian: $\Delta u = \nabla^2 u = \nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$
- Rotation: $\nabla \times \mathbf{v} = \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}, \frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}, \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right)$

5.1.2. Boundary-value-problems (BVPs)

- Domain Ω with boundary Γ, the domain Ω is where the PDE holds. (Mathematically, a domain is an open, non-empty, connected subset of $\mathbb{R}^n$ (e.g., a 2D region or 3D volume), and the boundary is its edge)
- $\Gamma = \partial\Omega$ is where we specify boundary conditions, n is the outward normal vector on Γ.

Boundary conditions:

- Dirichlet: $u = u_D$ on Γ_D (specify value of u)
- Neumann: $-k\nabla u \cdot n = g_N$ on Γ_N (specify flux across boundary)
- Robin: $\alpha u + \beta(-k\nabla u \cdot n) = g_R$ on Γ_R (combination of value and flux)

A BVP:

$$\begin{cases} -\nabla \cdot (k\nabla u) = f & \text{in } \Omega, \\ u = u_D & \text{on } \Gamma_D, \\ -k\nabla u \cdot n = g_N & \text{on } \Gamma_N. \end{cases}$$

5.1.3. Exact solution

For some simpler BVPs, we can find an exact solution analytically.

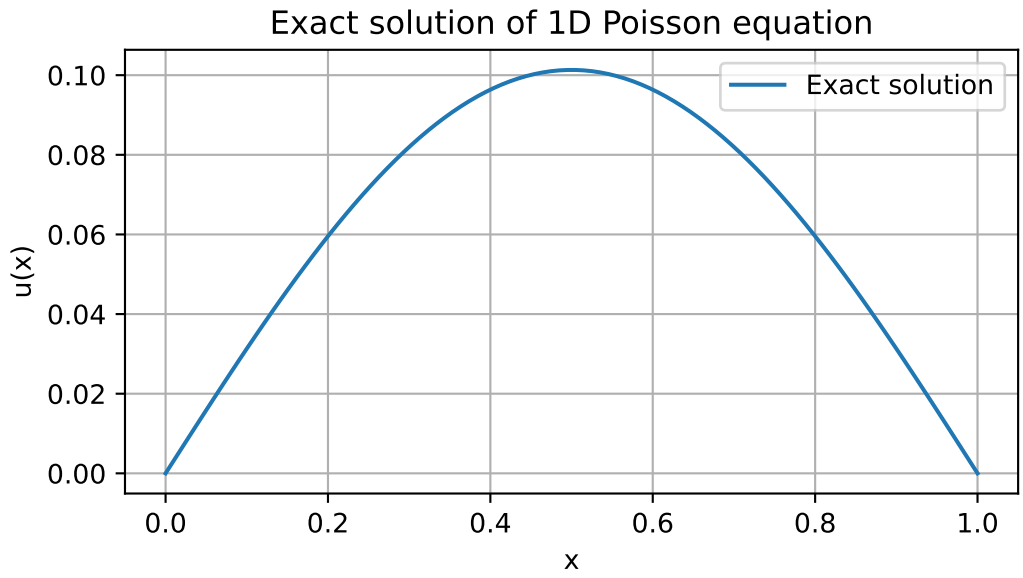
$$-\partial_x(k\partial_x u) = f(x) \quad \text{on } (0,1), \qquad u(0) = u(1) = 0.$$

with constant $k \in \mathbb{R}, k \neq 0$ and $f = \sin(\pi x)$ has exact solution:

$$u(x) = \tfrac{\sin(\pi x)}{k\pi^2}.$$

Since $\partial_x(k\partial_x u) = \partial_x\left(k\tfrac{\pi\cos(\pi x)}{k\pi^2}\right) = -\sin(\pi x)$, and the BCs are satisfied.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 1, 100)
k = 1.0
u = np.sin(np.pi * x) / (k * np.pi**2)
plt.plot(x, u, label='Exact solution')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title('Exact solution of 1D Poisson equation')
# change size
plt.gcf().set_size_inches(6, 3)
plt.grid(); plt.legend(); plt.show()
```



5.1.4. More complex domains

- For structured domains, the finite difference method (see exercise) can sometimes help.
- However, for more complex domains (e.g., with holes, curved boundaries), we need a more flexible method.

Finite element method (FEM): Can handle complex domains, unstructured meshes, and variable coefficients.

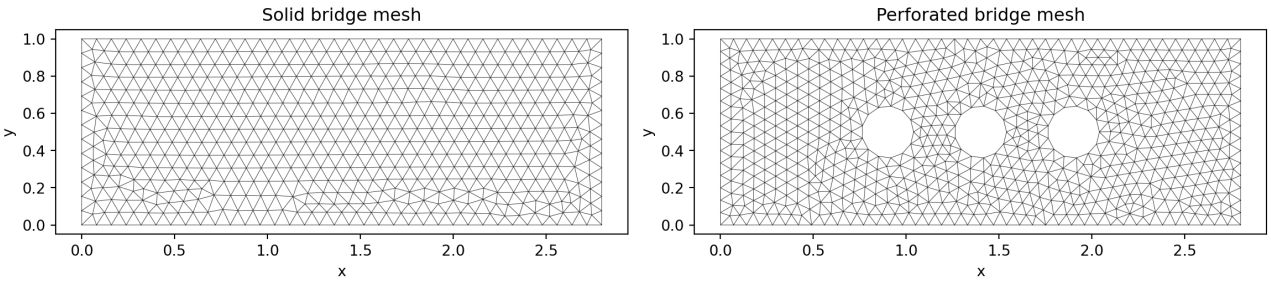


Figure 5.1.: Mesh example

> **Note**
>
> The starting point for FEM is a *weak* (variational) formulation of the PDE.

5.2. Weak forms

For a given PDE (strong form), e.g., the Poisson equation with mixed BCs:

$$-\nabla \cdot (k\nabla u) = f \quad \text{in } \Omega, \qquad u = u_D \text{ on } \Gamma_D, \qquad -k\nabla u \cdot n = g_N \text{ on } \Gamma_N$$

1. multiply by a test function v and integrate over Ω to get the weak form:

$$\int_\Omega -\nabla \cdot (k\nabla u)v \, dx = \int_\Omega fv \, dx.$$

2. Apply integration by parts to move derivatives from u to v:

$$\int_\Omega k\nabla u \cdot \nabla v \, dx - \int_{\Gamma_N} (k\nabla u \cdot n)\, v \, ds = \int_\Omega fv \, dx$$

3. Insertion of Neumann BC:

$$\int_\Omega k\nabla u \cdot \nabla v \, dx = \int_\Omega fv \, dx + \int_{\Gamma_N} g_N v \, ds.$$

5.2.1. Variational problem

- Define trial space V and test space V_0
  - Sobolev space $H^1(\Omega)$ include functions with square-integrable derivatives, i.e., $H^1(\Omega) = \{w \in L^2(\Omega) \,|\, \nabla w \in (L^2(\Omega))^d\}$ meaning w and its gradient are square-integrable, i.e., $\int_\Omega |w|^2 \, dx < \infty$ and $\int_\Omega |\nabla w|^2 \, dx < \infty$.

$$V = \{w \in H^1(\Omega) \,|\, w = u_D \text{ on } \Gamma_D\}, \qquad V_0 = \{w \in H^1(\Omega) \,|\, w = 0 \text{ on } \Gamma_D\}.$$

Resulting problem: Find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in V_0,$$

with a form $a : V \times V_0 \to \mathbb{R}$ and a linear form $L : V_0 \to \mathbb{R}$ defined as

$$a(u, v) = \int_\Omega k\nabla u \cdot \nabla v \, dx, \qquad L(v) = \int_\Omega fv \, dx + \int_{\Gamma_N} g_N v \, ds.$$

5.2.2. The energy viewpoint (homogeneous BCs, $k = 1$)

- Start from an energy functional: $E(u) = \frac{1}{2}\int_\Omega |\nabla u|^2 \, dx - \int_\Omega fu \, dx$.
- Gateaux derivative: $E'(u; v) = \lim_{\epsilon \to 0} \frac{E(u+\epsilon v) - E(u)}{\epsilon} = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_\Omega fv \, dx$.
- Energy minimization (calculus of variations): $\min_{u \in H_0^1(\Omega)} E(u)$ leads to $E'(u; v) = 0$ for all "direction" $v \in V_0$ as the first-order optimality condition, which is exactly the weak form.

$$\begin{aligned} E'(u; v) &= \lim_{\epsilon \to 0} \frac{E(u + \epsilon v) - E(u)}{\epsilon} \\ &= \lim_{\epsilon \to 0} \frac{1}{\epsilon} \left(\frac{1}{2} \int_\Omega \nabla(u + \epsilon v) \cdot \nabla(u + \epsilon v) \, dx - \int_\Omega f(u + \epsilon v) \, dx - \left(\int_\Omega |\nabla u|^2 \, dx - \int_\Omega fu \, dx \right) \right) \\ &= \int_\Omega \nabla u \cdot \nabla v \, dx - \int_\Omega fv \, dx \stackrel{!}{=} 0 \quad \forall v \in V_0. \end{aligned}$$

See [Brezzi & Fortin, 1991]

5.2.3. Implementation: Galerkin

- Start from a variational form: find $u \in V_0$ such that $a(u, v) = L(v) \forall v \in V_0$.
- Galerkin: choose $V_h \subset V_0$ and solve $a(u_h, v_h) = L(v_h) \quad \forall v_h \in V_h$.
- Therefore, $u_h = \sum_{j=1}^{N} U_j \phi_j$ with basis functions $\{\phi_j\}$ of V_h and unknown coefficients U_j.
- Inseration into the variational form and testing with basis ϕ_i leads to:

$$\sum_{j=1}^{N} U_j a(\phi_j, \phi_i) = L(\phi_i) \quad \forall i = 1, \dots, N,$$

5.2.4. Coefficient equation

Linear system of equations:

$$AU = b \quad \Leftrightarrow \quad \sum_{j=1}^{N} A_{ij} U_j = b_i \quad \forall i = 1, \dots, N,$$

with $A_{ij} = a(\phi_j, \phi_i) = \int_\Omega \nabla \phi_j \cdot \nabla \phi_i \, dx$ and $b_i = L(\phi_i) = \int_\Omega f \phi_i \, dx$.

P1 (linear) elements have 3 nodes per triangle (vertices)

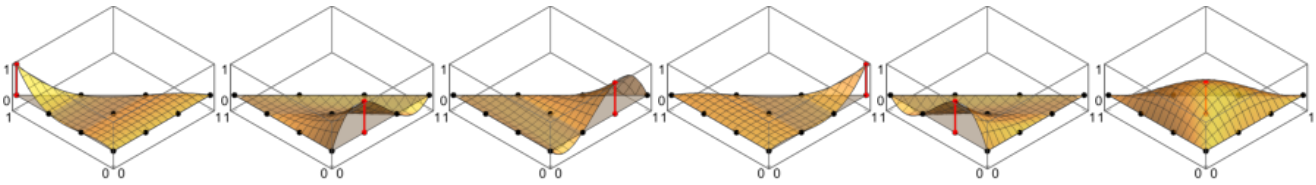
(a) P1 Lagrange 1

(a) P1 Lagrange 2

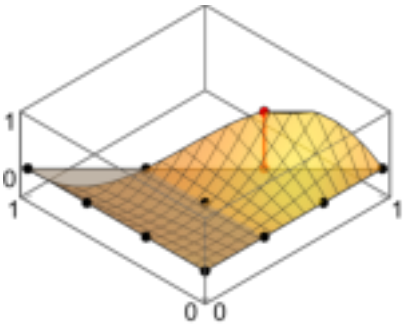
(a) P1 Lagrange 2

- Implementation can be annoying: See, e.g., these notes

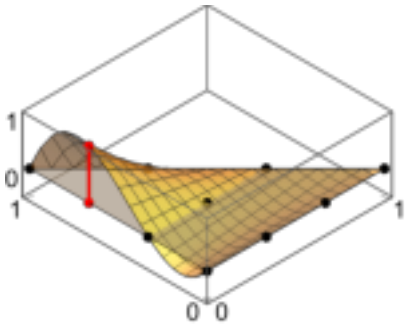
P3 Lagrange elements (cubic polynomials) have more nodes and higher-order basis functions, leading to better accuracy but also larger linear systems.



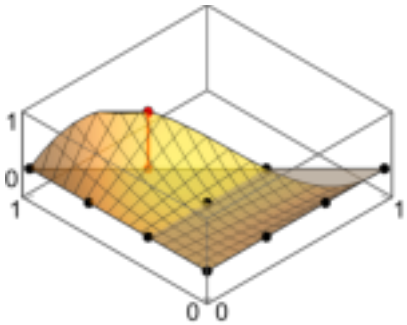
(a) P3 Lagrange 1 (a) P3 Lagrange 2 (a) P3 Lagrange 3 (a) P3 Lagrange 4 (a) P3 Lagrange 5 (a) P3 Lagrange 6



(a) P3 Lagrange 7



(a) P3 Lagrange 8



(a) P3 Lagrange 9

5.3. FEniCS framework

- FEniCS automates the whole process: from mesh generation, function space construction, variational form definition, assembly, to solving linear systems.
- You can focus on the model (weak form)
- Software Components (c.f. Lambert's M.Sc. thesis):
  - UFL: The Unified Form Language is a domain-specific language to formulate problems. It serves as the input for the form compiler.
  - FFCX: The FEniCS Form Compiler automatically generates DOLFIN code from a given variational form. The goal of a form compileris to use a well-tested compiler, performing automated code generation tasks, in order to improve the correctness of the resulting code.
  - BASIX/FIAT: The FInite element Automatic Tabulator enables the automatic computation of basis functions for nearly arbitrary finite elements.

5.3.1. Elements and forms

- FEniCS supports a wide range of finite elements (Lagrange, Raviart–Thomas, Nédélec, etc.), see BASIX
- UFL allows to express variational forms (linear, nonlinear, time-dependent), for example:

```
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
a = k * ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx
L = f * v * ufl.dx + g_N * v * ulf.ds
```

Other operators from UFL: `ufl.div`, `ufl.curl`, `ufl.cross`, `ufl.inner`, `ufl.outer`, etc.

Einstein summation convention is supported: `Dx(v, i)*Dx(w, i)` means $\sum_{i=1}^{d} \frac{\partial v}{\partial x_i} \frac{\partial w}{\partial x_i}$.

5.3.2. Algorithmic Workflow

1. Generate (or load) mesh $\mathcal{T}_h$ and boundary markers.
2. Build finite element space V_h.
3. Define variational forms $a(\cdot,\cdot), L(\cdot)$.
4. Assemble sparse matrix/vector.
5. Apply boundary conditions.
6. Solve with direct or iterative solver.
7. [Estimate error / refine mesh postprocess]

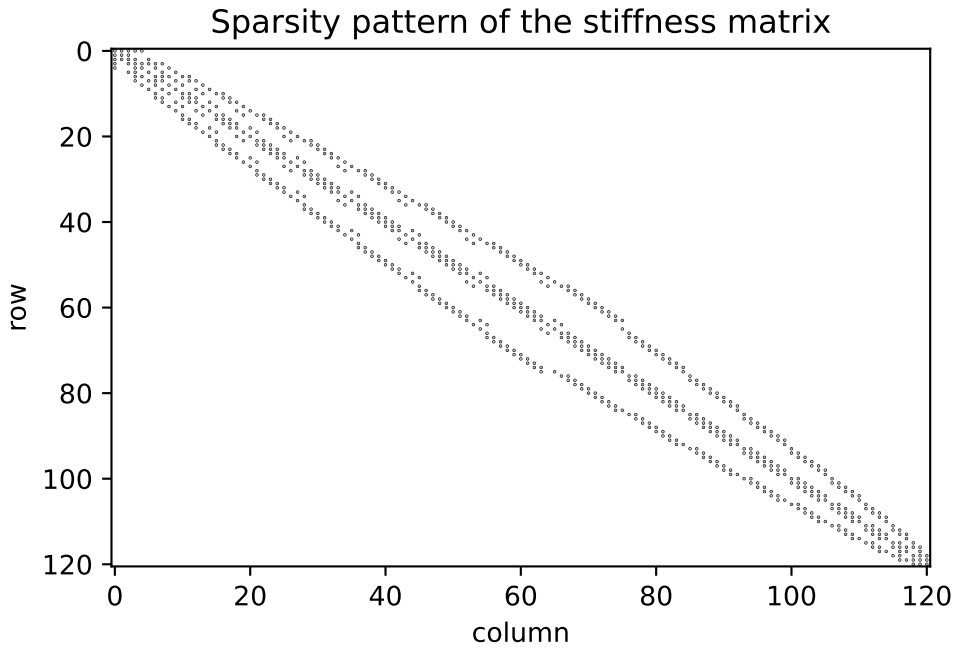
5.3.3. Minimal FEniCSx Example (Unit Square)

```
from mpi4py import MPI
from petsc4py import PETSc
import inspect
import numpy as np
import ufl
from dolfinx import fem, mesh
```

```python
from dolfinx.fem.petsc import LinearProblem

# Mesh and function space
msh = mesh.create_unit_square(MPI.COMM_WORLD, 10, 10)
V = fem.functionspace(msh, ("Lagrange", 1))

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

k = fem.Constant(msh, PETSc.ScalarType(1.0))
f = 10.0 * ufl.sin(ufl.pi * x[0]) * ufl.sin(ufl.pi * x[1])

a = k * ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx
L = f * v * ufl.dx

# Dirichlet boundary u=0 on full boundary
fdim = msh.topology.dim - 1
boundary_facets = mesh.locate_entities_boundary(
    msh,
    fdim,
    lambda x: np.isclose(x[0], 0.0)
    | np.isclose(x[0], 1.0)
    | np.isclose(x[1], 0.0)
    | np.isclose(x[1], 1.0),
)
dofs = fem.locate_dofs_topological(V, fdim, boundary_facets)
bc = fem.dirichletbc(PETSc.ScalarType(0.0), dofs, V)

linear_problem_kwargs = {"petsc_options": {"ksp_type": "preonly", "pc_type": "lu"}}
if "petsc_options_prefix" in inspect.signature(LinearProblem).parameters:
    linear_problem_kwargs["petsc_options_prefix"] = "poisson02_"

problem = LinearProblem(a, L, bcs=[bc], **linear_problem_kwargs)
uh = problem.solve()
uh.name = "temperature"
# show some terminal output
if MPI.COMM_WORLD.rank == 0:
    print("Assembled linear system:")
    print(f"  Matrix size: {problem.A.getSize()}")

if MPI.COMM_WORLD.size == 1:
    import matplotlib.pyplot as plt

    print("  Spy of matrix (nonzeros in black):")
    row_ptr, col_idx, _ = problem.A.getValuesCSR()
    row_idx = np.repeat(np.arange(len(row_ptr) - 1), np.diff(row_ptr))
    nrows, ncols = problem.A.getSize()

    fig, ax = plt.subplots()
    ax.plot(col_idx, row_idx, "k.", markersize=0.5)
```

```
    ax.set_xlim(-0.5, ncols - 0.5)
    ax.set_ylim(nrows - 0.5, -0.5)
    ax.set_xlabel("column")
    ax.set_ylabel("row")
    ax.set_title("Sparsity pattern of the stiffness matrix")
    plt.show()
elif MPI.COMM_WORLD.rank == 0:
    print("  Spy skipped because the PETSc matrix is distributed across MPI ranks.")
```

```
Assembled linear system:
  Matrix size: (121, 121)
  Spy of matrix (nonzeros in black):
```



5.3.4. Simulation Output Plot

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

tdim = msh.topology.dim
msh.topology.create_connectivity(tdim, 0)
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)
dofs = fem.locate_dofs_topological(V, 0, vertex_ids)
u_vertex = uh.x.array[dofs].real
```

```
tri = mtri.Triangulation(coords[:, 0], coords[:, 1], triangles=cells)
fig, ax = plt.subplots(figsize=(6, 4))
pcm = ax.tripcolor(tri, u_vertex, shading="gouraud", cmap="inferno")
ax.triplot(tri, lw=0.15, color="white", alpha=0.3)
ax.set_aspect("equal")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Temperature field $u_h$")
fig.colorbar(pcm, ax=ax, label="temperature")
plt.tight_layout()
plt.show()
```

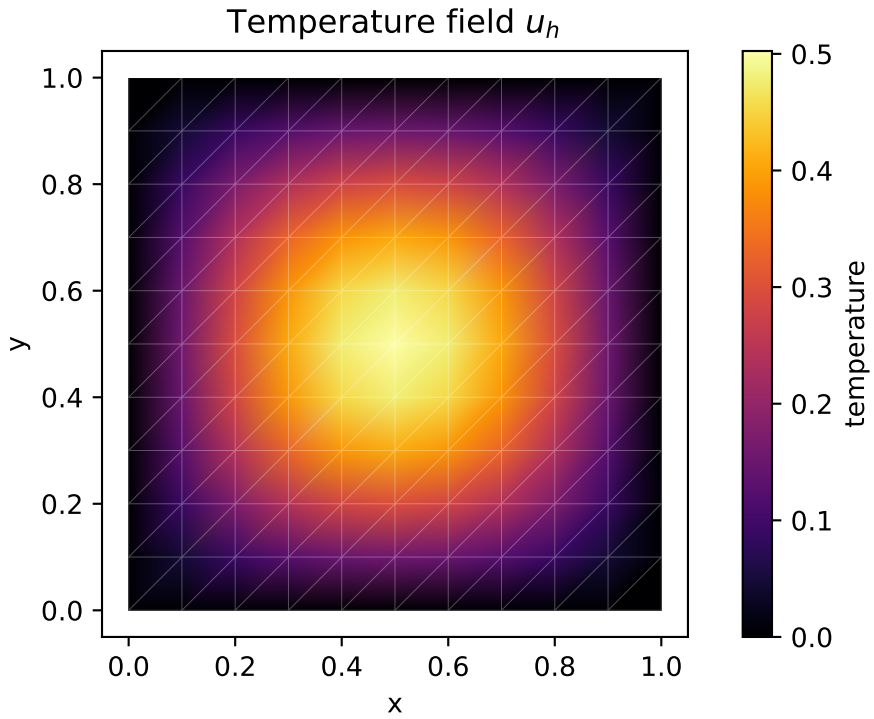


Figure 5.14.: Lecture 02: FEM solution on the unit square

5.3.5. Manufactured Solution and Convergence Check

To verify an FEM implementation, we often use a *manufactured solution*: choose a smooth exact solution u_{ex}, insert it into the PDE, and derive the matching right-hand side f.

For the Poisson problem on $\Omega = (0,1)^2$, take

$$u_{\text{ex}}(x,y) = \sin(\pi x)\sin(\pi y), \qquad -\Delta u_{\text{ex}} = 2\pi^2 \sin(\pi x)\sin(\pi y),$$

with homogeneous Dirichlet boundary conditions. Since $u_{\text{ex}} = 0$ on $\partial\Omega$, it fits the same setup as before. For smooth solutions, we expect the L^2 error to behave like $O(h^{p+1})$ for Lagrange elements of degree p.

```
def solve_manufactured_problem(n, degree):
    msh = mesh.create_unit_square(MPI.COMM_WORLD, n, n)
    V = fem.functionspace(msh, ("Lagrange", degree))

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

    u_exact = ufl.sin(ufl.pi * x[0]) * ufl.sin(ufl.pi * x[1])
    f = 2.0 * ufl.pi**2 * u_exact

    a = ufl.dot(ufl.grad(u), ufl.grad(v)) * ufl.dx
    L = f * v * ufl.dx

    fdim = msh.topology.dim - 1
    boundary_facets = mesh.locate_entities_boundary(
        msh,
        fdim,
        lambda x: np.isclose(x[0], 0.0)
        | np.isclose(x[0], 1.0)
        | np.isclose(x[1], 0.0)
        | np.isclose(x[1], 1.0),
    )
    dofs = fem.locate_dofs_topological(V, fdim, boundary_facets)
    bc = fem.dirichletbc(PETSc.ScalarType(0.0), dofs, V)

    linear_problem_kwargs = {
        "petsc_options": {"ksp_type": "preonly", "pc_type": "lu"}
    }
    if "petsc_options_prefix" in inspect.signature(LinearProblem).parameters:
        linear_problem_kwargs["petsc_options_prefix"] = f"mms_p{degree}_{n}_"

    problem = LinearProblem(a, L, bcs=[bc], **linear_problem_kwargs)
    uh = problem.solve()

    error_sq_local = fem.assemble_scalar(fem.form((uh - u_exact) ** 2 * ufl.dx))
    error_sq = msh.comm.allreduce(error_sq_local, op=MPI.SUM)
    return 1.0 / n, np.sqrt(error_sq.real)


n_values = [4, 8, 16, 32]
convergence_results = {}

for degree in (1, 2):
    hs = []
    errors = []
    for n in n_values:
        h, error_L2 = solve_manufactured_problem(n, degree)
        hs.append(h)
        errors.append(error_L2)
```

```
    hs = np.array(hs)
    errors = np.array(errors)
    orders = np.log(errors[:-1] / errors[1:]) / np.log(hs[:-1] / hs[1:])
    fitted_order = np.polyfit(np.log(hs), np.log(errors), 1)[0]

    convergence_results[degree] = {
        "h": hs,
        "error_L2": errors,
        "orders": orders,
        "fitted_order": fitted_order,
    }

if MPI.COMM_WORLD.rank == 0:
    for degree, data in convergence_results.items():
        print(f"P{degree} elements")
        print("  h         L2 error      observed order")
        for i, (h, err) in enumerate(zip(data["h"], data["error_L2"])):
            order_text = "---" if i == 0 else f"{data['orders'][i - 1]:.3f}"
            print(f"  {h:0.5f}   {err:0.6e}   {order_text}")
```

```
P1 elements
  h         L2 error      observed order
  0.25000   7.907545e-02   ---
  0.12500   2.113277e-02   1.904
  0.06250   5.377435e-03   1.974
  0.03125   1.350436e-03   1.993
P2 elements
  h         L2 error      observed order
  0.25000   4.327631e-03   ---
  0.12500   5.480619e-04   2.981
  0.06250   6.873916e-05   2.995
  0.03125   8.600535e-06   2.999
```

```
if MPI.COMM_WORLD.rank == 0:
    fig, ax = plt.subplots(figsize=(6, 4))

    for degree in (1, 2):
        data = convergence_results[degree]
        line, = ax.loglog(
            data["h"],
            data["error_L2"],
            "o-",
            linewidth=2,
            label=f"P{degree}: observed order {data['fitted_order']:.2f}",
        )
        ref = data["error_L2"][-1] * (data["h"] / data["h"][-1]) ** (degree + 1)
        ax.loglog(
            data["h"],
            ref,
            "--",
```

```
            color=line.get_color(),
            alpha=0.6,
        )

    ax.set_xlabel("mesh size h")
    ax.set_ylabel(r"$L^2$ error")
    ax.set_title("Convergence check with manufactured solution")
    ax.grid(True, which="both", linestyle=":")
    ax.legend()
    plt.tight_layout()
    plt.show()
```

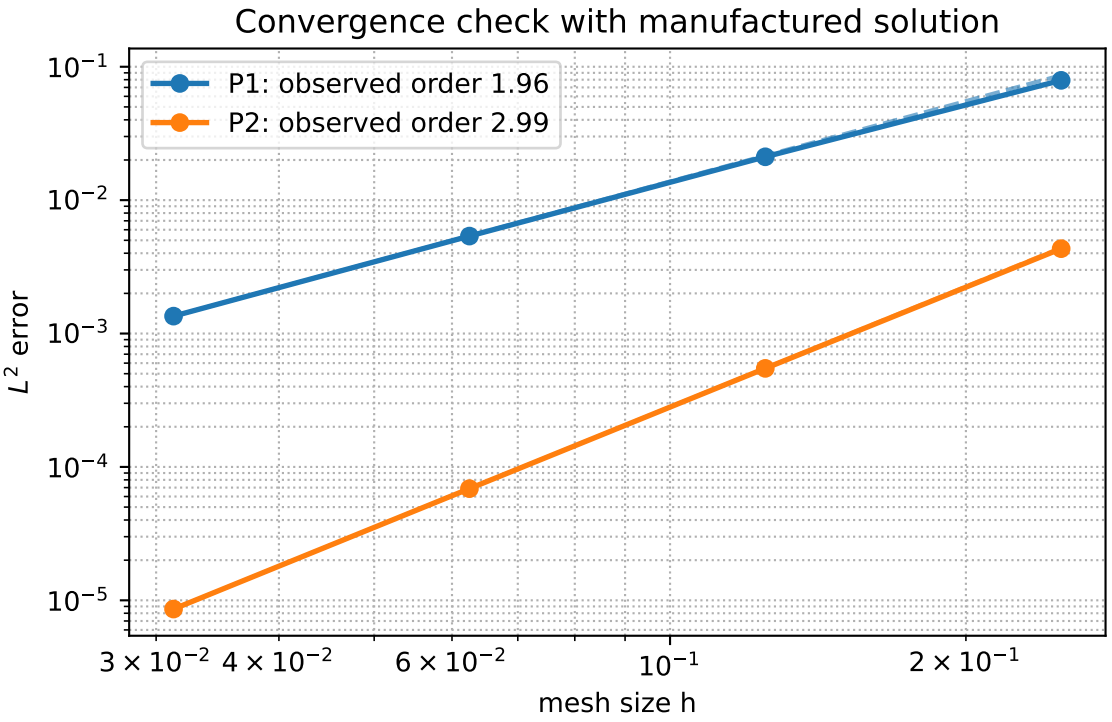


Figure 5.15.: Manufactured-solution error in the L^2 norm for P1 and P2 elements

5.4. Summary

- FEM starts from a weak form and a function space setting.
- Geometry enters naturally via unstructured meshes.
- Assembly yields sparse systems suitable for modern solvers.

5.5. Exercises

1. Derive the weak form for a 2d reaction-diffusion equation with zero BCs and implement it in FEniCSx.
2. Extend the Poisson problem to 3D and implement it in FEniCSx. What changes in the code?

3. Derive the weak form for the equations of linear elasticity, given by

$$-\nabla \cdot \sigma = f \quad \text{in } \Omega,$$

where σ is the stress tensor related to the displacement field u via Hooke's law $\sigma = \lambda(\nabla \cdot u)I + 2\mu\varepsilon(u)$, with $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ being the strain tensor, and λ and μ are the Lamé parameters. Assume appropriate boundary conditions (e.g., Dirichlet on part of the boundary and Neumann on the rest).
4. Formulate the Poisson equation as *mixed problem* by introducing an auxiliary variable $q = -k\nabla u$ (the flux), and derive the corresponding weak form. What function spaces would you choose for u and q?
5. Implement the Poisson equation with Robin boundary conditions in FEniCSx.

5.6. Questions

1. What are the advantages of the finite element method compared to finite difference methods, especially for complex geometries?
2. How does the choice of finite element (e.g., P1 vs P2) affect the accuracy and computational cost of the solution?
3. What are some common challenges in implementing FEM, and how does a framework like FEniCS help to address them?

5.7. References

- Solving the Poisson equation by Jørgen S. Dokken

6. Lecture 03 - FEM II

Nonlinear problems, mixed problems, eigenvalue problems

6.1. 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).

6.1.1. Internals of the FEniCS project

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

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

6.2. 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

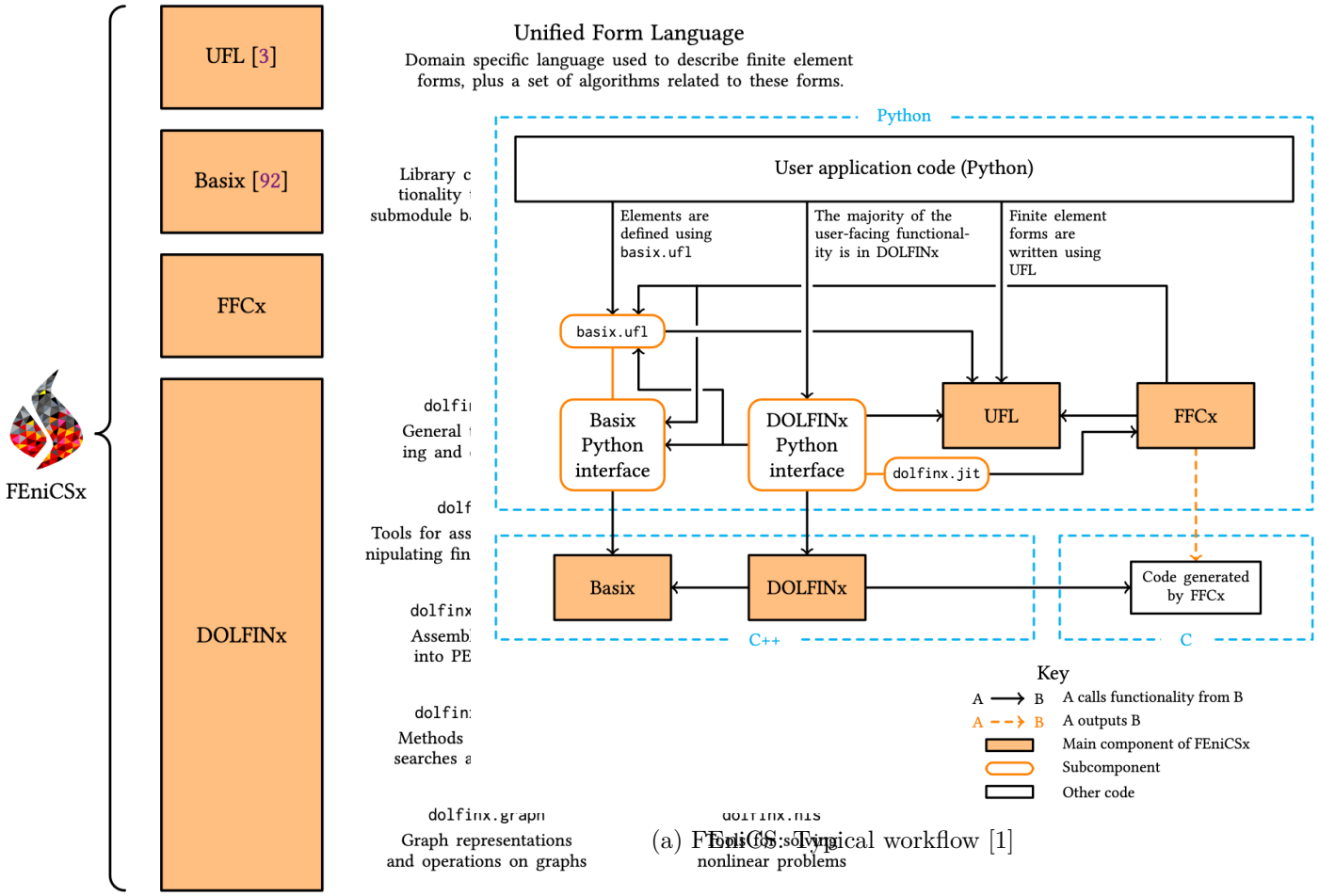
$$-\nabla \cdot (q(u)\nabla u) = f$$

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

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



(a) FEniCS: packages overview [1]

(a) FEniCS: Typical workflow [1]

Known coefficient k.

Nonlinear Poisson

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

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

6.2.1. 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)
```

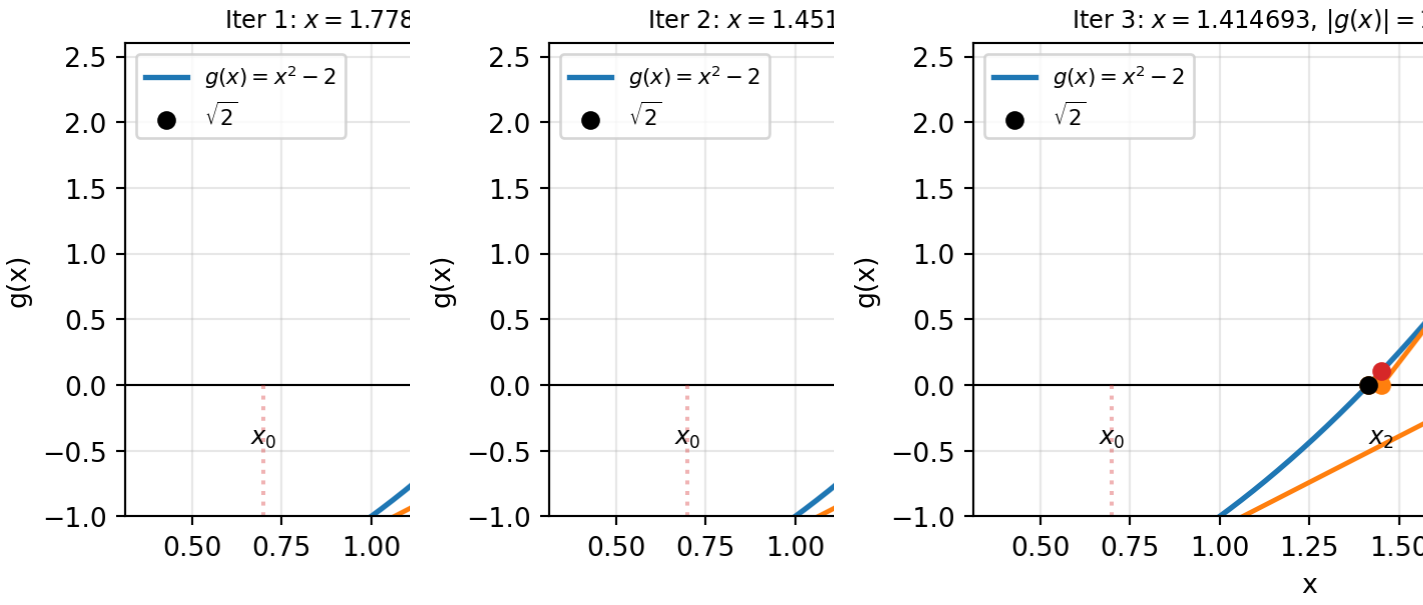
```
        ax.plot(tangent_xs, tangent, color="tab:orange", linewidth=1.8)
        ax.plot(
            [xk_val, xk_val], [0, g(xk_val)], color="tab:red", alpha=0.35,
            linestyle=":"
        )
        ax.scatter([xk_val], [g(xk_val)], color="tab:red", zorder=3)
        ax.scatter([x_next], [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.

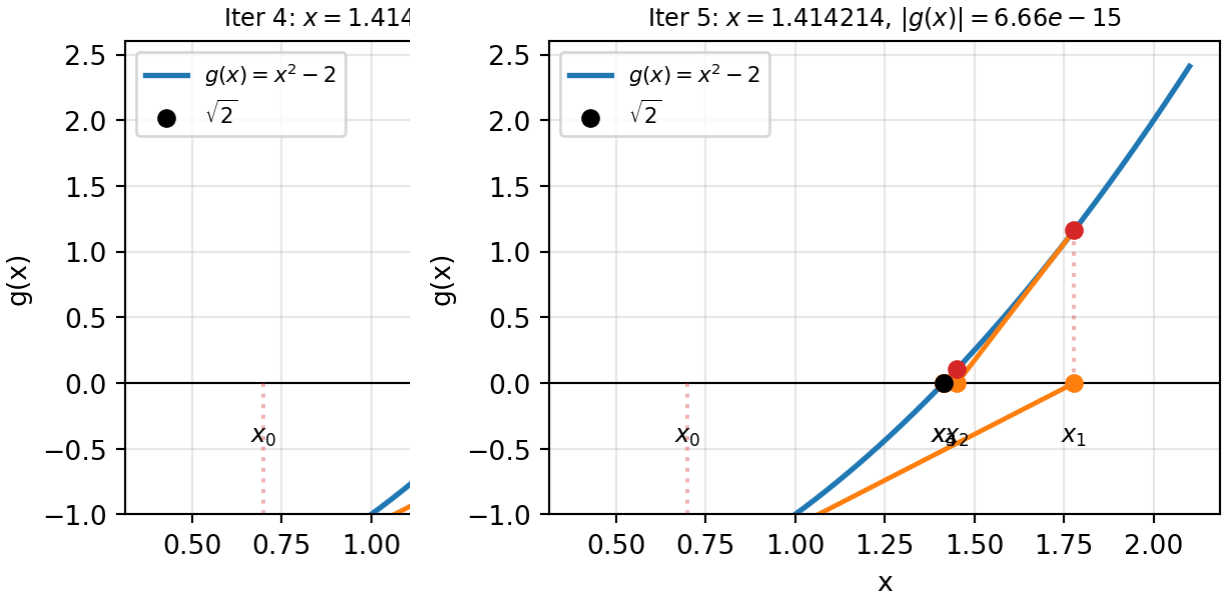
> **From scalar Newton to PDE Newton**
>
> The scalar derivative $g'(x_k)$ becomes the Jacobian form $J(u_h; \delta u, v)$. The scalar correction $x_{k+1} - x_k$ becomes a finite element function δu_h.

```
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 6.3.: Newton's method for $g(x) = x^2 - 2$

6.2.2. 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
```

6.2.3. 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()
```

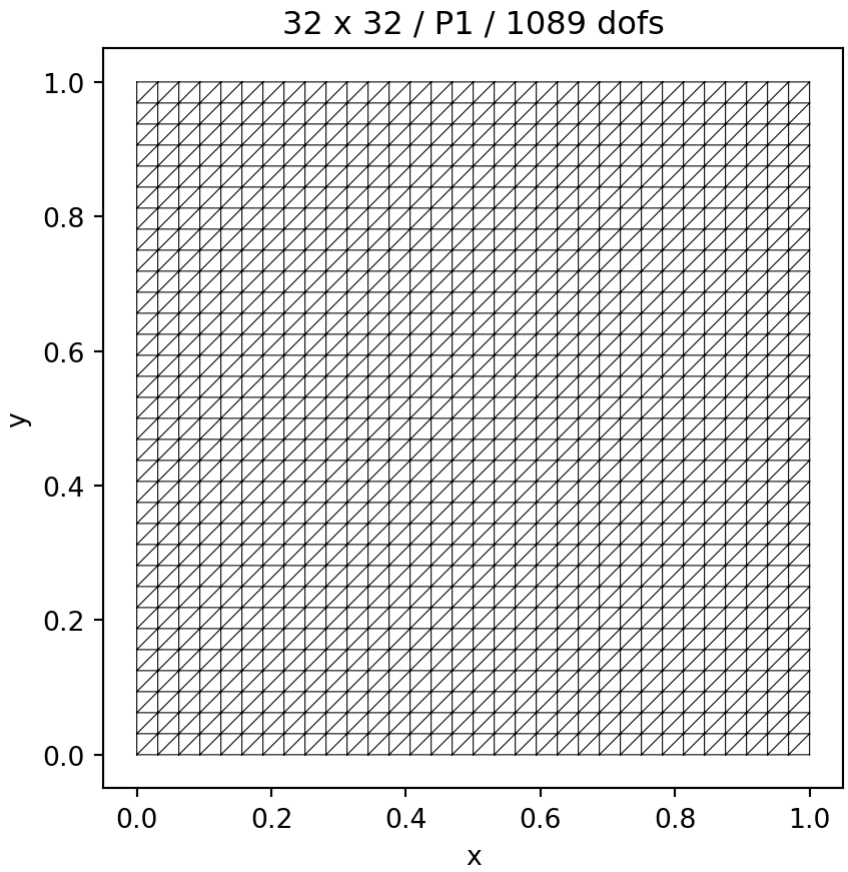


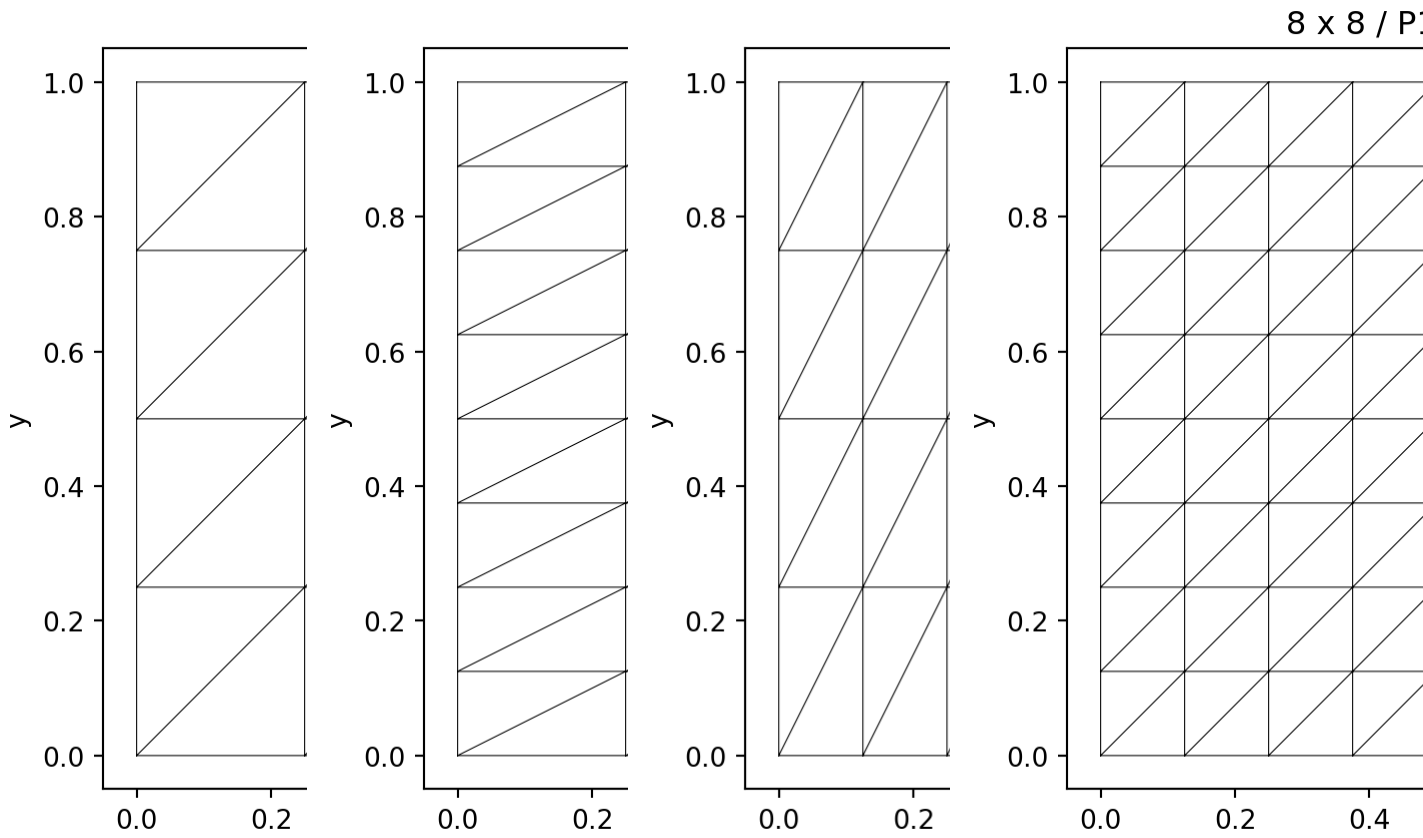
Figure 6.4.: Triangular unit-square mesh.

6.2.4. 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 6.5.: Effect of mesh resolution on the unit-square triangulation.

6.2.5. 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.

```python
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()
```

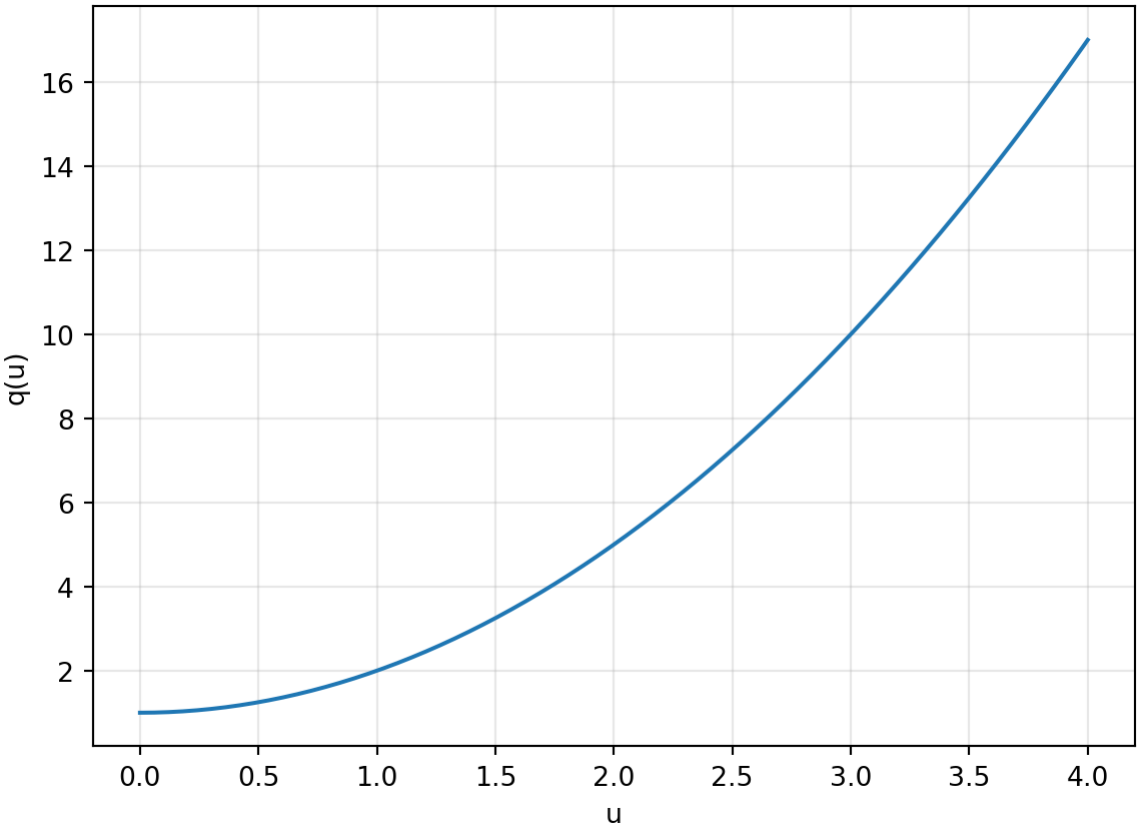


Figure 6.6.: Nonlinear diffusion coefficient $q(u) = 1 + u^2$.

6.2.5.1. 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()
```

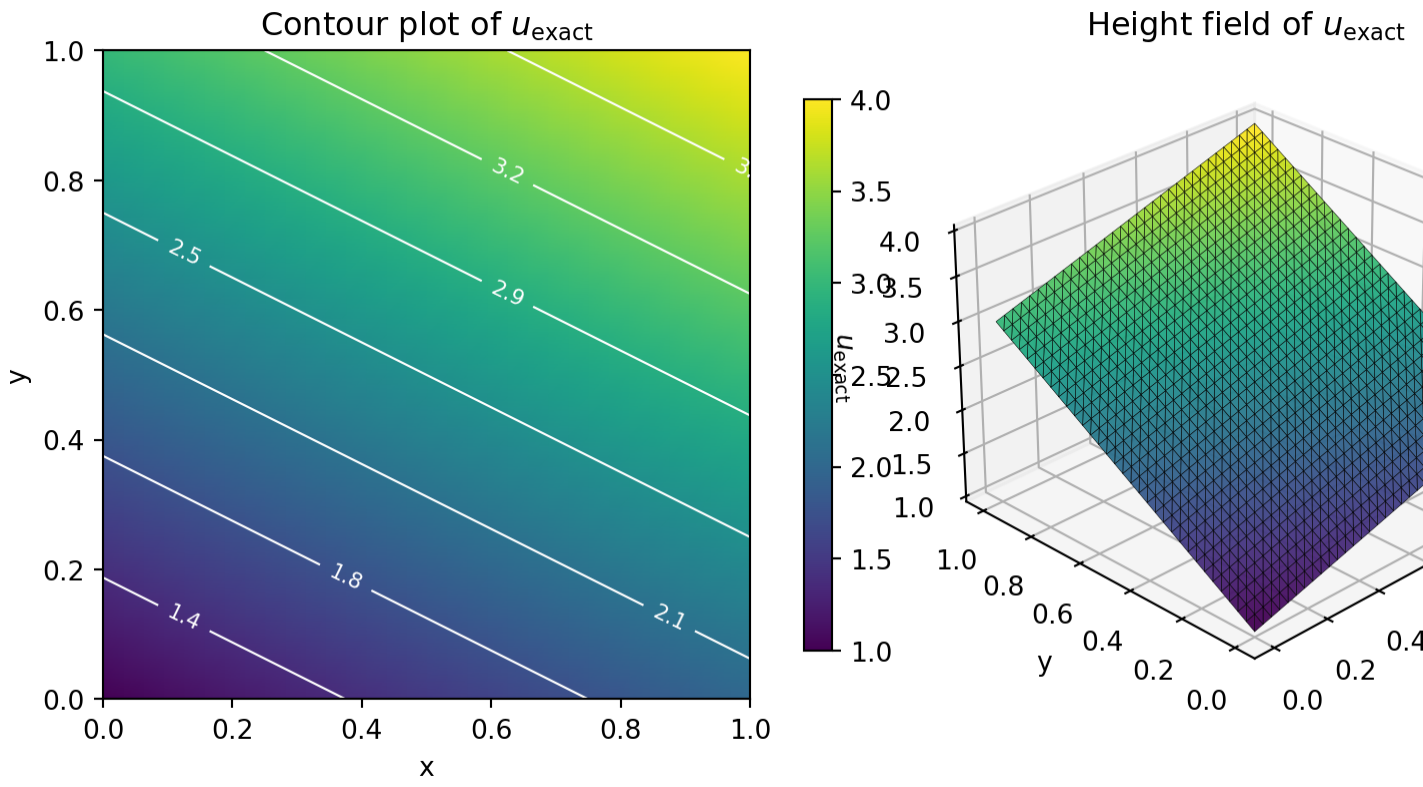


Figure 6.7.: Manufactured exact solution.

6.2.5.2. 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)
```

6.2.6. 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:

6.2.6.1. 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]).
```

6.2.7. Unknown, test function, residual

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

> **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
```

6.2.8. 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
)
```

6.2.9. 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: $u^{k+1} = u^k + \delta u$

6.2.10. 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)
```

6.2.11. 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 5.444213338728e-14
  1 SNES Function norm 4.633381267836e+00
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 4.633381267836e+00
    1 KSP Residual norm 1.010357646312e-13
  2 SNES Function norm 1.828141534723e+00
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 1.828141534723e+00
    1 KSP Residual norm 4.843463627381e-14
  3 SNES Function norm 2.306519444364e-01
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 2.306519444364e-01
    1 KSP Residual norm 4.334392112088e-15
  4 SNES Function norm 5.674866171689e-03
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 5.674866171689e-03
    1 KSP Residual norm 7.218060840330e-17
  5 SNES Function norm 3.108314093175e-06
    Residual norms for nlpoisson_ solve.
    0 KSP Residual norm 3.108314093175e-06
    1 KSP Residual norm 1.661336402630e-20
  6 SNES Function norm 7.129537350988e-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.
```

6.2.12. 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()
```

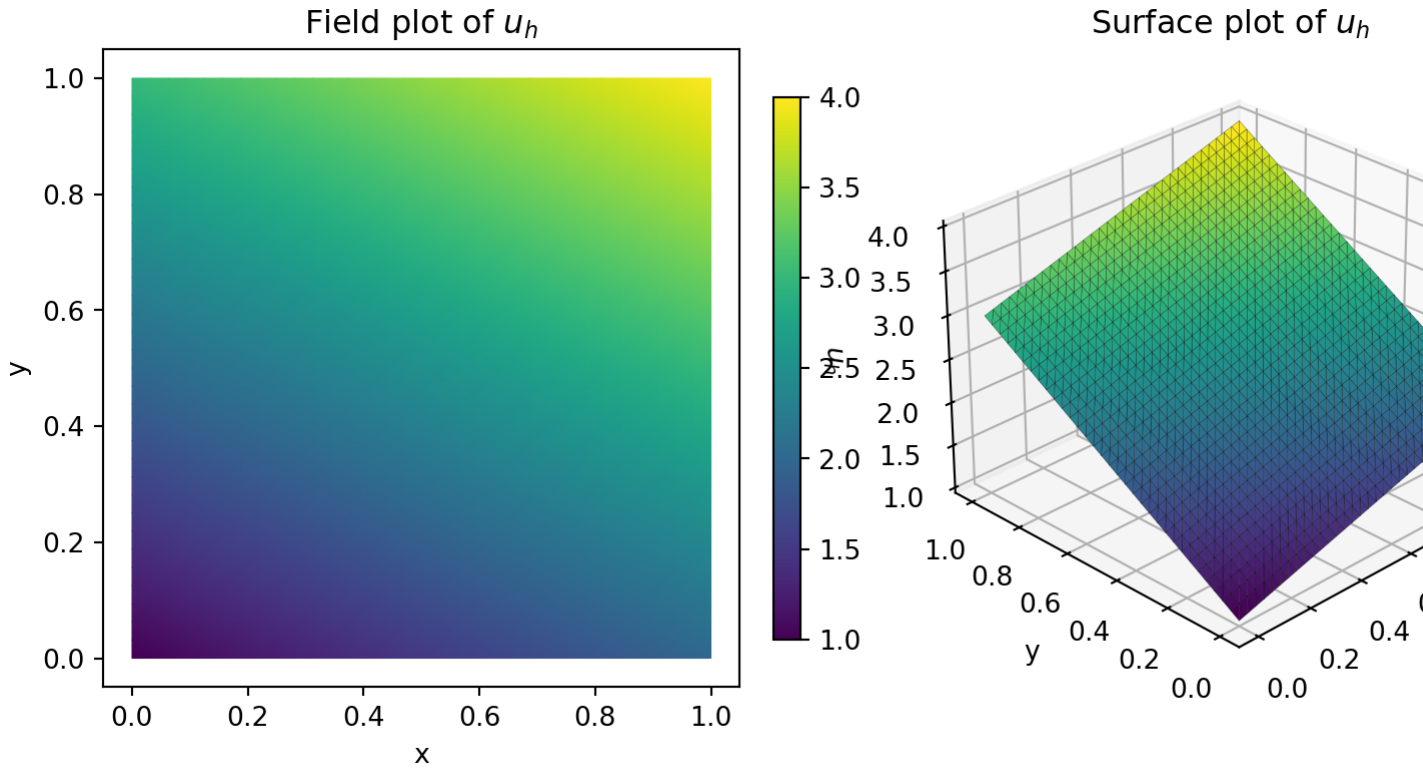


Figure 6.8.: Computed nonlinear Poisson solution u_h.

6.2.13. Error check

Since the exact solution is known from Section 6.2.5, 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.

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

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

6.2.13.1. 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      = 9.193e-15
H1-semi error = 1.475e-13
H1-full error = 1.478e-13
```

6.2.13.2. 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()
```

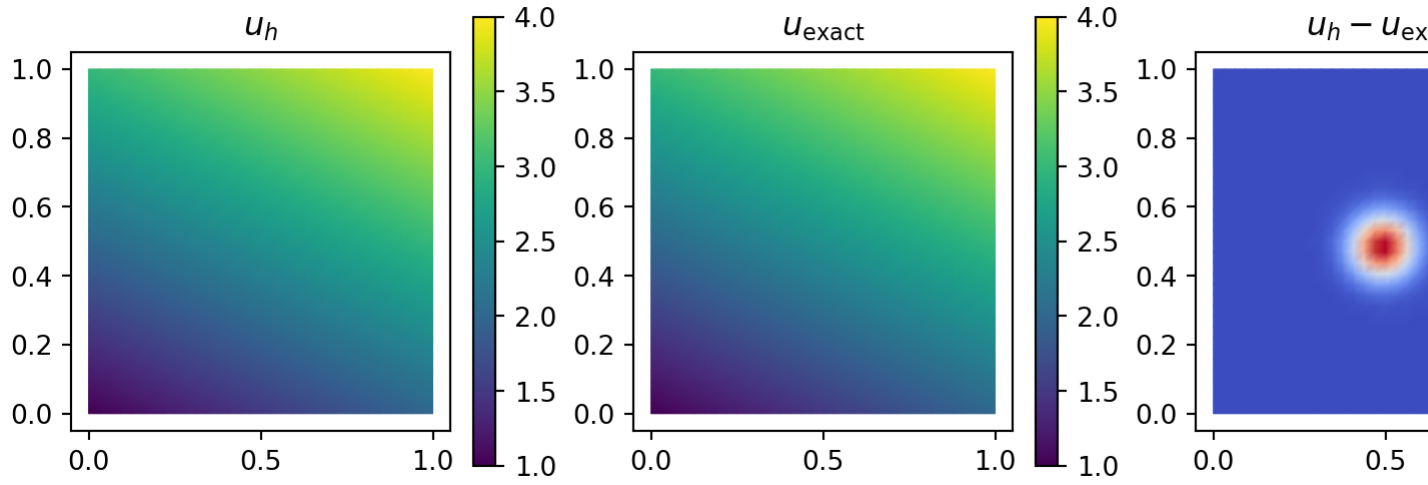


Figure 6.9.: Numerical solution, exact solution, and pointwise error.

6.2.14. 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
```

6.2.15. 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.

6.3. 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.

6.3.1. 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()
```

```
27
28      hole_surfaces = set(entity_map[1]) | set(entity_map[2])
29      cut_surfaces = set(entity_map[3])
30      removed_surfaces = hole_surfaces | cut_surfaces
31      domain_surfaces = [
32          tag
33          for dim, tag in entity_map[0]
34          if dim == 2 and (dim, tag) not in removed_surfaces
35      ]
36
37      def boundary_curves(surface_entities):
38          curves = set()
39          for surface in surface_entities:
40              for dim, tag in gmsh.model.getBoundary([surface], oriented=False):
41                  if dim == 1:
42                      curves.add(tag)
43          return curves
44
45      domain_entities = [(2, tag) for tag in domain_surfaces]
46      domain_curves = boundary_curves(domain_entities)
47      hole_curves = boundary_curves(hole_surfaces)
48
49      # The heated boundary is the part of the domain boundary that is shared
50      # with a disk surface. This also works if a disk is split by the cutout.
51      hot_wall = sorted(domain_curves & hole_curves)
52
53      # The disk and cutout surfaces were only kept to identify shared curves.
54      # Remove them before meshing, otherwise Gmsh also meshes the holes.
55      occ.remove(list(removed_surfaces), recursive=False)
56      occ.synchronize()
57
58      remaining_curves = domain_curves - set(hot_wall)
59
60      def curves_on_vertical_boundary(curves, x_value, tol=1e-5):
61          selected = []
62          for tag in curves:
63              xmin, _, _, xmax, _, _ = gmsh.model.getBoundingBox(1, tag)
64              if abs(xmin - x_value) < tol and abs(xmax - x_value) < tol:
65                  selected.append(tag)
66          return sorted(selected)
67
68      # Inlet and outlet are semantic geometric labels: the left and right
69      # parts of the exterior boundary of the final fluid domain. The heated
70      # hole boundary above is tagged purely topologically.
71      left = curves_on_vertical_boundary(remaining_curves, 0.0)
72      right = curves_on_vertical_boundary(remaining_curves, 0.28)
73      wall = sorted(remaining_curves - set(left) - set(right))
74
75      gmsh.model.addPhysicalGroup(2, domain_surfaces, 1)
76      gmsh.model.setPhysicalName(2, 1, "solid")
77      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
```

6.3.1.1. 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()
```

6.3.2. 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    : [  0%] Fragments
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 0.000275s, CPU 0.000293s)
Info    : Meshing 2D...
Info    : Meshing surface 5 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.0764232s, CPU 0.075834s)
Info    : 6089 nodes 12172 elements
```

6.3.3. 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
```

6.3.4. 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.

6.3.5. 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].

6.3.5.1. 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"
```

6.3.6. 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.

> **i** 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.

6.3.6.1. 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"
```

6.3.7. 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)
```

6.3.8. 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)
```

6.4. Linear elasticity

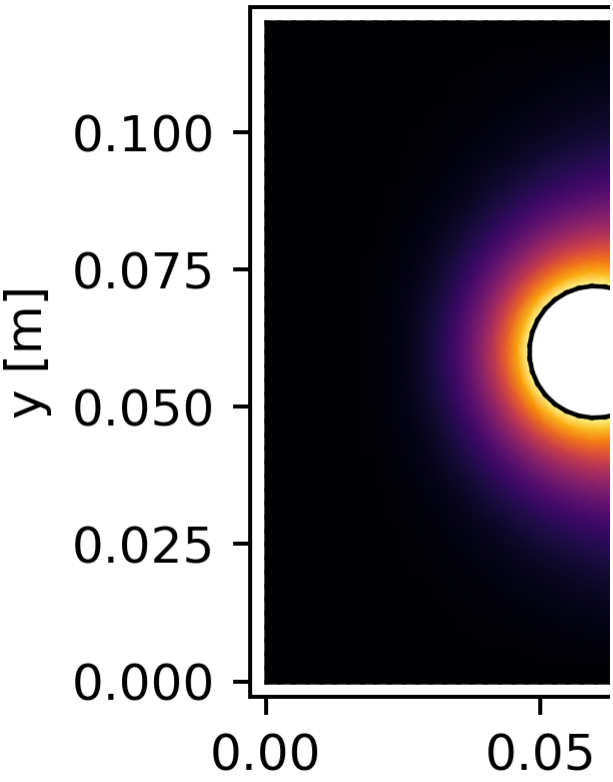
- TODO
- See very good material by Jeremy Bleyer

6.5. Schrödinger's equation

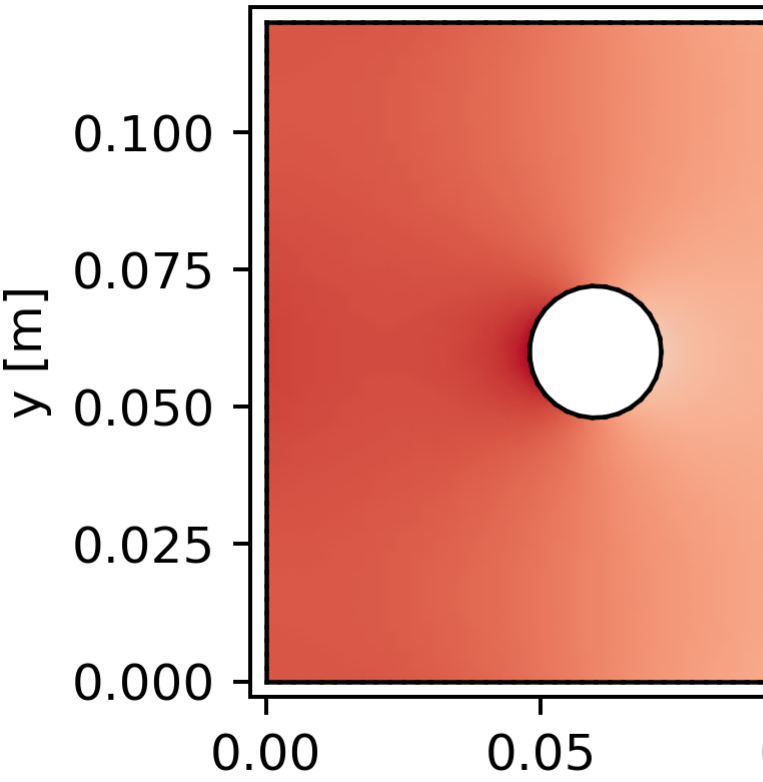
We now turn from boundary value problems to **eigenvalue problems (EVPs)**. A canonical example from quantum mechanics is the time-independent Schrödinger equation

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

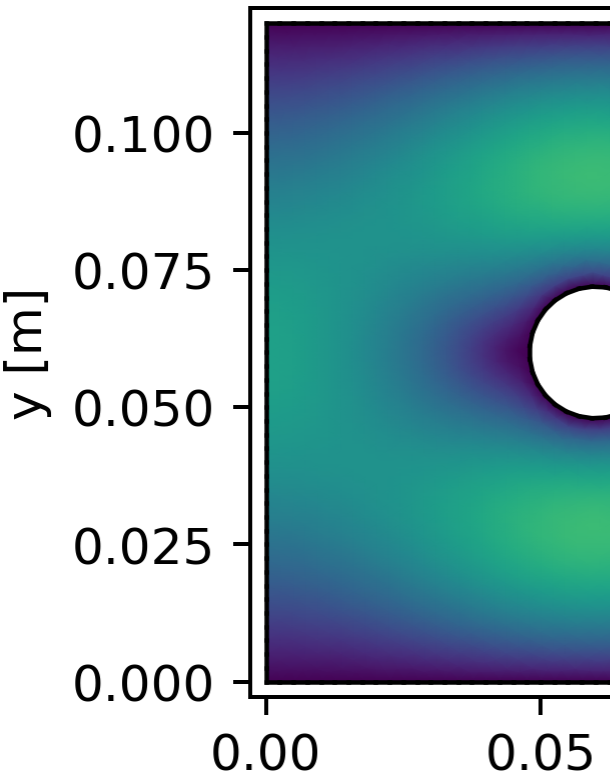
where we use atomic units ($\hbar = m = 1$). The eigenvalues E_n are the allowed energies and the eigenfunctions ψ_n are the stationary states.



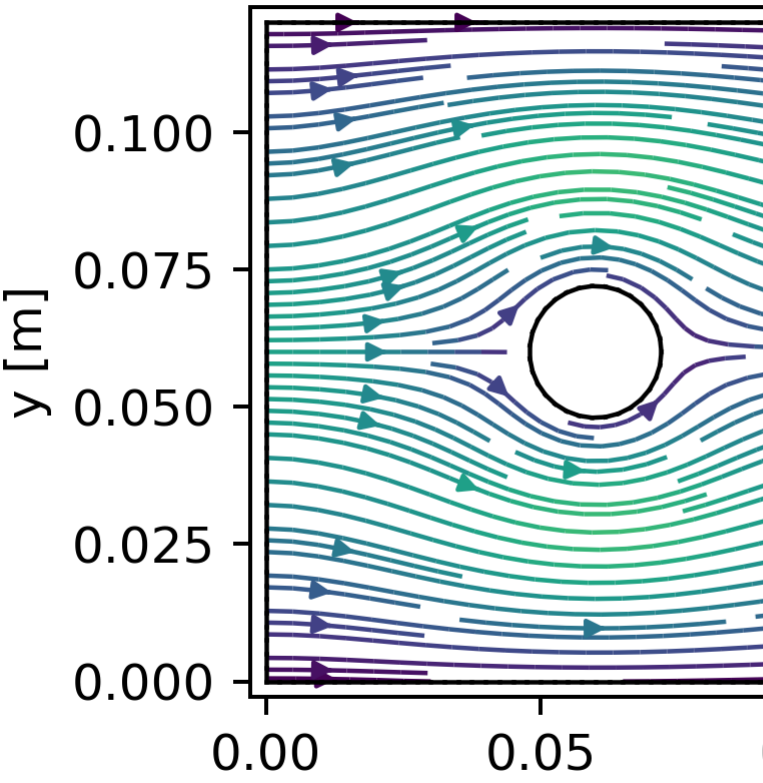
(a) Temperature



(b) Pressure



(c) Velocity magnitude



(d) Velocity streamlines

6.5.1. 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].

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.

i 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.

6.5.2.1. 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]
```

6.5.3. 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.

6.5.4. 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()
```

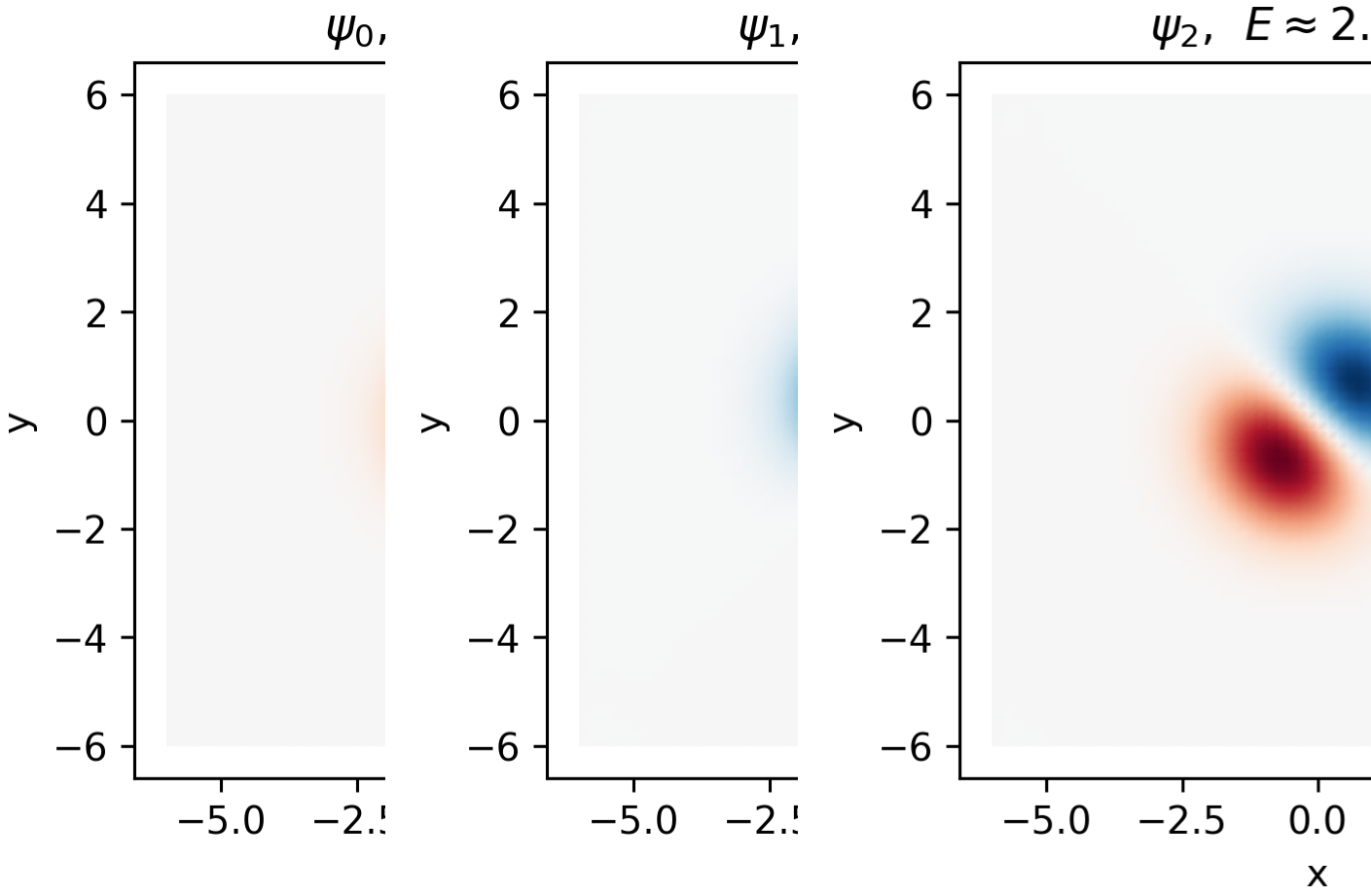
6.5.5. Remarks and extensions

- **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.

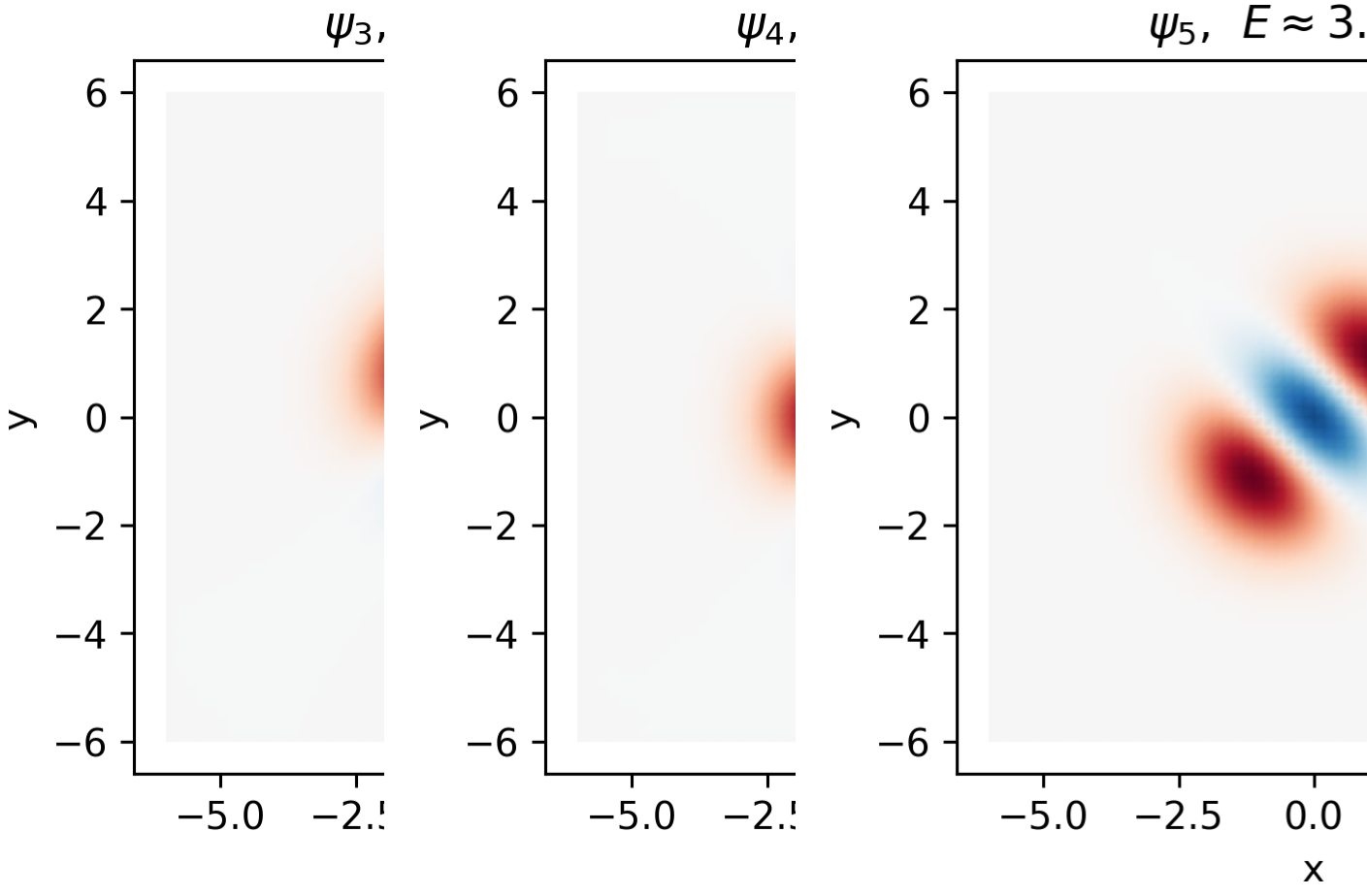
6.6. Navier–Stokes (Non-Newtonian)

- 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.

6.7. References

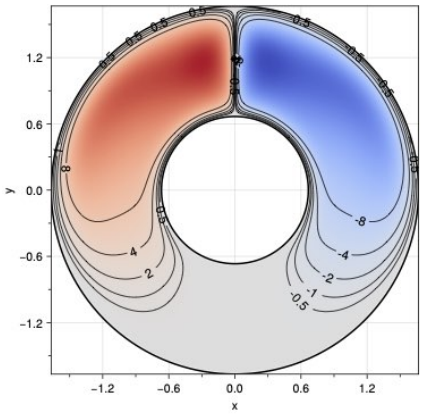


(a) n=0: E 1 (ground state) (b) n=1: E 2 (c) n=2: E 2

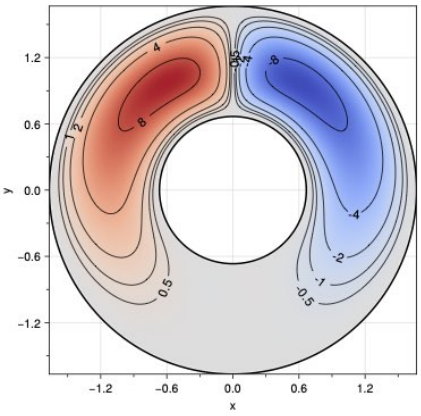


(d) n=3: E 3 (e) n=4: E 3 (f) n=5: E 3

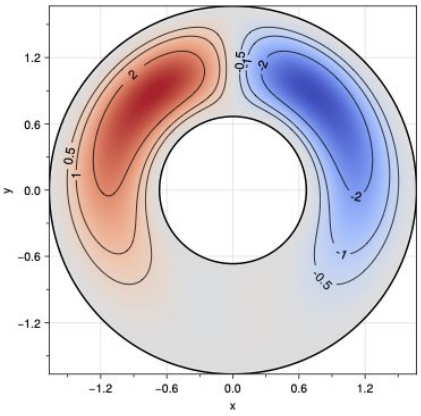
Figure 6.12.: Lowest six eigenfunctions of the 2D quantum harmonic oscillator computed with FEniCSx



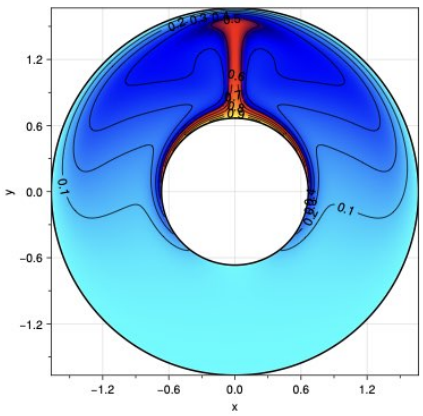
(a) Stream function, $n = 0.6$



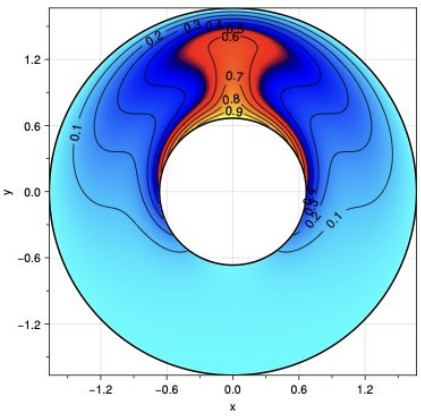
(a) Stream function, $n = 1.0$



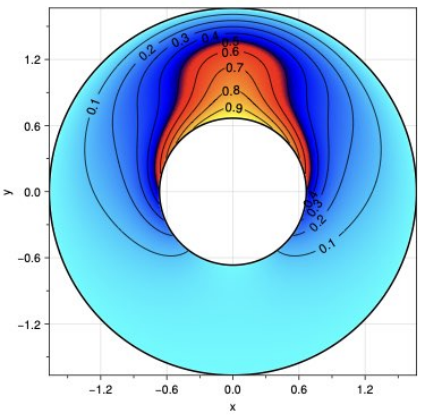
(a) Stream function, $n = 1.4$



(a) Temperature, $n = 0.6$



(a) Temperature, $n = 1.0$



(a) Temperature, $n = 1.4$

7. Lecture 04 - FEM III

PETSc, preconditioning, TS, AMR, …

7.1. 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

```
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
```

7.2. 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.

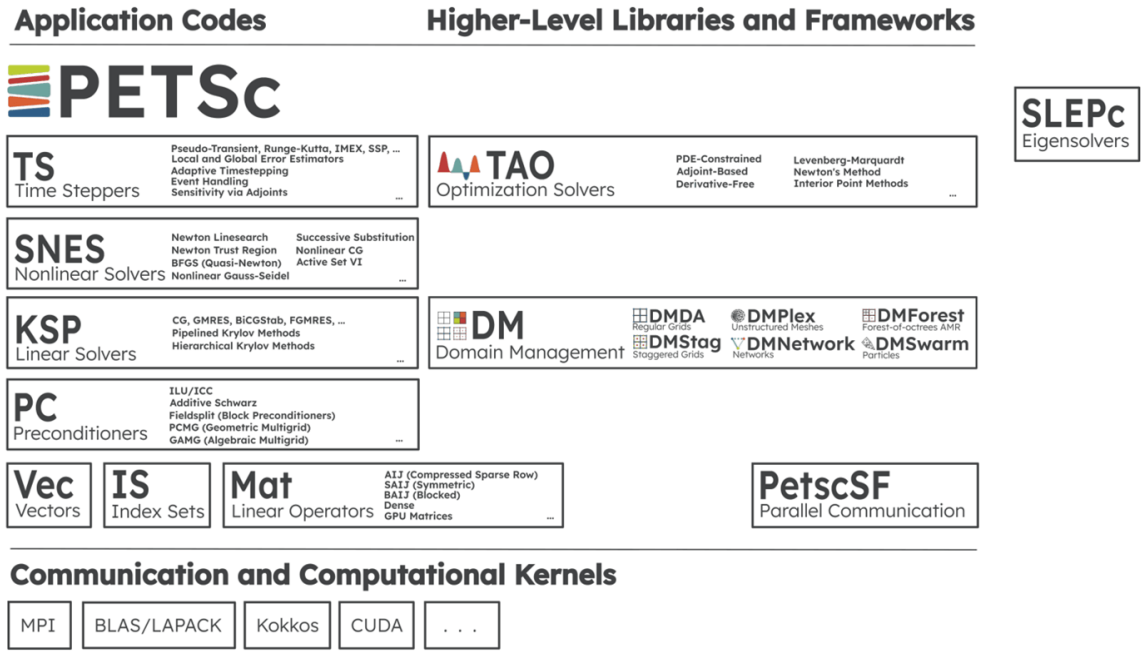


Figure 7.1.: PETSc overview [8],[9]

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.

7.2.1. 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",
    },
}
```

7.3. 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),
    }
```

7.3.1. 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
```

7.3.2. 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)
```

7.3.3. 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()
```

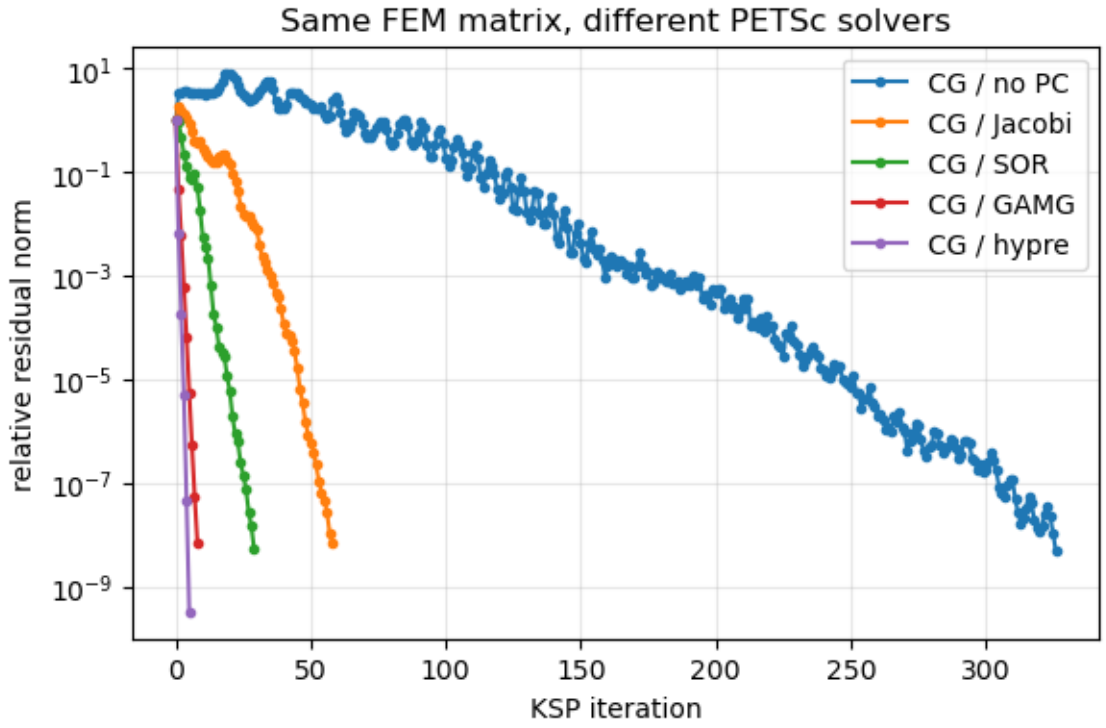


Figure 7.2.: PETSc residual histories for the high-contrast elliptic solve.

```
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()
```

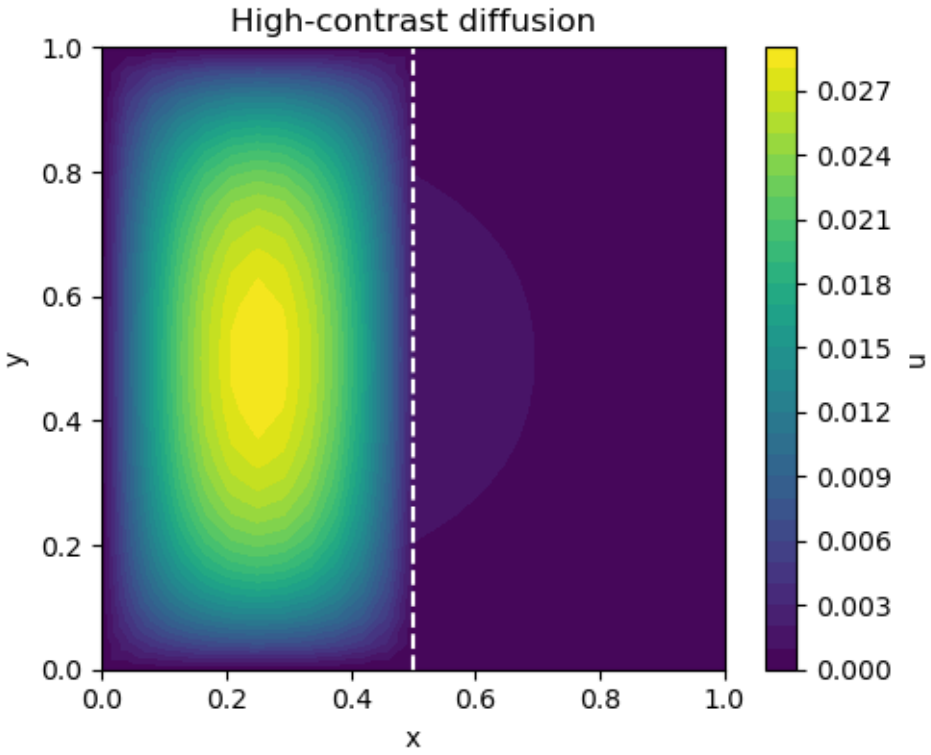


Figure 7.3.: Diffusion solution computed with CG and PETSc GAMG preconditioning.

7.3.4. 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.

```
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)

```

```
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                 ---
```

```
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()
```

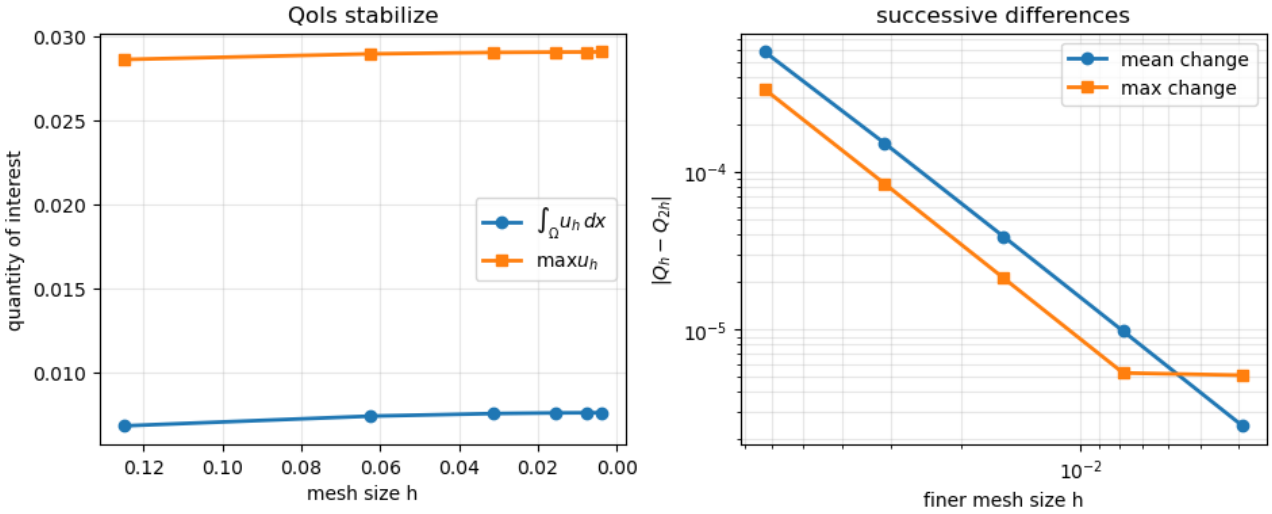


Figure 7.4.: Scalar quantities of interest stabilize under refinement, and successive changes estimate the remaining discretization effect.

```
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()
```

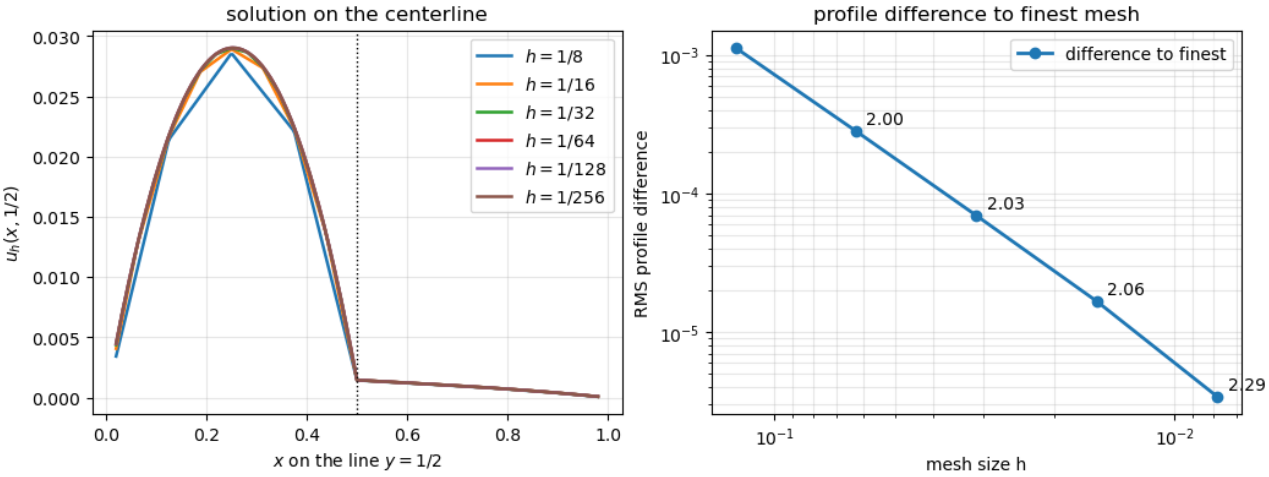


Figure 7.5.: Solution restricted to the centerline $y = 1/2$ for different mesh resolutions. The right panel compares each centerline profile against the finest available mesh, which is only a numerical reference.

```
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()
```

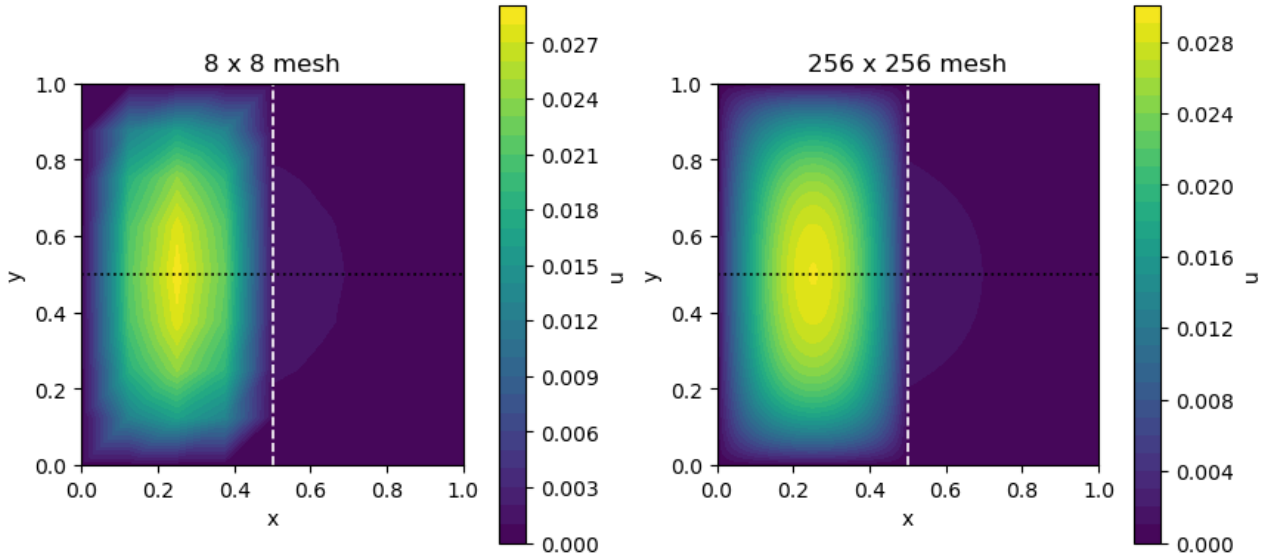


Figure 7.6.: Coarse and finest mesh solutions for the same high-contrast elliptic problem. The white vertical line marks the coefficient jump, and the black horizontal line marks the sampled centerline $y = 1/2$.

> **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.

7.3.5. 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
```

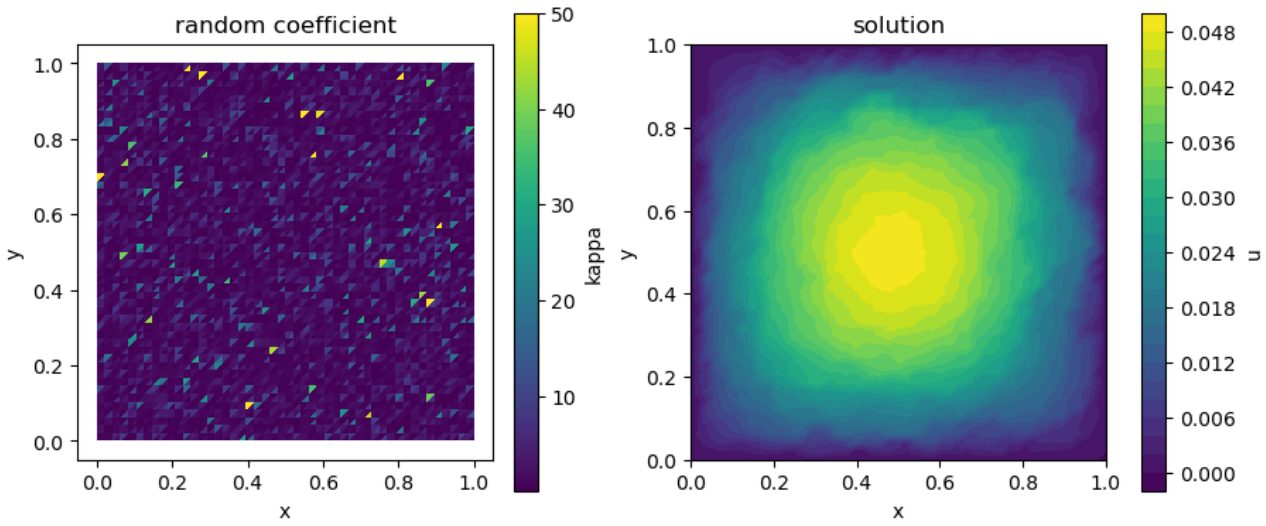


Figure 7.7.: Random high-contrast diffusion coefficient and resulting elliptic solution.

7.4. 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.

7.4.1. 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,

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

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

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

- `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",
    },
}
```

7.4.2. 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
```

```
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()
```

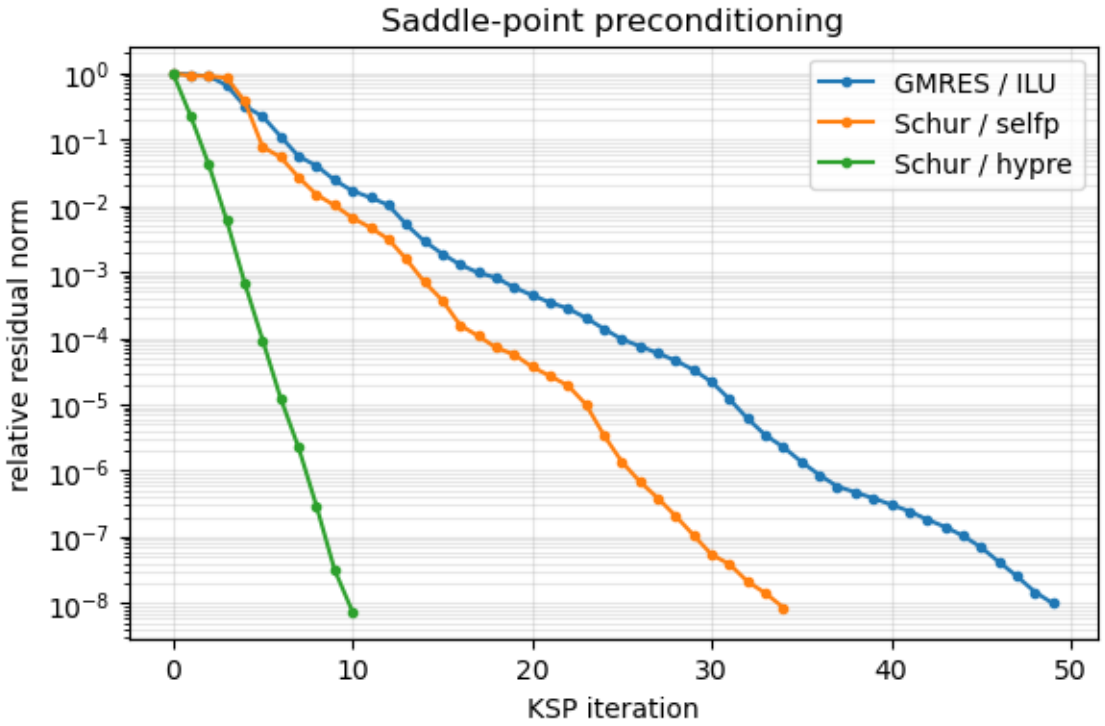


Figure 7.8.: Residual histories for a Firedrake-style mixed Poisson saddle-point system.

7.4.3. 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
```

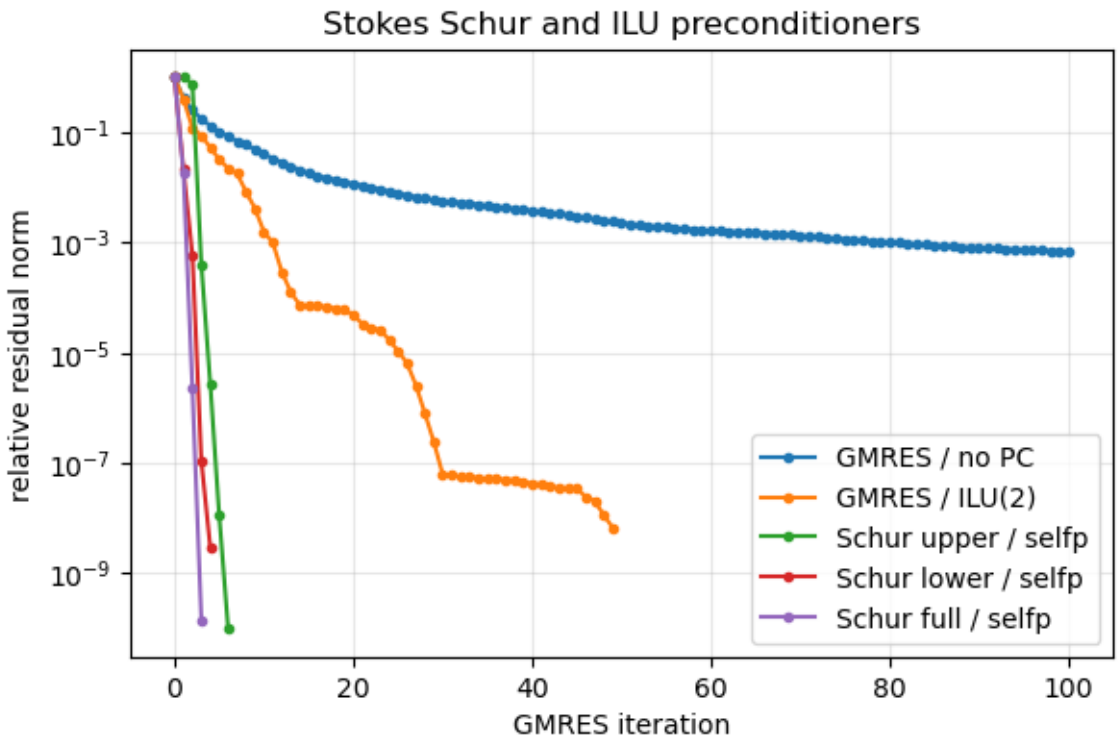


Figure 7.9.: PETSc residual histories for nested Stokes solves and monolithic ILU(k).

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()
```

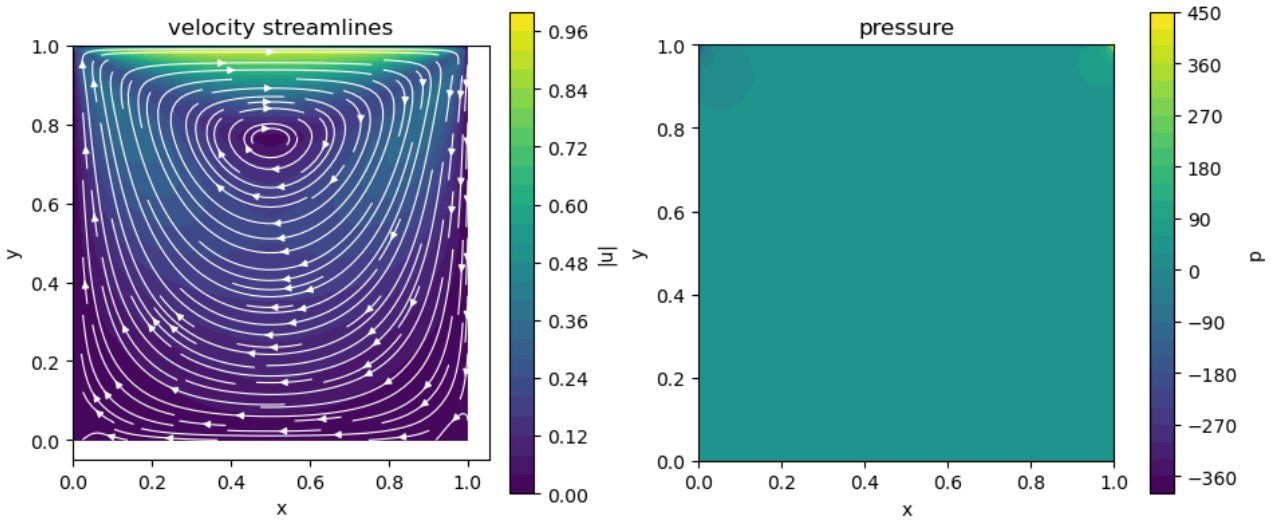


Figure 7.10.: Taylor–Hood Stokes lid-driven cavity: streamlines, velocity magnitude, and pressure.

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.

7.5. Some further examples...

7.5.1. 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,
```

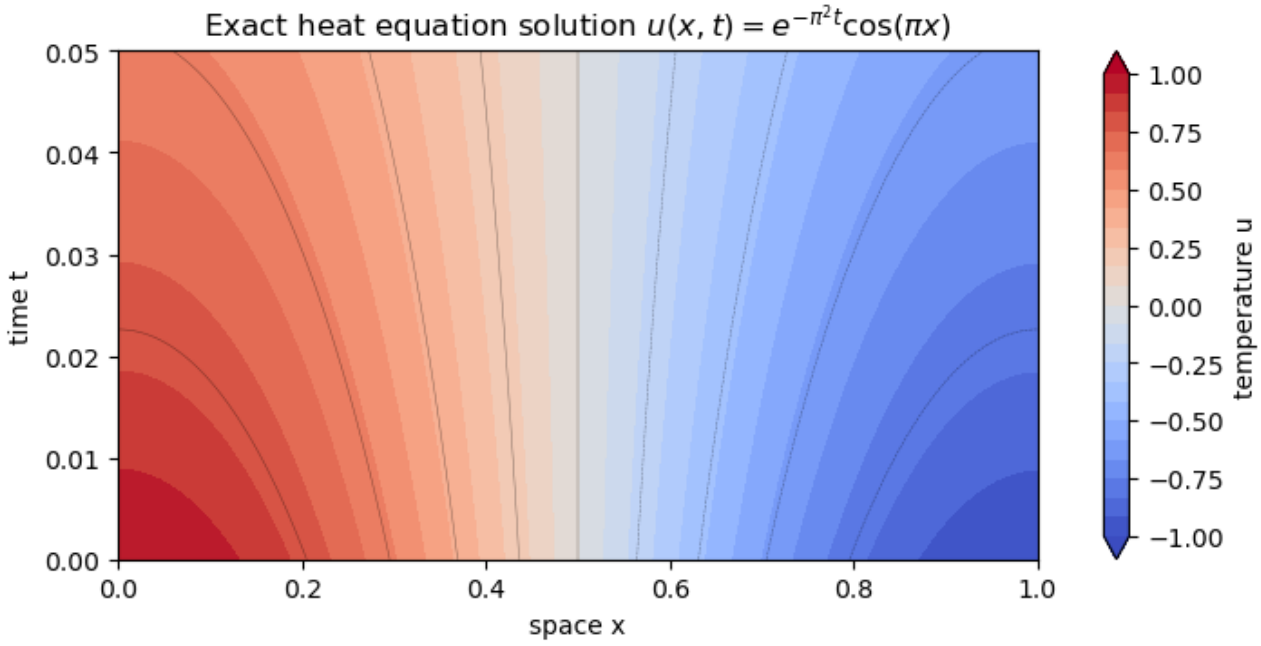
```
        points=((0.0, 0.0), (1.0, T)),
        n=(nx, nt),
        cell_type=mesh.CellType.triangle,
    )
    V = fem.functionspace(domain, ("Lagrange", degree))

    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(
```

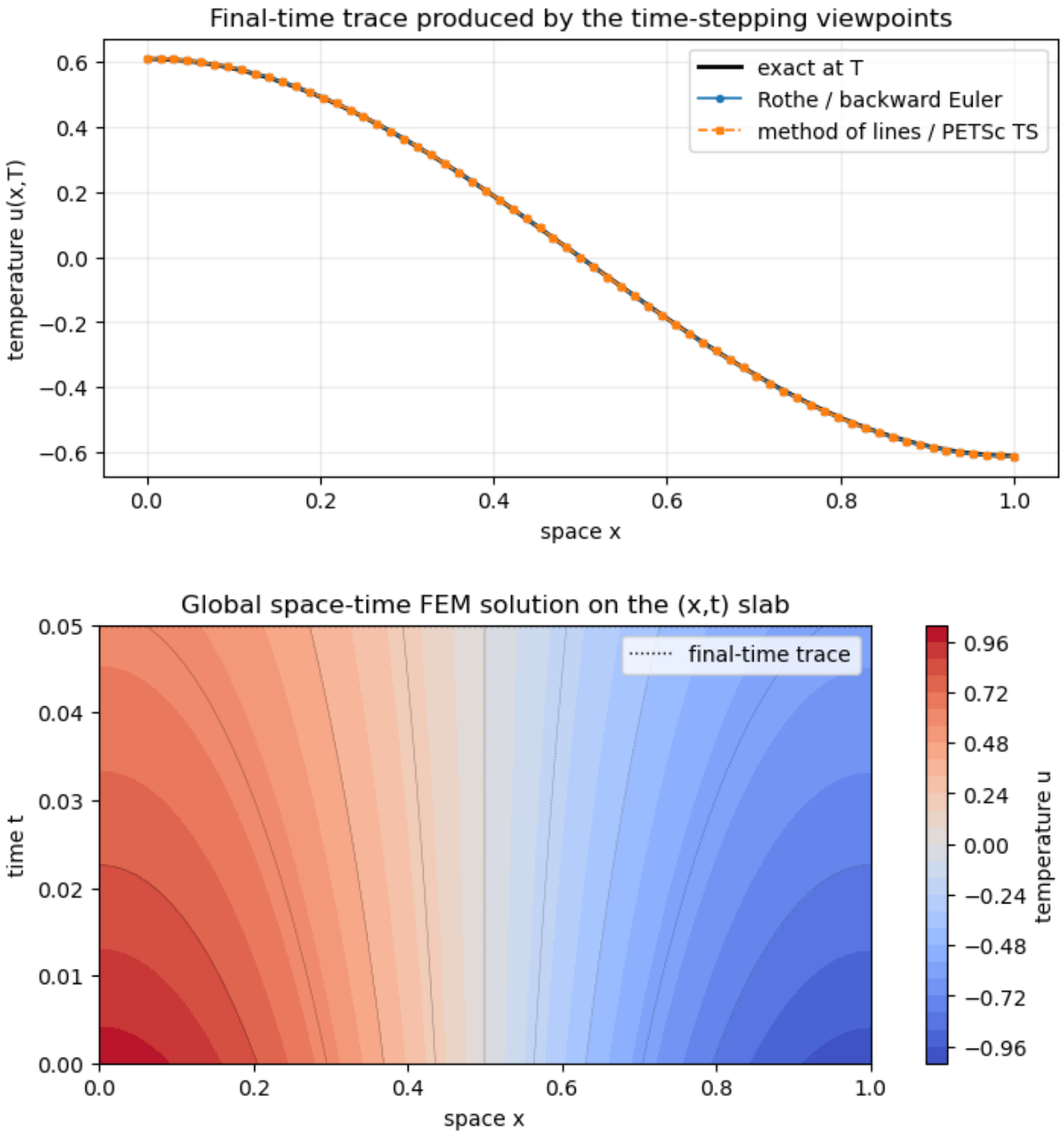
```
            np.abs(
                vals[dofs_final]
                - np.exp(-math.pi**2 * T) * np.cos(math.pi * dof_coords[owned][dofs_final, 0])
            )
        )
    else:
        err_local = 0.0
    err_global = comm.allreduce(err_local, op=MPI.MAX)
    return Uxt, float(err_global)


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
```





7.5.2. 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.

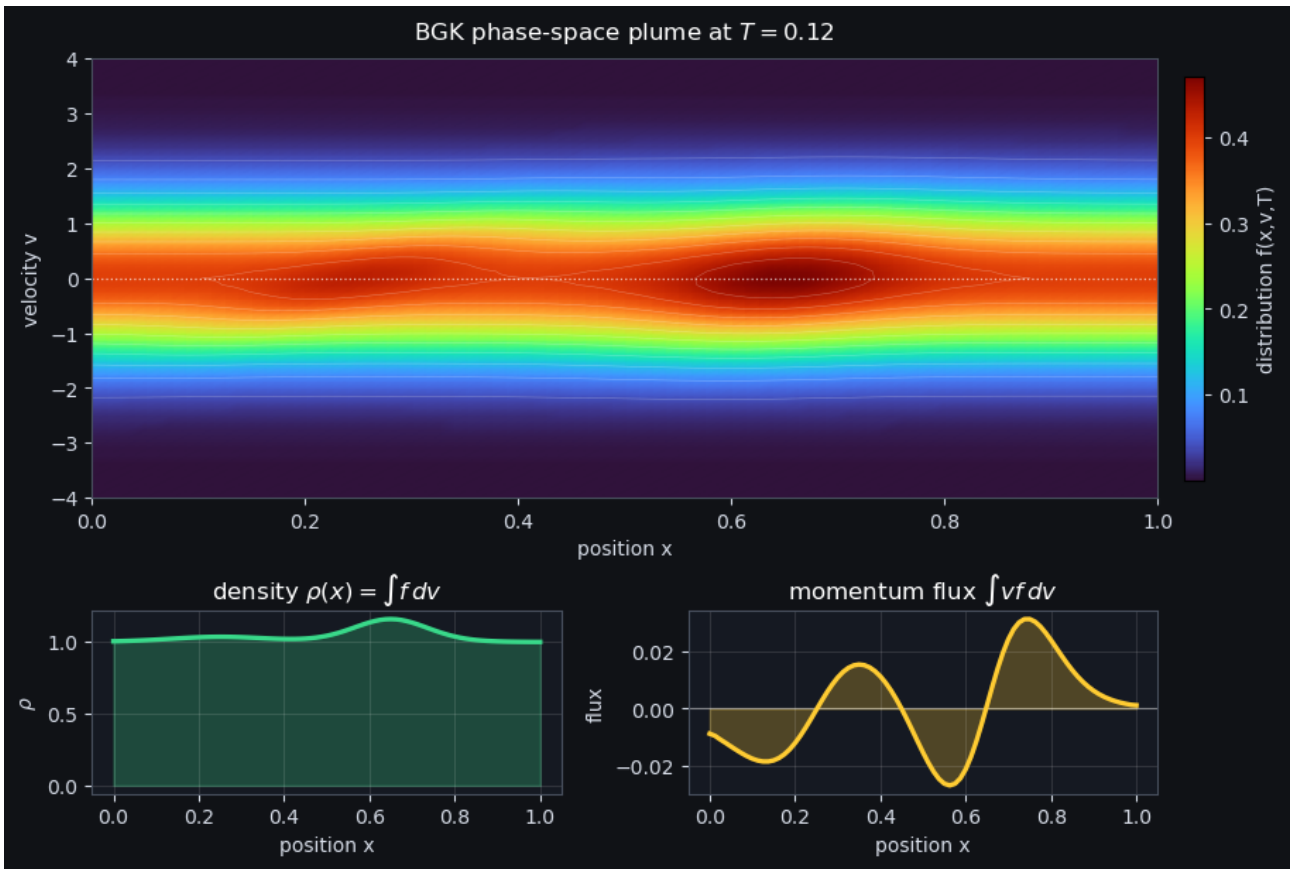
```
import math

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 7.11.

7.5.3. Hemker problem (convection-dominated)

As a final solver/stabilization example, we compare three discretizations for the Hemker convection-diffusion problem, see, e.g. [10] or [11], 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()
```

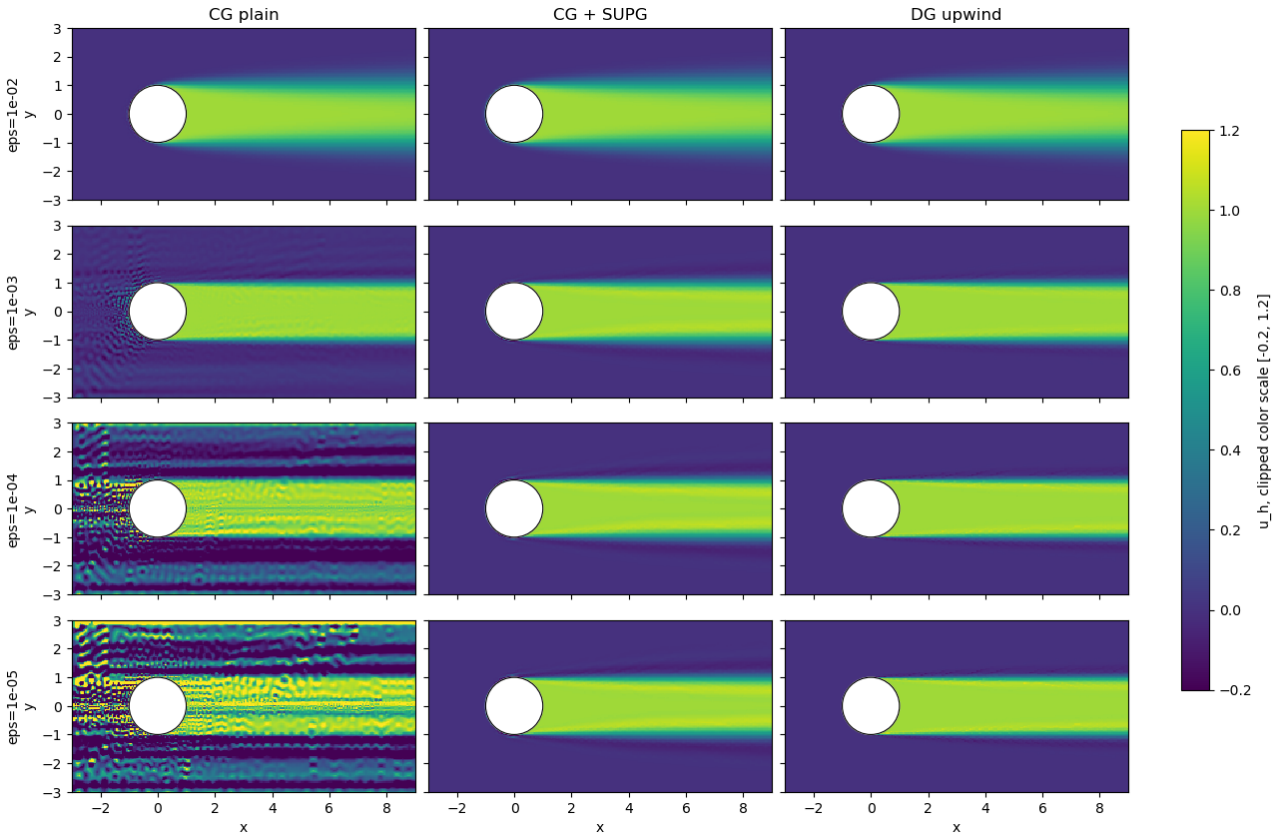


Figure 7.12.: Hemker convection-dominated benchmark for decreasing diffusion.

> **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.

7.5.4. 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()
```

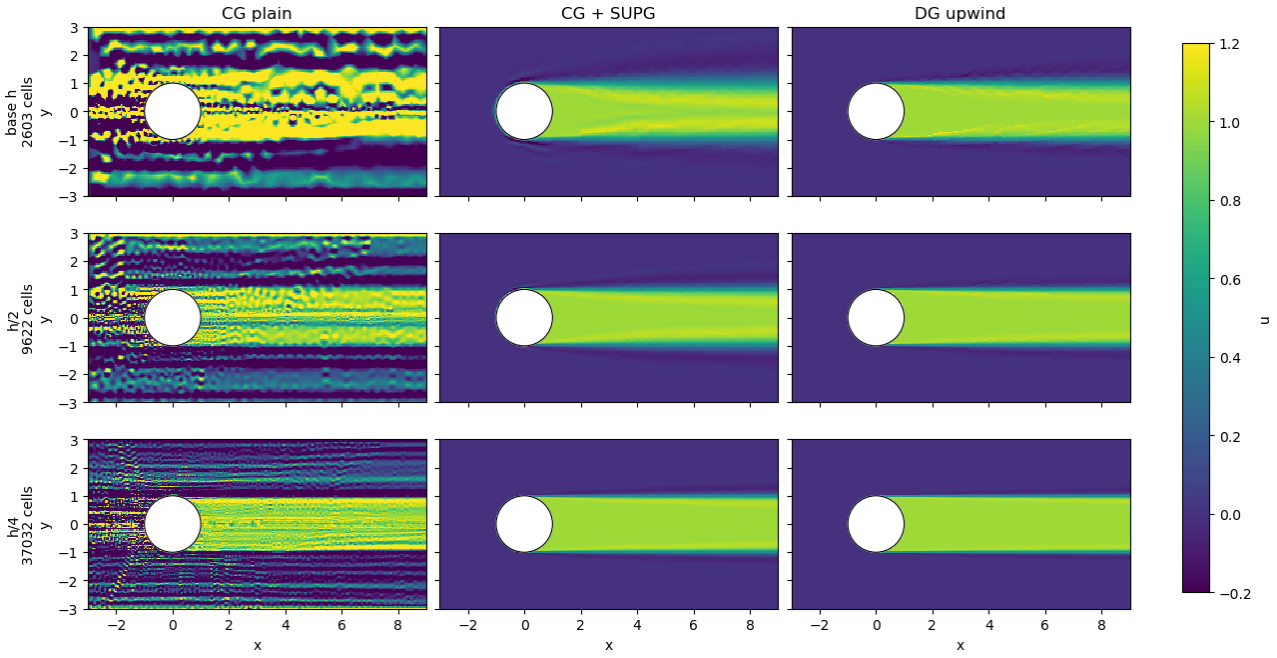


Figure 7.13.: Hemker benchmark under mesh refinement.

7.5.5. Other interesting examples

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

7.6. 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

8. Lecture 05 - Time integration

SSP and symplectic methods

8.1. Objectives

In this lecture, we will learn about advanced methods for solving time-dependent PDEs and ODEs. In particular, we will be discussing

1. Stability
2. Monotonicity
3. "Structure" preservation

8.2. ODEs: why?

Want to solve ODE

$$d\frac{u}{dt} = f(u,t), \quad u \in \mathbb{R}^n,\ t > 0, \quad u(0) = u^0.$$

ODEs come from - direct description of system (i.e. biology, chemical reactors) - semi-discretization of system (i.e. u — vector of cell values from a FVM discretization, RHS are the fluxes + sources)

> **Note**
>
> **Method of lines**: discretize in space, solve in time.
> Take PDE
> $$\partial_t u(x,t) + \sum_\alpha \partial_\alpha a_\alpha u(x,t) = f(u(x,t),t),$$
> discretize in space (FD/FVM/FEM/DG): $u(x_i,t) = u_i(t)$, discretize operators in terms of $\{u_j\}$, obtain ODEs
> $$\partial_t \mathbf{u} + \mathbf{A}\mathbf{u}(t) = \mathbf{f}(t).$$

8.2.1. PDE discretization - remark

> **!** Alternatives to Method of Lines
>
> Are all discretizations of time-dependent PDEs in time done via Method of Lines?

Lax-Wendroff: couples time and space discretization. For linear advection

$$\partial_t u + a_0 \partial_x u = 0,\ C = \frac{a_0 \Delta t}{\Delta x}$$

$$u_j^{n+1} = u_j^n + \frac{C}{2}(1+C)u_{j-1}^n + (1-C^2)u_j^n - \frac{C}{2}(1+C)u_{j+1}^n$$

8.2.2. ODE solvers: computational cost

8.2.3. ODE solvers: accuracy

Example 1: you want to model a system of chemical reactions. Which method would you choose?

1. Method with $\mathcal{O}(\Delta t^4)$ error, but still might lead to negative densities
2. Method with $\mathcal{O}(\Delta t^2)$ error, but preserves non-negativity of densities

Example 2: you want to model a system of particles. Which method would you choose?

1. Method with $\mathcal{O}(\Delta t^4)$ error, but does not conserve energy
2. Method with $\mathcal{O}(\Delta t^2)$ error, but conserves energy

We therefore will focus on different (not all!) notions of error/accuracy and talk about some problem-specific methods.

8.3. Recap of some standard methods

Let's recall some textbook methods first.

$$t_n = n\Delta t, \quad u^n = u(t_n).$$

8.3.1. Forward Euler

Simplest approach:

$$u^{n+1} = u^n + \Delta t f(u^n, t^n)$$

> **Properties?**
>
> $\mathcal{O}(\Delta t)$ error; what else?

8.3.2. RK4

$$k_1 = \Delta t f(u^n, t^n),$$

$$k_2 = \Delta t f(u^n + \frac{1}{2}k_1, t^n + \frac{1}{2}\Delta t),$$

$$k_3 = \Delta t f(u^n + \frac{1}{2}k_2, t^n + \frac{1}{2}\Delta t),$$

$$k_4 = \Delta t f(u^n + k_3, t^n \Delta t),$$

$$u^{n+1} = u^n + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4$$

Properties?

$\mathcal{O}((\Delta t)^4)$ error; what else?

8.4. Stability, monotonicity

Let's discuss stability and monotonicity. These are not identical properties!

8.4.1. Stability

Linear analysis:

$$u' = -\alpha u,\ \alpha \in \mathbb{C} > 0 \Rightarrow u(t) = u_0 e^{-\alpha t}.$$

Expect numerically for $\alpha : Re(\alpha) < 0$:

$$\lim_{n\to\infty} u^n = 0$$

I.e. solution does not "blow up".

Stability

For forward Euler: $\Delta t < \frac{2}{\alpha}$

8.4.2. Monotonicity

$$u(t) = u_0 e^{-\alpha t} : t_2 > t_1 \Rightarrow |u(t_1)| > |u(t_2)|$$

Expect numerically:

$$|u^n| \leq |u^{n-1}|$$

Monotonicity

For forward Euler: $\Delta t < \frac{1}{\alpha}$

Example where monotonicity may be desirable:

$$\mathbf{u} = (\rho, \rho v_1, \rho v_2, \rho e),$$

if ρ or ρe become negative - solver might crash or produce meaningless results!

Generalized stability/monotonicity

Consider $G(u^n)$ instead of $|u^n|$, where G is a 1) norm 2) semi-norm or 3) convex functional

8.4.3. Numerical example

$$u' = -\alpha u,\ \alpha > 0,\ u(0) = u_0$$

Note

For forward Euler: Monotonicity: $\Delta t < \frac{1}{\alpha}$, stability: $\Delta t < \frac{2}{\alpha}$.

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


def forward_euler(alpha, u0, times):
    u = np.zeros(len(times))
    u[0] = u0
    for (i, t) in enumerate(times[1:]):
        u[i+1] = u[i] + (times[i+1] - times[i]) * (-alpha * u[i])
    return u


fig, ax = plt.subplots(figsize=(6.5, 4.2))

u0 = 2.0

alpha = 10.0
times = np.linspace(0, 1, 100)

times_fe1 = np.linspace(0, 1, 5)
dt1 = times_fe1[1] - times_fe1[0]

times_fe2 = np.linspace(0, 1, 10)
dt2 = times_fe2[1] - times_fe2[0]

times_fe3 = np.linspace(0, 1, 12)
dt3 = times_fe3[1] - times_fe3[0]
```

```
28
29 dts = r"$\Delta t$"
30 oas = r"$1/\alpha$"
31 tas = r"$2/\alpha$"
32
33
34 ax.plot(times, u0 * np.exp(-alpha * times),
35         'k-', label=f"Analytical")
36 ax.plot(times_fe1, forward_euler(alpha, u0, times_fe1),
37         color='tab:red', label=f"F.E. ({dts} = {dt1:.3f}; {tas} = {2.0/alpha:.2f})")
38 ax.plot(times_fe2, forward_euler(alpha, u0, times_fe2),
39         color='tab:purple', label=f"F.E. ({dts} = {dt2:.3f}; {oas} = {1.0/alpha:.2f})")
40 ax.plot(times_fe3, forward_euler(alpha, u0, times_fe3),
41         color='tab:blue',
42         label=f"F.E. ({dts} = {dt3:.3f})")
43
44 ax.legend(fontsize=12)
45 ax.set_xlabel("t", fontsize=10)
46 ax.set_ylabel("u", fontsize=10)
47 ax.tick_params(axis='both', labelsize=8)
48
49 ax.set_yscale("log")
50 plt.show()
```

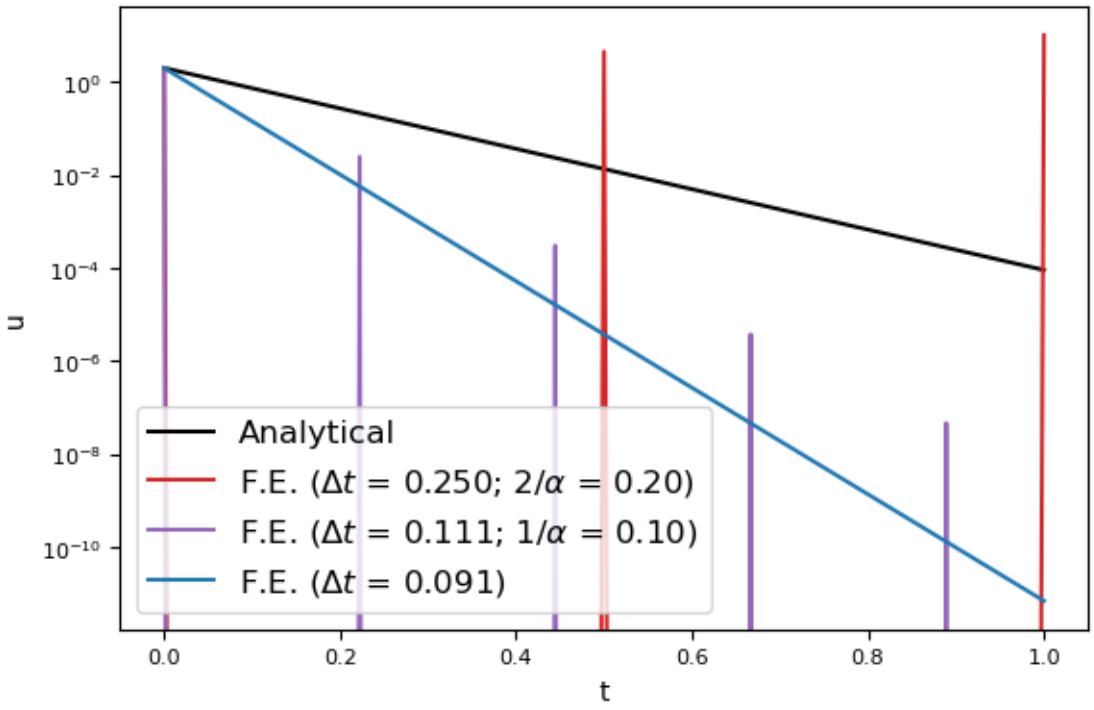


Figure 8.1.: Euler's method for $du/dt = -au$

8.4.4. Region of absolute stability

Re-write discretization of ODE ($-\alpha \to \lambda$)

$$u'(t) = \lambda u,\ u^n = R(\lambda \Delta t) u^{n-1},$$

$R(z), z \in \mathbb{C}$ is called the stability function.

Region of absolute stability:

$$\mathcal{S} = \{z \in \mathbb{C} || R(z)| < 1\}$$

8.4.5. Region of absolute stability: examples

Forward Euler:

$$R(z) = 1 + z$$

Backward (implicit Euler):

$$u^{n+1} = u^n + \Delta \lambda t u^{n+1} \implies R(z) = \frac{1}{1+z}$$

RK4:

$$R(z) = 1 + z + \frac{1}{2!}z^2 + \frac{1}{3!}z^3 + \frac{1}{4!}z^4$$

8.5. SSPRK methods

Strong stability-preserving (Total Variation Diminishing) Runge-Kutta methods:

- ensure strong stability/monotonicity of solutions **and** have higher order than forward Euler: $||u^{n+1}|| \leq ||u^n||$ (or a generalized $G(u)$) for all n
- *usually* written as convex combination of forward Euler steps

8.5.1. SSPRK methods: formulation

An m-stage method for $u' = f(u)$ is written as (*Shu-Osher form*)

$$u^{(0)} = u^n,$$

$$u^{(i)} = \sum_{j=0}^{i-1} \left(\alpha_{ij} u^{(j)} + \Delta t \beta_{ij} f(u^{(j)}) \right), \quad i = 1, \dots, m;\ \alpha_{ij} \geq 0,$$

$$u^{n+1} = u^{(m)}.$$

8.5.2. Theorem

Theorem

If $||u^{n+1}|| \leq ||u^n||$ in the forward Euler method with $\Delta \leq \Delta_{FE}$, then the SSPRK method is stable for $\Delta t \leq c_{SSP}\Delta_{FE}$.

- c_{SSP} is called the **SSP coefficient**, $c_{SSP} = \min \frac{\alpha_{ij}}{\beta_{ij}}$
- Most research focused on finding SSPRK methods with largest c_{SSP}
- Δ_{FE} - depends on spatial discretization, c_{SSP} - on time-stepping scheme

8.5.3. SSPRK examples

SSPRK2:

$$u^{(1)} = u^n + \Delta t f(u^n),$$

$$u^{(n+1)} = \frac{1}{2}u^n + \frac{1}{2}u^{(1)} + \frac{1}{2}\Delta t f(u^{(1)}).$$

SSPRK3:

$$u^{(1)} = u^n + \Delta t f(u^n),$$

$$u^{(2)} = \frac{3}{4}u^n + \frac{1}{4}u^{(1)} + \frac{1}{4}\Delta t f(u^{(1)}),$$

$$u^{(n+1)} = \frac{1}{3}u^n + \frac{2}{3}u^{(2)} + \frac{2}{3}\Delta t f(u^{(2)}).$$

8.5.4. SSP coefficient, effective CFL

Question

What does $c_{SSP} = 1$ (for example, SSPRK3) mean?

Makes more sense to talk about **effective CFL**: $c_{eff} = c_{SSP}/k$, k - number of evaluations of RHS. For example, SSPRK3 has $c_{eff} = 1/3$ (but third-order convergence!).

Stability vs. monotonicity vs. accuracy

SSPRK methods may have same restrictions on Δt to achieve monotonicity, but

- are more accurate
- have a larger stability region (monotone $\neq$ stable!)

8.6. Stability for oscillatory problems

Consider harmonic oscillator

$$\frac{d}{dt}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} u_2 \\ -\omega^2 u_1 \end{pmatrix}$$

Analytical solution:

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = c_1 \begin{pmatrix} \cos(\omega t) \\ -\omega \sin(\omega t) \end{pmatrix} + c_2 \begin{pmatrix} \frac{1}{\omega}\sin(\omega t) \\ \cos(\omega t) \end{pmatrix}$$

8.6.1. Forward Euler

Energy is given by

$$E = (\omega^2 u_1^2 + u_1^2)/2$$

and is conserved for the analytical solution.

> **Energy blow-up**
>
> For forward Euler we can show that $E^{n+1} = \frac{1}{2}(1+\omega^2\Delta t^2)\,(\omega^2 (u_1^n)^1 + (u_2^n)^2)$. Eigenvalues of harmonic oscillator system are purely imaginary ($\pm i\omega$) and do not lie in stability region of forward Euler.

8.6.2. Forward Euler: analysis

Let's write the forward Euler scheme as a matrix equation

$$\begin{pmatrix} u_1^{n+1} \\ u_2^{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \end{pmatrix} =: J\mathbf{u}$$

> **Determinant of J**
>
> Note that $det(J) = (1+\omega^2\Delta t^2)$ is exactly the energy growth factor!

Re-write integration step via **growth factor** γ:

$$u_1^{n+1} = \gamma u_1^n,\; u_2^{n+1} = \gamma u_2^n \implies \begin{pmatrix} \gamma - 1 & -\Delta t \\ \omega^2 \Delta t & \gamma - 1 \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

> **Exercise**
>
> Non-trivial solution $\implies$ $det(LHS) = 0$; can show that $|\gamma| > 1$, i.e. method is unstable.

8.6.3. Example - harmonic oscillator

```
def forward_euler_generic(rhs, u0, times, omega):
    u = np.zeros((len(times), len(u0)))
    u[0,:] = u0
    for (i, t) in enumerate(times[1:]):
        u[i+1,:] = u[i,:] + (times[i+1] - times[i]) * rhs(u[i,:], omega)
    return u

def ssprk22_generic(rhs, u0, times, omega):
    u = np.zeros((len(times), len(u0)))
    u[0,:] = u0
    for (i, t) in enumerate(times[1:]):
        u_1 = u[i,:] + (times[i+1] - times[i]) * rhs(u[i,:], omega)
        u[i+1,:] = 0.5 * u[i,:]+ 0.5 * u_1 + 0.5 * (times[i+1] - times[i]) * rhs(u_1, omega)
    return u

def rk2_generic(rhs, u0, times, omega):
    # Ralston's method
    u = np.zeros((len(times), len(u0)))
    u[0,:] = u0
    for (i, t) in enumerate(times[1:]):
        dt = (times[i+1] - times[i])
        k1 = rhs(u[i,:], omega)
        k2 = rhs(u[i,:] + 0.75 * k1 * dt, omega)
        u[i+1,:] = u[i,:] +  dt * (0.25 * k1 + 0.75 * k2)
    return u

def harmonic_rhs(u, omega):
    return np.array([u[1], -omega**2 * u[0]])

def energy(u, omega):
    return 0.5 * (u[:,1]**2 + omega**2 * u[:,0]**2)

fig, axes = plt.subplots(figsize=(12.0, 4.2), ncols=2)

ax = axes[0]
ax_e = axes[1]

t_max = 5.0
omega = 3.0
times = np.linspace(0, t_max, 100)

u0 = [0.5, 0.5]

analytical = np.array([u0[0] * np.cos(omega * times) + (u0[0]/omega) * np.sin(omega * times),
                       -u0[0] * omega * np.sin(omega * times) + u0[0] * np.cos(omega * times)]



times_fe1 = np.linspace(0, t_max, 100)
dt1 = times_fe1[1] - times_fe1[0]
```

```
sol_fe1 = forward_euler_generic(harmonic_rhs, u0, times_fe1, omega)

times_fe2 = np.linspace(0, t_max, 400)
dt2 = times_fe2[1] - times_fe2[0]
sol_fe2 = forward_euler_generic(harmonic_rhs, u0, times_fe2, omega)

times_ssprk2 = np.linspace(0, t_max, 80)
dt3 = times_ssprk2[1] - times_ssprk2[0]
sol_ssprk2 = ssprk22_generic(harmonic_rhs, u0, times_ssprk2, omega)

times_rk2 = np.linspace(0, t_max, 80)
dt4 = times_rk2[1] - times_rk2[0]
sol_rk2 = rk2_generic(harmonic_rhs, u0, times_rk2, omega)


ax.plot(sol_fe1[:,0], sol_fe1[:,1],
        color='tab:red', label=f"F.E. dt = {dt1:.3f}")

ax.plot(sol_fe2[:,0], sol_fe2[:,1],
        color='tab:purple', label=f"F.E. dt = {dt2:.3f}")

ax.plot(sol_ssprk2[:,0], sol_ssprk2[:,1],
        color='tab:blue', label=f"SSPRK2 dt = {dt3:.3f}")

ax.plot(sol_rk2[:,0], sol_rk2[:,1],
        color='tab:green', label=f"RK2 dt = {dt4:.3f}")


ax.plot(analytical[:,0], analytical[:,1],
        'k-', label=f"Analytical")

ax_e.plot(times, energy(analytical, omega), 'k')
ax_e.plot(times_fe1, energy(sol_fe1, omega), 'tab:red')
ax_e.plot(times_fe2, energy(sol_fe2, omega), 'tab:purple')
ax_e.plot(times_ssprk2, energy(sol_ssprk2, omega), 'tab:blue')
ax_e.plot(times_rk2, energy(sol_rk2, omega), 'tab:green')

ax.legend(fontsize=12)
ax.set_xlabel("x(t)", fontsize=10)
ax.set_ylabel("v(t)", fontsize=10)
ax.tick_params(axis='both', labelsize=8)


print(f"dE F.E. (dt={dt1:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_fe1,
print(f"dE F.E. (dt={dt2:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_fe2,
print(f"dE SSPRK2 (dt={dt3:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_ss
print(f"dE RK2 (dt={dt4:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_rk2,

ax_e.set_yscale("log")
plt.show()
```

```
dE F.E. (dt=0.051) at t=5.0: -1.057e+01
dE F.E. (dt=0.013) at t=5.0: -9.460e-01
dE SSPRK2 (dt=0.063) at t=5.0: -3.250e-02
dE RK2 (dt=0.063) at t=5.0: 3.461e-01
```

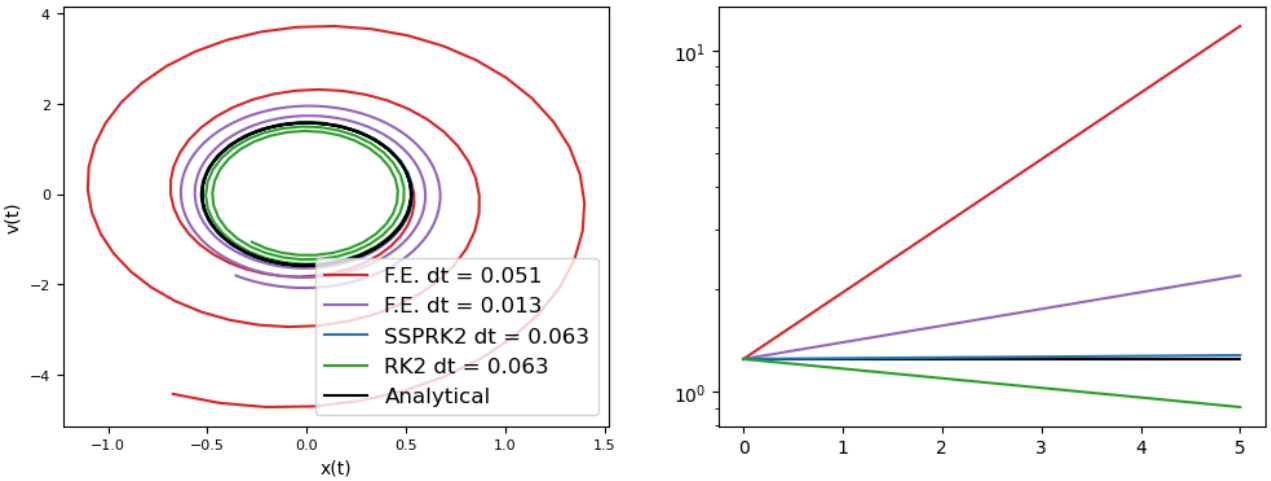


Figure 8.2.: Harmonic oscillator phase space (left) and energy (right), omega=3

> **Question**
>
> Why does SSPRK2 still cause a growth of energy?

8.7. Stiffness

We don't go into detail about stiff ODEs in this lecture, but still mention some definitions.

- **Definition 1** Step size required for stability is much smaller than the step size required for accuracy
- **Definition 2** Computational cost of an explicit method becomes larger than that of an implicit one

Arise in biology, chemistry, low-Mach number compressible flows.

8.8. Symplectic integrators

Consider ODE system given by

$$\frac{dq_i}{dt} = -\frac{\partial H}{\partial p_i}, \quad \frac{dp_i}{dt} = \frac{\partial H}{\partial q_i},$$

where H is the **Hamiltonian** of the system.

- Why do are we interested in specific methods for such systems?
- What can these systems describe?

Naming

q_i are called "generalized coordinates", p_i - "generalized momenta"

8.8.1. Example: interacting particles

$$H(p,q) = \frac{1}{2}\frac{1}{m}\sum_i p_i^2 + \sum_{i<j} V(|q_i - q_j|),$$

V - *interaction potential*. Examples: gravity, electrostatic potential (both $V(r)$ are proportional to $1/r$). Potential gives force:

$$F = -\nabla V.$$

Hamiltonian of interacting particles?

What does Hamiltonian resemble?

Hamiltonian is kinetic + potential energy! Time integration of particle movement should conserve total energy.

8.8.2. First integrals

Cosider the system

$$\frac{dy}{dt} = f(y).$$

$I(y)$ is called a **first integral** of a system, if

$$\frac{dI}{dt} = \frac{dI}{dy}f(y) = 0.$$

H as first integral

The Hamiltonian is a first integral of a Hamiltonian system!

8.8.3. Symplectic integrators

Define **mapping** ψ:

$$\psi : (\mathbf{q}(t), \mathbf{p}(t)) \to (\mathbf{q}(t + \Delta t\mathbf{p}), \mathbf{p}(t + \Delta t))$$

Theorem Symplectic iff.

$$(D\psi)^T \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} (D\psi) = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix},$$

$D\psi$ is the Jacobi matrix of the mapping (ψ - "canonical transformation").

8.8.4. Phase space volume-preserving integrators

A weaker condition: $\mu(\psi(\mathcal{S})) \equiv \mu(\mathcal{S})$, where μ is a measure (volume). This implies that the Jacobi matrix $D\psi$ has determinant 1.

What does this give?

- Ensures no spurious sources/sinks/attractors
- Ensures that time-averaging remains equivalent to ensemble averaging for a certain class of systems (*ergodic systems*): important for particle methods!

> **i** Difference between symplectic and volume-preserving mappings:
>
> The non-squeezing theorem explains the difference nicely and shows that symplecticity is a stronger condition.

8.8.5. Leapfrog integrator

If $H(\mathbf{p}, \mathbf{q}) = T(\mathbf{p}) + V(\mathbf{q})$ (*separable Hamiltonian*), then we can integrate with **leapfrog**:

$$\mathbf{p}^{n+1/2} = \mathbf{p}^n - \Delta t \nabla V(\mathbf{q}^n),$$
$$\mathbf{q}^{n+1} = \mathbf{q}^n + \Delta t \nabla T(\mathbf{p}^{n+1/2}),$$

Leapfrog is **symplectic** and has error $\mathcal{O}((\Delta t)^2)$.

> **i** Harmonic oscillator
>
> One can show that stability for harmonic oscillator requires $\Delta t \leq 2/\omega$; this can be done using the growth factor analysis (keeping in mind that a step of $\Delta t/2$ corresponds to a growth factor of $\sqrt{\gamma}$ and re-writing $u_2^{n+1/2}$ and $u_2^{n-1/2}$ in terms of u_2^n).

8.8.6. Verlet integrator

What if we need $\mathbf{q}$, $\mathbf{p}$ at the same point in time?

$$\mathbf{p}^{n+1/2} = \mathbf{p}^n - \frac{\Delta t}{2} \nabla V(\mathbf{q}^n),$$
$$\mathbf{q}^{n+1} = \mathbf{q}^n + \Delta t \nabla T(\mathbf{p}^{n+1/2}),$$
$$\mathbf{p}^{n+1} = \mathbf{p}^{n+1/2} - \frac{\Delta t}{2} \nabla V(\mathbf{q}^{n+1}),$$

> **i** Naming
>
> Often this is called “Leapfrog in kick-drift-kick” form or “Störmer-Verlet”.

8.8.7. Example - harmonic oscillator revisited

```
def verlet_generic(rhs, u0, times, omega):
    u = np.zeros((len(times), len(u0)))
    u[0,:] = u0
    for (i, t) in enumerate(times[1:]):
        dt = (times[i+1] - times[i])
        p_hs = u[i,1] + 0.5 * dt * rhs(u[i,:], omega)[1]
        u[i+1,0] = u[i,0] + dt * p_hs
        u[i+1,1] = p_hs + 0.5 * dt * rhs(u[i+1,:], omega)[1]
    return u

fig, axes = plt.subplots(figsize=(12.0, 4.2), ncols=2)

ax = axes[0]
ax_e = axes[1]

times_verlet = np.linspace(0, t_max, 20)
dt5 = times_verlet[1] - times_verlet[0]
sol_verlet = verlet_generic(harmonic_rhs, u0, times_verlet, omega)

ax.plot(sol_fe2[:,0], sol_fe2[:,1],
        color='tab:purple', label=f"F.E. dt = {dt2:.3f}")

ax.plot(sol_ssprk2[:,0], sol_ssprk2[:,1],
        color='tab:blue', label=f"SSPRK2 dt = {dt3:.3f}")

ax.plot(sol_verlet[:,0], sol_verlet[:,1],
        color='tab:red', label=f"Verlet dt = {dt5:.3f}; 2/omega={2/omega:.3f}")

ax.plot(analytical[:,0], analytical[:,1],
        'k-', label=f"Analytical")

ax_e.plot(times, energy(analytical, omega), 'k')
ax_e.plot(times_fe2, energy(sol_fe2, omega), 'tab:purple')
ax_e.plot(times_ssprk2, energy(sol_ssprk2, omega), 'tab:blue')
ax_e.plot(times_verlet, energy(sol_verlet, omega), 'tab:red')

ax.legend(fontsize=12, loc="upper right")
ax.set_xlabel("x(t)", fontsize=10)
ax.set_ylabel("v(t)", fontsize=10)
ax.tick_params(axis='both', labelsize=8)


print(f"dE F.E. (dt={dt2:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_fe2,
print(f"dE SSPRK2 (dt={dt3:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_ss
print(f"dE Verlet (dt={dt5:.3f}) at t={t_max}: {(energy(analytical, omega)[-1] - energy(sol_ve

# ax_e.set_yscale("log")
plt.show()
```

```
dE F.E. (dt=0.013) at t=5.0: -9.460e-01
dE SSPRK2 (dt=0.063) at t=5.0: -3.250e-02
dE Verlet (dt=0.263) at t=5.0: 4.697e-02
```

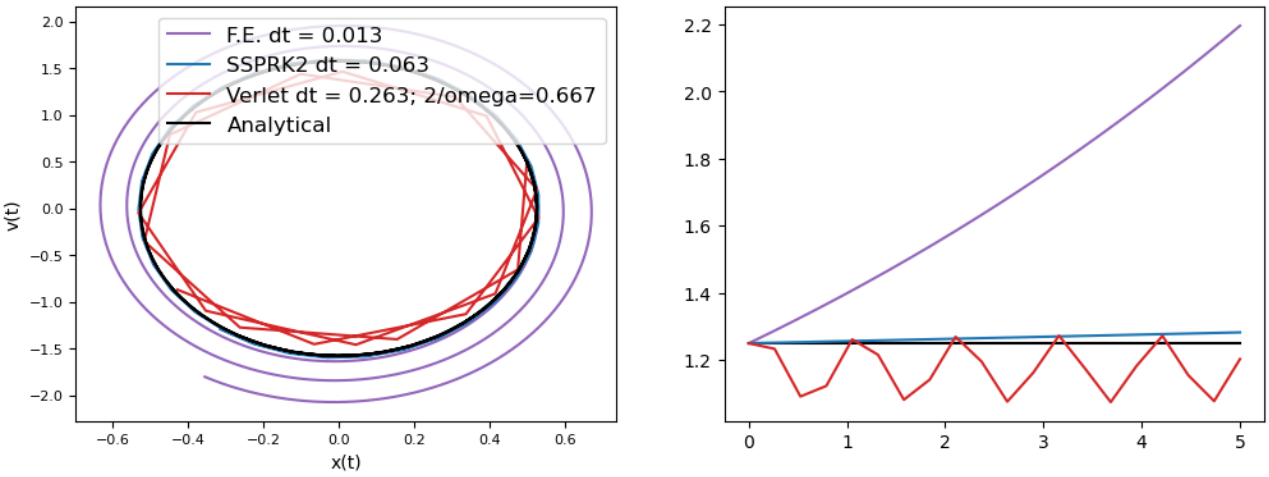


Figure 8.3.: Harmonic oscillator phase space (left) and energy (right), omega=3

8.9. Code

- DifferentialEquations.jl - huge amount of methods available
- Comparison of various methods (2019!!!) (see also summary table)

8.10. Summary

Methods not covered in this lecture but also useful/actively developed:

- IMEX (IMplicit/EXplicit): separate RHS into stiff and non-stiff parts, treat them implicitly and explicitly, respectively.
  - "Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations", Ascher, Ruth et al.
  - "Additive Runge–Kutta schemes for convection–diffusion–reaction equations", Kennedy & Carpenter
- MPRK (Modified Patankar Runge–Kutta) for production-destruction systems. Ensure non-negativity of densities.
  - "On order conditions for modified Patankar–Runge–Kutta schemes", Kopecz & Meister
  - Implementation available in PositiveIntegrators.jl
- Explicit integrators for particles with velocity-dependent force terms (for example, Lorentz force)
  - "On the Boris solver in particle-in-cell simulation", Zenitani & Umeda

8.10.1. Exercises

1. Derive the stability polynomial $R(z)$ for the RK4 method
2. Prove the SSP property of a method in Shu-Osher form (hint: use the TVD property of forward Euler, the triangle inequality and look at the definition of c_{SSP})
3. Derive energy growth rate for SSPRK2 for the harmonic oscillator
4. Show that forward Euler is **always** unstable for the harmonic oscillator via growth factor analysis
5. Show that the forward Euler method is not a symplectic method

8.10.2. Questions

1. What is the difference between monotonicity and stability of a method?
2. Why are implicit methods oftentimes more computationally expensive?
3. What is the difference between a symplectic and a phase space volume-preserving integrator? Which condition is stronger? Why?
4. What two formulations of explicit symplectic integrators do you know and what is the difference between them?

8.11. References

1. S. Gottlieb, On High Order Strong Stability Preserving Runge–Kutta and Multi Step Time Discretizations
2. Kubatko, Yeager et al., Optimal Strong-Stability-Preserving Runge–Kutta Time Discretizations for Discontinuous Galerkin Methods
3. A. Giorgilli, Notes on Hamiltonian Dynamical Systems

Part II.

Exercises

9. Project 00 – Poisson Solver with Finite Differences

Exercise Sheet

9.1. Overview

The exercise here aims to provide an overview of setting up a project repository together with testing, documentation, and continuous integration, along with a small refresher on finite difference methods. We want to solve the 2-dimensional Poisson equation

$$-\Delta u = f \quad \text{in} \quad \Omega = (0,1)^2$$

with the Dirichlet boundary condition

$$u = g \quad \text{on} \quad \partial\Omega.$$

We will discretize the domain with a uniform mesh with $n+2 \times n+2$ nodes, leading to a grid spacing

$$h = 1.0/(n+1).$$

Why $n+2$ nodes? Because we need to include the boundary nodes, but the values of u are fixed on the boundary due to the Dirichlet condition, so we do not need to solve for them.

We denote the discrete solution by $u_{i,j}$ for $i, j = 0, \ldots n+1$ - these are the **point** values of u at position $(x_i, y_j) = (ih, jh)$ (in other methods, the values are oftentimes not point values, but rather cell-averaged values).

The discrete Laplacian (which we denote by Δ_h) at a node (i, j) is given by

$$(\Delta_h u)_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2}.$$

In the interior of the domain, we can write the discrete PDE as

$$-(\Delta_h u)_{i,j} = f_{i,j}.$$

At the boundaries, we can use the Dirichlet boundary conditions $u_{i,j} = g_{i,j}$ to write and close the system. By "unrolling" the $n \times n$ matrix of unknown values of u, we can write the system as a linear system $A\mathbf{u} = \mathbf{b}$, where A is a sparse matrix of size $n^2 \times n^2$.

$$A = \frac{1}{h^2}\begin{bmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{bmatrix},$$

where T is of size $n \times n$:

$$T = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix}.$$

The RHS vector is given by

$$b = \begin{bmatrix} f_{1,1} \\ f_{2,1} \\ \vdots \\ f_{n,n} \end{bmatrix} + \frac{1}{h^2}\begin{bmatrix} g_{0,1} + g_{1,0} \\ g_{2,0} \\ \vdots \\ g_{n+1,n} + g_{n,n+1} \end{bmatrix},$$

where $f_{i,j} = f(x_i, y_j)$ and $g_{i,j} = g(x_i, y_j)$ (the function g is evaluated only on the boundaries of the domain).

So we can solve the linear system and obtained a discretized solution. Let us now implement this in code.

9.2. Julia

Julia is a just-in-time (JIT) compiled language that aims for a good combination of readability and efficiency, whilst allowing for modular reusable code via its flexible type system. The Julia programming language can be installed from the official website.

Calling `julia` in a terminal window will bring up the Julia REPL, which can be used for rapid prototyping and testing.

9.2.1. Setting up a Julia project

First, we create the working directory for our project, which we will call `JuFD26`.

This can be done by either

- calling `julia -e 'using Pkg; Pkg.generate("JuFD26")'`
- bringing up the Julia REPL, typing `]`, pressing Enter (this opens the Package manager interface), typing `generate JuFD26` and pressing Enter

Nex, navigate to the project directory:

```
cd JuFD26
```

- Add CairoMakie as a dependency:
  - Open the Julia REPL by typing `julia` in the terminal.
  - Press `]` to enter the package manager mode.

  - Add the dependency:

```
add CairoMakie
```

  - P.S. some packages are not registered on JuliaHub, can be installed directly via Github link: `add https://github.com/rdeits/NNLS.jl`

- Commit `Project.toml` and add `Manifest.toml` to `.gitignore`:
  - `Project.toml` contains the project's dependencies and should be committed to version control.
  - `Manifest.toml` contains the exact versions of dependencies and should not be committed to avoid conflicts. Add it to `.gitignore`:

```
echo "Manifest.toml" >> .gitignore
git add Project.toml .gitignore
git commit -m "Add Project.toml and update .gitignore"
```

- Set up the `test` directory and add the `Test` package:
  - Create a `test` directory for your tests:

```
mkdir test
cd test/
```

  - Activate the test project environment by entering the Julia REPL and adding the `Test` and `LinearAlgebra` packages:

```
using Pkg
Pkg.activate(".")
Pkg.add("Test")
Pkg.add("LinearAlgebra")
```

- Create `runtests.jl` and add an empty `@testset`:
  - Create a file named `runtests.jl` in the `test` directory:

```
touch runtests.jl
```

  - Add an empty `@testset` to `runtests.jl` start writing tests:

```
using JuFD26
using Test
using LinearAlgebra

@testset "Poisson Solver Tests" begin
    # Tests will go here
end
```

- Verify the setup by running the tests:
  - Go back to the root directory of the project
  - Enter Julia repl using current environment: `julia --project=.` (alternatively, simply enter the Julia REPL by calling `julia` and activate the environment by using `using Pkg; Pkg.activate(".")`)
  - Run the tests using the `Test` package: `Pkg.test()` (alternatively, enter the package manager by entering `]` and run `test`)

The expected output looks like this:

```
     Testing Running tests...
Test Summary:         | Total  Time
Poisson Solver Tests |     0  0.0s
     Testing JuFD26 tests passed
```

9.2.2. Adding code

First, we need to add some additional packages for linear algebra. Although they are part of the standard library, if we add them explicitly, their specific versions will be recorded in the `Project.toml` file. This is useful for reproducibility.

- Add the `LinearAlgebra` package to the project by entering the package manager and typing `add LinearAlgebra`
- Add the `SparseArrays` package to the project by entering the package manager and typing `add SparseArrays`

Now, create a file `src/solver.jl` and add the following code:

```julia
function construct_matrix(n::Int)
    h = 1.0 / (n + 1)
    N = n * n  # number of unknowns

    # Create sparse matrix with 5-point stencil
    A = spdiagm(
         0 => fill(4.0, N),       # diagonal
    )

    # Add the off-diagonal terms for the 2D Laplacian
    for i in 1:n
        for j in 1:n
            idx = (i-1) * n + j
            if j < n
                A[idx, idx+1] = -1.0
                A[idx+1, idx] = -1.0
            end
            if i < n
                A[idx, idx+n] = -1.0
                A[idx+n, idx] = -1.0
            end
        end
    end

    # Scale by 1/h^2
    A = A / (h^2)

    return A
end

function construct_rhs(f, g, n::Int)
```

```
    h = 1.0 / (n + 1)
    invh2 = 1.0 / (h^2)
    b = zeros(n * n)

    # Interior points
    for i in 1:n
        x = i * h
        for j in 1:n
            y = j * h
            idx = (i - 1) * n + j
            b[idx] = f(x, y)
        end
    end

    # Boundary conditions
    for k in 1:n
        val = k * h
        # Bottom (i=0) & Top (i=n+1)
        b[k] += g(0.0, val) * invh2           # idx = (1-1)*n + k
        b[(n-1)*n + k] += g(1.0, val) * invh2 # idx = (n-1)*n + k

        # Left (j=0) & Right (j=n+1)
        b[(k-1)*n + 1] += g(val, 0.0) * invh2
        b[(k-1)*n + n] += g(val, 1.0) * invh2
    end

    return b
end

function solve_poisson(f, g, n::Int)
    # Construct matrix and RHS
    A = construct_matrix(n)
    b = construct_rhs(f, g, n)

    # Solve the linear system
    u = A \ b

    return u
end

function reshape_solution(u::Vector, n::Int)
    return reshape(u, n, n)
end
```

Now we need to export functions from the module. Edit the `src/JuFD26.jl` file to read the following:

```
module JuFD26

include("solver.jl")

export solve_poisson
```

```
export reshape_solution

end
```

9.2.3. Writing tests

Now we need to write some actual tests for our code. First, let's test the construction of the matrix:

```
@testset "Poisson Matrix Test" begin
    A = JuFD26.construct_matrix(3)

    @test size(A) == (9, 9)

    h = 1.0 / 4.0

    tested_entries = []

    # test diagonal terms
    for i in 1:9
        @test isapprox(A[i, i], 4.0/h^2; rtol=eps())
        push!(tested_entries, (i,i))
    end

    # test off-diagonal terms
    for i in 1:6
        @test isapprox(A[3+i, i], -1.0/h^2; rtol=eps())
        @test isapprox(A[i, i+3], -1.0/h^2; rtol=eps())
        push!(tested_entries, (3+i, i))
        push!(tested_entries, (i, i+3))
    end

    # test -1 terms in blocks
    for i in 1:3

        # upper part
        @test isapprox(A[(i-1)*3+1, (i-1)*3+2], -1.0/h^2; rtol=eps())
        @test isapprox(A[(i-1)*3+2, (i-1)*3+3], -1.0/h^2; rtol=eps())
        push!(tested_entries, ((i-1)*3+1, (i-1)*3+2))
        push!(tested_entries, ((i-1)*3+2, (i-1)*3+3))

        # lower part
        @test isapprox(A[(i-1)*3+2, (i-1)*3+1], -1.0/h^2; rtol=eps())
        @test isapprox(A[(i-1)*3+3, (i-1)*3+2], -1.0/h^2; rtol=eps())
        push!(tested_entries, ((i-1)*3+2, (i-1)*3+1))
        push!(tested_entries, ((i-1)*3+3, (i-1)*3+2))
    end

    n_tested = 0

    # check that all other entries are zero
```

```
    for i in 1:9
        for j in 1:9
            if !((i,j) in tested_entries)
                @test isapprox(A[i,j], 0.0; rtol=eps())
                n_tested += 1
            end
        end
    end

    @test n_tested + length(tested_entries) == 9*9
end
```

Here we constructed a small matrix and explicitly tested the values of all of its elements. Next, let us apply a similar approach to the RHS vector; we will test two versions: one with 0-values BCs, and one with 0-valued source term.

Add the following lines to the `runtests.jl` file:

```
@testset "Poisson RHS Test" begin

    f = (x,y) -> 2.0  # RHS function
    g = (x,y) -> 0.0  # BC

    h = 1.0 / 5.0

    b = JuFD26.construct_rhs(f, g, 4)

    @test length(b) == 4*4
    @test isapprox(maximum(abs.(b .- 2.0)), 0.0; rtol=eps())

    f = (x,y) -> 0.0  # RHS function
    g = (x,y) -> 1.0  # BC

    h = 1.0 / 5.0

    b = JuFD26.construct_rhs(f, g, 4)

    @test length(b) == 4*4

    for i in 2:3
        @test b[(i-1)*4+1:i*4] == [1.0, 0.0, 0.0, 1.0] / h^2
    end

    for i in [1,4]
        @test b[(i-1)*4+1:i*4] == [2.0, 1.0, 1.0, 2.0] / h^2
    end
end
```

Finally, let's test an the full solver by comparing it to the analytical solution. We set the BCs to 0, and the source term to $f = \sin(\pi x) sin(\pi y)$, this will give an analytical solution $u = \frac{1}{2\pi^2} \sin(\pi x) \sin(\pi y)$. We compute the numerical solution on a series of grids, each one finer than the previous, and check

that the error is reduced by a factor larger than 3, which is expected for a second order method (we expect a factor of 4 convergence ideally). We also check that on the finest grid, the error is less than 10^{-4}.

Add the following lines to the `runtests.jl` file:

```
@testset "Poisson analytical solution test" begin
    f = (x, y) -> sin(pi*x) * sin(pi*y)
    g = (x,y) -> 0.0  # BC

    n_grid = [4,8,16,32]
    errors = []

    for n in n_grid
        u = solve_poisson(f, g, n)

        @test length(u) == n^2

        u_mat = reshape_solution(u, n)
        @test size(u_mat) == (n,n)

        u_analytical_mat = [(1.0/(2.0 * pi^2)) * sin(pi*x)*sin(pi*y) for x in range(0,1,length

        # we discard the boundary values
        push!(errors, norm(u_mat - u_analytical_mat[2:end-1, 2:end-1], Inf))
    end

    # in theory should have convergence rate 4 (2nd order + doubling of grid resolution)
    convergence = [errors[i]/errors[i+1] for i in 1:length(errors)-1]
    @test all(convergence .>= 3.0)

    @test errors[end] < 1e-4
end
```

Running the tests should produce the following output:

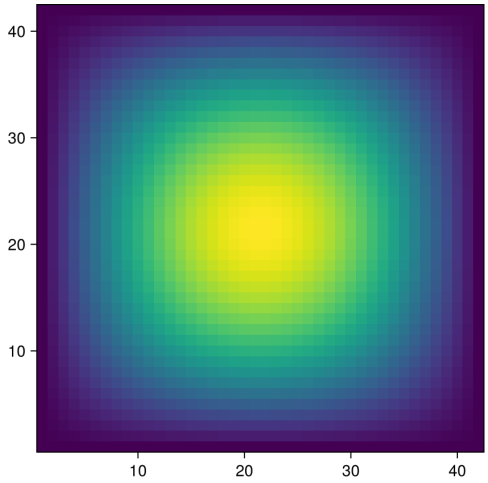
```
     Testing Running tests...
Test Summary:       | Pass  Total  Time
Poisson Matrix Test |   83     83  0.1s
Test Summary:     | Pass  Total  Time
Poisson RHS Test |    7      7  0.2s
Test Summary:                     | Pass  Total  Time
Poisson analytical solution test |   10     10  2.0s
     Testing JuFD26 tests passed
```

9.2.4. Using the code

Now, if we want to use the developed Poisson solver, we can do the following:

- Start the Julia REPL with the project environment activated: `julia --project=.`
- Load the package: `using JuFD26`

- Load CairoMakie for plotting: `using CairoMakie`
- Define our BC and source term functions: `f = (x, y) -> sin(pi*x) * sin(pi*y); g = (x,y) -> 0.0`
- Define the number of grid points in each direction in the interior: `n_grid = 40`
- Solve the Poisson equation: `u = solve_poisson(f, g, n)`
- Convert the solution to a matrix: `u_mat = reshape_solution(u, n_grid)`
- `u_mat` does not include the values at the boundaries, so we construct the full field: `u_full = zeros(n_grid+2,n_grid+2)`
- We fill it with `u_mat` values in the interior: `u_full[2:end-1,2:end-1] = u_mat`
- We then fill the boundary values of `u_mat`:
  - `h = 1.0 / (n_grid + 1); u_full[1,:] = [g((i-1)*h, 0.0) for i in 1:n_grid+2]`
  - `u_full[end,:] = [g((i-1)*h, 1.0) for i in 1:n_grid+2]`
  - `u_full[:,1] = [g(0.0, (i-1)*h) for i in 1:n_grid+2]`
  - `u_full[:,end] = [g(1.0, (i-1)*h) for i in 1:n_grid+2]`
- Plot the solution: `fig = Figure(); ax = Axis(fig[1,1], aspect = 1); heatmap!(ax, u_full); fig`



9.2.5. Adding documentation

Next, let's add documentation to the functions in the project; see also the Documenter.jl guide. First, create a `docs` directory. Assuming you are using Julia v.1.12+, add the following to the `Project.toml` file:

```
[workspace]
projects = ["docs"]
```

Next, start a Julia REPL in the `docs` by calling `julia --project=docs`, enter `pkg>` mode with the `]` key and install `Documenter` and `JuFD26` by typing `add Documenter` and `dev .` (adding development version of base package). Add a `src` subdirectory to `docs` and create a `make.jl` file. in `docs` with the following code:

```
using Documenter, JuFD26

makedocs(sitename="JuFD26", remotes = nothing)
```

Trying to run this (by navigating to `docs`, and running `julia --project=. make.jl`) will fail, since no files are present in `src/`, but we can now add documentation to the functions in the project.

Add the following docstrings before the functions in `src/solver.jl`:

```
"""
    construct_matrix(n::Int)

Construct the finite difference matrix for the 2D Poisson equation on [0,1]x[0,1]
with Dirichlet boundary conditions.

Parameters:
- n: number of interior points in each direction (total grid is (n+2)x(n+2))
- h: grid spacing (1/(n+1))

Returns:
- A: sparse matrix representing the discrete Laplacian
"""



"""
    construct_rhs(f, g, n::Int)

Construct the right-hand side vector for the Poisson equation.

Parameters:
- f: function f(x,y) giving the RHS
- g: function g(x,y) giving Dirichlet boundary conditions
- n: number of interior points in each direction
- h: grid spacing (1/(n+1))

Returns:
- b: vector of length n*n containing the RHS values
"""

"""
    solve_poisson(f, g, n::Int)

Solve the 2D Poisson equation -Δu = f with Dirichlet boundary conditions u=g.

Parameters:
- f: function f(x,y) giving the RHS
- g: function g(x,y) giving Dirichlet boundary conditions
- n: number of interior points in each direction

Returns:
- u: solution vector of length n*n
"""

"""
    reshape_solution(u::Vector, n::Int)
```

```
Reshape the solution vector into a 2D grid for visualization.

Parameters:
- u: solution vector of length n*n
- n: number of interior points in each direction

Returns:
- U: n x n matrix representing the solution
"""
```

Next, create a file `docs/src/index.md` and add the following to the file:

````
# JuFD26

This is a finite difference code for solving the Poisson equation on a unit square.

## Public API

```@docs
solve_poisson
reshape_solution
```

## Private API

```@docs
JuFD26.construct_matrix
JuFD26.construct_rhs
```
````

Modify the `docs/make.jl` file to include the following:

```
makedocs(sitename="JuFD26", remotes = nothing,
         pages = [
        "Home" => "index.md",
        ])
```

You can have multiple pages, but for now, one is sufficient. Run `julia --project=. make.jl` from `docs`, and a `build` directory should be produced. To view the documentation, open `docs/build/index.html` in your browser (or alternatively, navigate to `docs/build`, run `python3 -m http.server`, and open the link).

9.2.6. Setting up Gitlab CI

First, make sure that `Manifest.toml` and `docs/build` have been added to `.gitignore`.

Next, you need to create a `.gitlab-ci.yml` file in the root directory of your project. This file will contain the instructions for Gitlab CI to build your documentation. You also need an NRW Gitlab account - create an empty remote repository there, and push the JuFD26 code to it.

Add the following to `.gitlab-ci.yml`:

```
stages:          # List of stages for jobs, and their order of execution
  - test

default:
  image: julia:1.12

unit-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build stage completes successfully.
  script:
    - echo "Running unit tests..."
    # Let's run the tests. Substitute `coverage = false` below, if you do not
    # want coverage results.
    - julia -e 'using Pkg; Pkg.activate("."); Pkg.instantiate; Pkg.test(coverage = true)'
    # Comment out below if you do not want coverage results.
    - julia -e 'using Pkg; Pkg.activate("."); Pkg.add("Coverage");
      using JuFD26; cd(joinpath(dirname(pathof(JuFD26)), ".."));
      using Coverage; cl, tl = get_summary(process_folder());
      println("(", cl/tl*100, "%) covered")'
```

You can test the pipeline by pushing to the NRW Gitlab.

Now we add another stage to `stages` which we will name `deploy`:

```
stages:          # List of stages for jobs, and their order of execution
  - test
  - deploy
```

We will add a job to the `deploy` stage which will build and deploy the documentation:

```
build-docs-job:
  stage: deploy
  script:
      - julia -e 'using Pkg; Pkg.activate("."); Pkg.instantiate; Pkg.add("Documenter")';
      - julia --color=yes --project=. docs/make.jl # make documentation
      - mv docs/build public # move to the directory picked up by Gitlab pages
  artifacts:
    paths:
      - public
  only:
      - main
  pages: true
```

Once the pipeline has been successfully executed, under Deploy/Pages in Gitlab one should see an active deployment.

An example of the project can be found at: https://gitlab.git.nrw/georgii.oblapenko/jufd26/ The gitlab pages documentation can be found at: https://jufd26-57cab6.pages.git.nrw

9.3. Python

A similar approach can be taken in Python, see https://gitlab.git.nrw/rwth-acom/teaching/pyfd26 for an example repository.

9.3.1. Structure

The structure of the repository is as follows:

```
.
   docs
       api.md
       index.md
   examples
       plot_two_source_terms.py
       README.md
   mkdocs.yml
   pyproject.toml
   README.md
   src
       pyfd26
           __init__.py
           solver.py
   tests
        test_solver.py
```

9.3.2. Installation

```
python -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install -e .[docs]
python -m unittest discover -s tests -v
```

creates a virtual environment, activates it, installs the package in editable mode with the documentation dependencies, and runs the tests.

An alternative approach is to use `conda` (or work globally).

9.3.3. Important Files

The `src/pyfd26` directory contains the code.

`__init__.py`:

```
"""Public package interface for the finite-difference Poisson solver."""

from .solver import construct_matrix, construct_rhs, reshape_solution, solve_poisson

__all__ = [
    "construct_matrix",
    "construct_rhs",
    "reshape_solution",
    "solve_poisson",
]
```

solver.py:

```
# ...
def construct_matrix(n: int):
    # ...
```

pyproject.toml:

```
[build-system]
requires = ["setuptools>=69", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "pyfd26"
version = "0.1.0"
description = "Finite-difference Poisson solver on the unit square."
readme = "README.md"
requires-python = ">=3.11"
authors = [
  { name = "Lambert Theisen", email = "lambert.theisen@rwth-aachen.de" },
]
license = "MIT"
keywords = ["finite differences", "poisson equation", "teaching", "pde"]
dependencies = [
  "numpy>=1.26",
]
classifiers = [
  "Development Status :: 3 - Alpha",
  "Intended Audience :: Education",
  "Programming Language :: Python :: 3",
  "Programming Language :: Python :: 3.11",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: 3.13",
  "Topic :: Scientific/Engineering :: Mathematics",
]

[project.optional-dependencies]
docs = [
  "mkdocs>=1.6",
```

```
  "mkdocstrings[python]>=0.29",
]
examples = [
  "matplotlib>=3.8",
]

[tool.setuptools]
package-dir = {"" = "src"}

[tool.setuptools.packages.find]
where = ["src"]
```

mkdocs.yml:

```
site_name: pyfd26
site_description: Finite-difference Poisson solver documentation
repo_url: https://gitlab.git.nrw/rwth-acom/teaching/pyfd26

theme:
  name: readthedocs

nav:
  - Home: index.md
  - API: api.md

plugins:
  - search
  - mkdocstrings:
      handlers:
        python:
          options:
            docstring_style: google
            show_root_heading: true
            show_source: false

markdown_extensions:
  - admonition
  - toc:
      permalink: true
```

.gitlab-ci.yml:

```
stages:
  - test
  - deploy

default:
  image: python:3.13

variables:
```

```
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

cache:
  paths:
    - .cache/pip

test:
  stage: test
  before_script:
    - python -m pip install --upgrade pip
    - python -m pip install -e . coverage
  script:
    - coverage run -m unittest discover -s tests -v
    - coverage report -m
    - coverage xml
  artifacts:
    when: always
    paths:
      - coverage.xml
  coverage: '/TOTAL\s+\d+\s+\d+\s+(\d+%)/'

pages:
  stage: deploy
  before_script:
    - python -m pip install --upgrade pip
    - python -m pip install -e .[docs]
  script:
    - mkdocs build --strict --site-dir public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH'
```

9.3.4. References

- python.org Packaging

9.4. Questions

- What is the expected convergence rate of the finite difference method for the Poisson equation? How can we verify it?
- How can we modify the code to solve the Poisson equation on a rectangular domain instead of a unit square?
- How can we modify the code to solve the Poisson equation on a non-uniform grid?
- How can we modify the code to solve the Poisson equation in 3D?

10. Project 01 – Open FEM Project with FEniCSx

Exercise Sheet

10.1. Overview

In this mini-project you will design, implement, verify, and analyze a finite element model for a **physical problem or PDE system of your own choice** using the Python/FEniCSx stack. The final submission is a **research-style minipaper** of around **6 pages** (excluding figures) together with a **Git repository** containing code, meshes, tests, project documentation, and all metadata needed to reproduce the results.

> **Important**
>
> **Deadline (end of that date):** 01.06.2026
> **Submission:** Via Moodle as repository link and PDF. Add us as members (`@lambert.theisen` / `@georgii.oblapenko`) to the project repository.

> **Note**
>
> **Own initiative is part of the task.** You should choose a physical setup that interests you, formulate the PDE model, and decide what numerical and physical questions are worth studying. Discuss your idea with us early if you are unsure whether the scope is realistic.

The project should combine:

- a clearly motivated physical problem and PDE model,
- derivation of the weak form including boundary conditions,
- a FEniCSx implementation with boundary conditions handled generically through tags or reusable functions,
- verification by a manufactured solution **or** comparison with another method/reference solution where appropriate,
- a mesh convergence study with error norms and convergence rates,
- a realistic geometry or scenario study comparing meshes, geometries, boundary conditions, or material/physical parameters,
- analysis of at least one physical quantity of interest,
- clear documentation of solver choices, software versions, testing, and reproducibility.

> **Important**
>
> **Software-engineering requirement.** Treat this as a small research software project, not only as a report. Your work should live in a **Git repository** and be reproducible from the command line. Include code, a short project documentation, and lightweight but meaningful automated checks. CI/CD pipelines shall perform the automated testing and hosting of the project documentation, for example via GitLab Pages.

> **Note**
>
> **Scientific focus.** Derivation, verification, comparison, interpretation, and reproducibility matter at least as much as the final visualization.

10.1.1. Learning Outcomes

After completing this project, you should be able to:

- select and justify a PDE model for a physical problem,
- derive and interpret the weak form including boundary terms,
- implement a finite element discretization in FEniCSx,
- verify a PDE solver by mesh refinement, error norms, and convergence rates,
- compare your FEM result with a manufactured solution, another method, or a documented reference where applicable,
- generate or import realistic meshes programmatically,
- analyze how geometry, mesh resolution, boundary conditions, or physical parameters affect meaningful quantities of interest.

10.1.2. Topic Selection

Choose a physical problem/PDE system that is interesting to you and feasible within the project time. Suitable examples include, but are not limited to:

- heat conduction or diffusion,
- nonlinear PDEs such as nonlinear diffusion, nonlinear elasticity, or reaction-diffusion models,
- linear elasticity,
- Stokes or incompressible flow,
- advection-diffusion or reaction-diffusion,
- eigenvalue problems,
- electrostatics or potential flow,
- coupled or parameter-dependent variants of the above, if the scope remains manageable.

Your project may be stationary or time-dependent. If you choose a time-dependent problem, the spatial FEM part must still be central, and the verification/convergence study must remain manageable.

> **Tip**
>
> **Need ideas?** Start from the official DOLFINx Python demos, Dokken's DOLFINx tutorial, the lectures, or your own literature research. Use external examples as learning material, but your submitted setup, implementation choices, discussion, and report must be your own work.

10.1.3. Requirements for the Project and the Minipaper

Other than your lecture notes, consult online resources, papers, textbooks, or software documentation related to your chosen topic. Reference all sources you use. Everyone needs to hand in their own repository containing the minipaper. The minipaper must include a written statement that the work has been done individually. The statement may for example read:

> I hereby declare that I wrote this report on my own and without the use of any other than the cited sources and tools and all explanations that I copied directly or in their sense are marked as such.

Structure the report like a short paper, including, for example:

- introduction and physical setup,
- model and weak formulation,
- implementation and solver choices,
- verification or benchmark comparison,
- mesh convergence study,
- realistic scenario study,
- conclusion.

10.1.3.1. Submission Guidelines

- Submit one PDF report plus the Git repository used to generate the results.
- Submit via Moodle as Link to repo and PDF.
- The repository should have a clean, understandable structure and meaningful file names.
- Use Git throughout the project instead of collecting everything at the end in one folder.
- Label all plots and tables clearly.
- Use consistent notation and include units where appropriate.
- Keep the workflow reproducible and scriptable from start to finish.
- Avoid undocumented manual steps in meshing, postprocessing, or figure generation (or at least document them clearly in the project documentation).
- You may discuss ideas with classmates, but all submitted text, code, plots, and conclusions must be your own.

10.2. Mathematical Model and Weak Form

State your chosen strong-form PDE or PDE system on a domain $\Omega \subset \mathbb{R}^d$, $d = 1, 2$, or 3, and define all unknown fields, material parameters, source terms, and units.

The following scalar second-order equation is only a small illustrative example of the kind of notation one might write down:

$$-\nabla \cdot (A\nabla u) + b \cdot \nabla u + cu = f \quad \text{in } \Omega.$$

Do not treat this as the default project model. Your own project may look quite different: it may be nonlinear, vector-valued, mixed, time-dependent, an eigenvalue problem, or a coupled system.

Describe the boundary decomposition relevant to your model, for example

$$\partial\Omega = \Gamma_D \cup \Gamma_N \cup \Gamma_R \cup \Gamma_{\text{other}},$$

where the subsets are disjoint and may be empty. Explain the physical meaning of every boundary part and every boundary condition.

Typical boundary-condition types include:

$$u = u_D \quad \text{on } \Gamma_D,$$

$$(A\nabla u) \cdot n = g_N \quad \text{on } \Gamma_N,$$

$$(A\nabla u) \cdot n + h(u - u_\infty) = 0 \quad \text{on } \Gamma_R.$$

Adapt signs and notation to your own PDE. For systems such as Stokes or elasticity, formulate the corresponding vector or mixed boundary conditions instead.

In your report, derive the weak form by multiplying with test functions, integrating by parts where appropriate, and explaining:

1. which function spaces contain the trial and test functions,
2. how essential boundary conditions enter the space or are imposed algebraically,
3. which natural boundary conditions appear in the linear form,
4. which Robin, reaction, coupling, or stabilization terms appear in the bilinear or nonlinear residual form,
5. what finite element spaces you use, for example continuous Lagrange elements, Taylor-Hood elements, or another justified choice.

10.3. Required Software Stack

Use the same implementation stack as Lecture 03:

- Python with the `fenicsx` environment/kernel,
- `dolfinx`, `ufl`, `basix`, `mpi4py`, `petsc4py`,
- `dolfinx.fem.petsc.LinearProblem`, `NonlinearProblem`, explicit PETSc assembly, SLEPc, or another appropriate PETSc-backed workflow,
- `gmsh` and `dolfinx.io.gmshio.model_to_mesh` or `dolfinx.io.gmsh.read_from_msh` for generated geometries where applicable,
- `numpy` and `matplotlib` for convergence data and visualization.

ParaView may also be used for visualization if applicable. If you use ParaView, write XDMF/VTK output from your scripts and document the exact plotting workflow well enough that the figures are reproducible.

For reproducibility, your repository must document the environment and include commands that can be run from a clean checkout. A good minimal check is:

```
python -c "import dolfinx, gmsh, petsc4py, mpi4py; print('FEniCSx stack ok')"
```

Document the installation and execution instructions in the `README.md`.

> **Note**
>
> Use Python code and FEniCSx abstractions for the submitted FEM solver. Implementations from Project 00 or other methods may be useful as comparison baselines, but they are not the target stack for the main solver.

10.4. Exercise 1: Problem Choice, Model, and Weak Form

1. Choose a physical problem or PDE system and explain why it is interesting.
2. State the PDE, unknowns, material parameters, source terms, and physical assumptions.
3. Define the boundary decomposition and explain the role of each boundary subset.
4. Derive the weak form from the strong form, including boundary terms.
5. State the finite element spaces and polynomial orders you use.
6. Explain any simplifications, nondimensionalization, or assumptions needed to keep the project feasible.

Your write-up should be specific enough that another student could identify exactly which mathematical problem you solved.

10.5. Exercise 2: FEniCSx Implementation and Verification

Implement your chosen PDE first on a simple domain where verification is possible, such as an interval, unit square, rectangle, unit cube, or another geometry with a known solution.

1. Implement a reusable FEniCSx solver for your chosen PDE.
2. Use the Lecture 03 pattern: build function spaces, define UFL forms/residuals, impose boundary conditions, and solve with PETSc-backed linear algebra.
3. Accept geometry, material data, source terms, and boundary data in a way that lets you change test cases without rewriting the full solver.
4. Represent boundary subsets by facet tags whenever tags are available.
5. Verify the implementation by at least one of the following:
   - **manufactured solution:** choose an exact solution and derive matching source terms and boundary data,
   - **comparison with another method:** for example finite differences on a structured-grid problem if applicable,
   - **comparison with a trusted reference:** for example an analytical solution, benchmark value, or carefully documented literature result.
   - [**mesh convergence study:** show that the solver converges under mesh refinement]
6. Explain why your chosen verification method is appropriate for your PDE.
7. Include plots or tables that compare numerical and reference results.

10.6. Exercise 3: Mesh Convergence Study

Perform a mesh convergence study for the verification problem.

1. Use at least four mesh resolutions.
2. Define the mesh-size measure h you use.

3. Compute at least one error norm. Use more than one where meaningful, for example L^2, H^1 seminorm, energy norm, maximum norm, divergence error, eigenvalue error, or an application-specific quantity.
4. Plot error versus h on a log-log scale.
5. Estimate experimental convergence rates.
6. Compare the observed rates with theory or with a justified expectation for your element choice and PDE.
7. If exact error norms are not available, compare against a sufficiently refined reference solution and state the limitations.
8. If no theoretical rates are known for your problem, study the changes between consecutive mesh refinements. For example, compare quantities of interest or solution differences from mesh to mesh and explain whether the sequence appears to stabilize.

For many smooth scalar elliptic problems with P1 elements, one often expects approximately

$$\|u - u_h\|_{L^2} = \mathcal{O}(h^2), \qquad \|u - u_h\|_{H^1} = \mathcal{O}(h).$$

These rates are not universal. State the expected behavior for your own PDE, element, regularity, and boundary conditions.

10.7. Exercise 4: Realistic Scenario Study

Create a more realistic physical scenario for your chosen problem and analyze it with your FEM solver.

10.7.1. Reproducibility Requirement

Your geometry, mesh, boundary tags, solver settings, and postprocessing must be reproducible. Do not rely on manual GUI-only meshing steps. Use one of the following:

- the Gmsh Python API with `gmsh.model.occ`,
- a scripted import of an `.geo` geometry,
- a documented mesh-generation script from another tool,
- a simple programmatically generated FEniCSx mesh if that is sufficient for your physical setup.

Import external meshes into FEniCSx in a reproducible way, for example via `dolfinx.io.gmshio.model_to_mesh` or `dolfinx.io.gmsh.read_from_msh`. Represent boundary conditions through facet tags or another documented tagging mechanism so the same solver can be reused across variants.

10.7.2. Required Study

1. Compare at least **two variants** under controlled conditions. Variants may be different meshes, geometries, boundary-condition choices, material parameters, source terms, or physical configurations.
2. Define at least **one quantitative physical quantity of interest** before running the comparison.
3. Suitable quantities include maximum/minimum field value, average value in a region, total flux, lift/drag-like integral, displacement, stress concentration, pressure drop, eigenfrequency, decay rate, reaction yield, or another quantity appropriate to your PDE.

4. Visualize the mesh, boundary tags where useful, and the computed field(s). Matplotlib, PyVista, and ParaView are all acceptable if the workflow is documented.
5. Interpret the results physically: where are the bottlenecks, hotspots, boundary layers, stress concentrations, dominant paths, unstable regions, or other relevant structures?
6. Identify which variant performs better with respect to your metric and explain why.

10.8. Backup Option: Stationary Heat Conduction

The intended project is an own-initiative FEM study. If you really do not know what to do, you may use a stationary heat-conduction project only as a **backup option after talking to us**.

A possible backup setup is:

- solve a stationary heat or diffusion equation,
- derive the weak form with Dirichlet, Neumann, and Robin boundary conditions,
- verify by manufactured solution or finite-difference comparison,
- perform a mesh convergence study with L^2 and H^1-type errors,
- create a realistic thermal scenario such as a thermal bridge, heat sink, pipe insulation defect, or bracket-like geometry,
- compare at least two geometries or boundary-condition variants using a quantity such as total heat flux, maximum temperature, or effective thermal resistance.

This fallback is deliberately less open-ended. Use it only if you have discussed the topic choice with us and cannot identify a suitable own project.

10.9. Optional Extensions

If the mandatory scope is complete, you may extend the project with one of the following:

- time-dependent simulation,
- nonlinear material laws or nonlinear PDEs,
- spatially varying coefficients,
- multi-material subdomains,
- mixed finite element methods,
- eigenvalue problems,
- coupling between two physical fields,
- solver-performance comparison for larger meshes,
- a 3D geometry instead of a 2D cross-section,
- comparison of different numerical solvers and/or preconditioners for the linear system.

10.10. Suggested Repository Layout

This is only a suggestion. The final structure will look different depending on your PDE, geometry, solver workflow, and plotting pipeline. Choose a layout that makes your own project reproducible and understandable.

```
project01/
  pyproject.toml or environment.yml
  README.md
  src/
    solver.py
    geometry.py
    ... more Python package files
  tests/
    test_verification.py
  scripts/
    run_convergence.py
    run_scenario_study.py
  results/
    figures/
  report/
    report.qmd
```

Keep generated results separate from solver source code. Commit the scripts and small configuration files needed to regenerate figures; avoid committing large generated mesh or output files unless you have a clear reason.

11. TODO (for next year)

- Mention that for nonlinear Problems, residual checking and convergence monitoring is important and shall be included.

Part III.

Skills

12. Shell for Simulation Workflows

Practical shell fundamentals for PDE/CFD/DSMC projects

12.1. Basic Shell for Simulation Software

For most of simulation software (except GUIs), the shell terminal is the primary interface to the tools. You either find executables (`a.out`, `programm.exe`, ..) or scripts (`bash run.sh`, `python3 launch.py`, `julia solver.jl`, ..) that you run from the terminal.

> **Note**
>
> For reproducability, we try to avoid GUIs and prefer command-line interfaces (CLIs) that can be scripted and version-controlled (see Git).

> **Important**
>
> Especially when running our simulation program on a cluster (RWTH CLAIX, see Figure 12.1) or server, we will not have access to a GUI and must rely on the shell.

Figure 12.1.

12.1.1. Overview

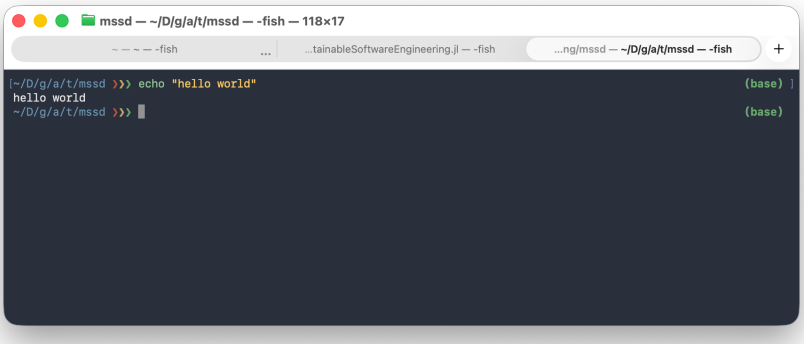
The Shell is a command-line interface (CLI) used in Linux/Unix operating systems. It allows users to interact with the system by typing commands, rather than using a graphical interface. It is text-based and provides powerful tools for file management, process control, and automation. Common shells include Bash, zsh, and fish.

You can excesss the shell through a terminal application on your operating system:

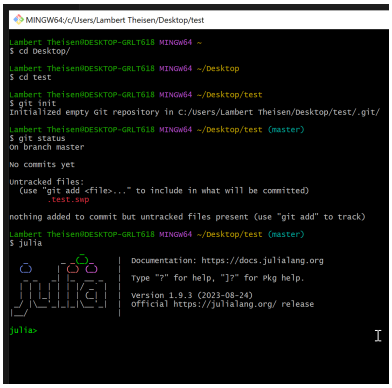
1. **Linux**: Terminal app
2. **macOS**: Terminal app
3. **Windows**: Windows Subsystem for Linux (WSL), Git Bash, PowerShell, …

12.1.2. Basic commands

- `pwd`: Print working directory. Shows the current directory you are in.



(a) macOS terminal



(a) Git Bash on Windows

```
pwd
```

```
/builds/rwth-acom/teaching/mssd/skills
```

- `ls`: List files and directories. Shows the contents of the current directory.

```
ls  # alias/function forces --color=never
```

```
01-shell.qmd
01-shell.quarto_ipynb
02-git.qmd
02-git.quarto_ipynb
03-gmsh.ipynb
04-paraview.ipynb
99-todo.qmd
```

```
ls -l
```

```
total 1952
-rw-rw-rw-. 1 root root   11892 May  4 12:50 01-shell.qmd
-rw-r--r--. 1 root root   22146 May  7 09:00 01-shell.quarto_ipynb
-rw-rw-rw-. 1 root root   18051 May  4 12:50 02-git.qmd
-rw-r--r--. 1 root root   26098 May  7 09:00 02-git.quarto_ipynb
-rw-rw-rw-. 1 root root 1581904 May  5 14:12 03-gmsh.ipynb
-rw-rw-rw-. 1 root root  319617 May  5 14:12 04-paraview.ipynb
-rw-rw-rw-. 1 root root     217 May  4 12:50 99-todo.qmd
```

```
ls -a
```

```
.
..
.gitignore
01-shell.qmd
```

```
01-shell.quarto_ipynb
02-git.qmd
02-git.quarto_ipynb
03-gmsh.ipynb
04-paraview.ipynb
99-todo.qmd
```

- `cd`: Change directory. Moves you to a different directory.

```
pwd; cd ..; pwd; cd skills; pwd
```

```
/builds/rwth-acom/teaching/mssd/skills
/builds/rwth-acom/teaching/mssd
/builds/rwth-acom/teaching/mssd/skills
```

- `echo`: Print text to the terminal. Useful for displaying messages or environment variable values in scripts.

```
echo "Hello, Shell!"
```

```
Hello, Shell!
```

```
NAME="Lambert"; echo "hello, my name is $NAME."
```

```
hello, my name is Lambert.
```

- `touch`: Create an empty file..

```
touch test_file.txt
```

```
ls -l test_file.txt
```

```
-rw-r--r--. 1 root root 0 May  7 09:00 test_file.txt
```

- `mkdir`: Make directory. Creates a new directory.

```
mkdir -p test_dir/subdir
```

```
ls -l test_dir
```

```
total 0
drwxr-xr-x. 2 root root 6 May  7 09:00 subdir
```

- `rmdir`: Remove directory. Deletes a directory and its contents.

```
mkdir new_dir
```

```
rmdir new_dir
```

- `cp`: Copy files and directories.

```
touch source_file.txt
```

```
cp source_file.txt copied_file.txt
```

```
ls -l source_file.txt copied_file.txt
```

```
-rw-r--r--. 1 root root 0 May  7 09:00 copied_file.txt
-rw-r--r--. 1 root root 0 May  7 09:00 source_file.txt
```

- `mv`: Move or rename files and directories.

```
mv copied_file.txt renamed_file.txt
```

```
ls -l renamed_file.txt
```

```
-rw-r--r--. 1 root root 0 May  7 09:00 renamed_file.txt
```

- `rm`: Remove files and directories. Use flag `-r` for recursive deletion.

```
ls -l source_file.txt renamed_file.txt
```

```
-rw-r--r--. 1 root root 0 May  7 09:00 renamed_file.txt
-rw-r--r--. 1 root root 0 May  7 09:00 source_file.txt
```

```
rm source_file.txt renamed_file.txt
```

```
ls -l source_file.txt renamed_file.txt
```

```
ls: cannot access 'source_file.txt': No such file or directory
ls: cannot access 'renamed_file.txt': No such file or directory
```

```
rm -r test_dir
```

```
ls -l test_dir
```

```
ls: cannot access 'test_dir': No such file or directory
```

- `cat`: Concatenate and display file contents. Useful for quickly viewing the contents of a file.

```
echo 'This is a test file.' > test_file.txt  # note the > operator to write to the file
```

```
cat test_file.txt
```

```
This is a test file.
```

- `grep`: Search for patterns in files. Useful for finding specific information in log files or code.

```
echo -e 'line1: error\nline2: warning\nline3: info' > log.txt
```

```
grep 'error' log.txt
```

```
line1: error
```

```
grep -n 'warning' log.txt
```

```
2:line2: warning
```

- `find`: Search for files and directories. Useful in combination with wildcards (see GNU Bash: Pattern Matching).

```
find . -name '*.txt'
```

```
./test_file.txt
./log.txt
```

- `man`: Manual. Displays the manual page for a command, providing detailed information on its usage and options.

```
man echo | col -bx | head -n 10  # col -bx to strip formatting
```

```
bash: line 2: col: command not found
```

- `|`, `>`, `>>`, `&&`, `||`, `;`, `&`: Shell operators for piping, redirection, and command chaining.

```
echo 'Hello, World!' | tr '[:lower:]' '[:upper:]'  # pipe output to another command (tr for up
```

```
HELLO, WORLD!
```

```
echo 'This is a test.' > test_output.txt  # redirect output to a file (overwrite)
```

```
echo 'This is another line.' >> test_output.txt  # redirect output to a file (append)
```

```
echo 'First command' && echo 'Second command'  # execute second command only if first succeeds
```

```
First command
Second command
```

```
false || echo 'First command failed, executing second command'  # execute second command only
```

```
First command failed, executing second command
```

```
echo 'Command 1'; echo 'Command 2'  # execute commands sequentially regardless of success
```

```
Command 1
Command 2
```

```
sleep 0.1 & echo 'This runs in the background while sleep is running'  # run command in the ba
```

```
This runs in the background while sleep is running
```

12.1.3. System Information and Utils

- **df**: Report file system disk space usage.

```
df -h | head -n 5
```

```
Filesystem      Size  Used Avail Use% Mounted on
overlay         297G  116G  181G  39% /
tmpfs            64M     0   64M   0% /dev
shm             512M     0  512M   0% /dev/shm
/dev/sdb1       200G   64G  137G  32% /cache
```

- **top**: Display Linux tasks.

```
top -n 1 | head -n 20
```

```
top: failed tty get
```

- **du**: Estimate file space usage.

```
du -sh *
```

```
12K 01-shell.qmd
24K 01-shell.quarto_ipynb
20K 02-git.qmd
28K 02-git.quarto_ipynb
1.6M    03-gmsh.ipynb
316K    04-paraview.ipynb
4.0K    99-todo.qmd
4.0K    log.txt
4.0K    test_file.txt
4.0K    test_output.txt
```

- **htop**: Interactive process viewer (if installed).
- **kill**: Terminate processes by PID.

```
sleep 10 & echo $! > sleep.pid
```

```
cat sleep.pid
```

```
826
```

```
kill $(cat sleep.pid)  # terminate the background process
```

```
bash: line 2: kill: (826) - No such process
```

12.1.4. Tips

- Use the **Tab** key for auto-completing commands and file names.
- Use **up and down arrow keys** to navigate through command history.
- Use **Ctrl + C** to terminate a running command.
- Use **Ctrl + L** to clear the terminal screen.
- Use **Ctrl + R** to search through command history.

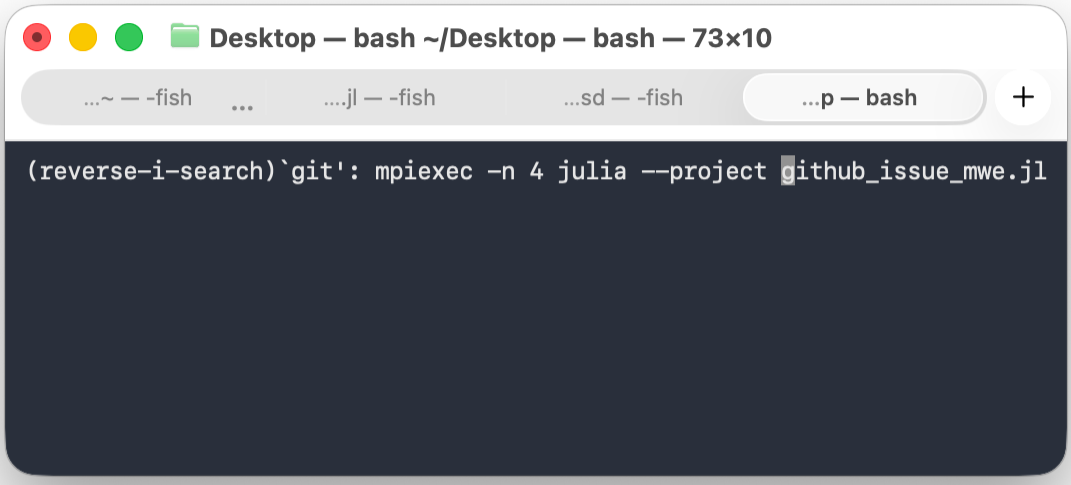


Figure 12.4.: Shell Search

12.2. Scripting

Scrripting and automation are key for reproducibility and efficiency in simulation workflows. You can write shell scripts (`.sh` files) to automate sequences of commands, set up environments, and run simulations with specific parameters.

An example script:

```
#!/bin/bash
# This is a simple shell script to run a simulation and post-process results
echo "Running simulation..."
echo "Filename: $1"

# Write some data to filename
echo "x,y,z" > $1
echo "1,2,3" >> $1
echo "Simulation complete. Output written to $1"
```

12.2.1. An example solver

```
echo "#!/bin/bash
echo 'Running solver...'
echo 'x,y,z' > output.csv
echo '1,2,3' >> output.csv
echo 'Write output...'" > solver.sh
```

```
ls -l solver.sh
chmod +x solver.sh  # make script executable
ls -l solver.sh
```

```
-rw-r--r--. 1 root root 113 May  7 09:01 solver.sh
-rwxr-xr-x. 1 root root 113 May  7 09:01 solver.sh
```

```
./solver.sh  # run the script
sh solver.sh  # alternative way to run the script
```

```
Running solver...
Write output...
Running solver...
Write output...
```

```
cat output.csv  # check the output
```

```
x,y,z
1,2,3
```

12.2.2. An example post-processing script

```
echo "#!/bin/bash
echo 'Calculate length of vector...'
awk -F, 'NR>1 {print sqrt(\$1^2 + \$2^2 + \$3^2)}' output.csv > length.txt
echo 'Write length to file...'" > postprocess.sh
cat postprocess.sh
```

```
#!/bin/bash
echo 'Calculate length of vector...'
awk -F, 'NR>1 {print sqrt($1^2 + $2^2 + $3^2)}' output.csv > length.txt
echo 'Write length to file...'
```

```
sh postprocess.sh  # run the post-processing script
cat length.txt  # check the output
```

```
Calculate length of vector...
Write length to file...
3.74166
```

12.3. Integrated Development Environments (IDEs)

IDEs (e.g., VS Code, Vim, Emacs, Jetbrains, Atom) combine a text editor with additional features like terminal integration, version control, debugging tools, and extensions for specific languages and frameworks.

- **VS Code**: All-in-one solution, rich extensions, user-friendly
- **Vim/Emacs**: Lightweight, highly customizable, steep learning curve

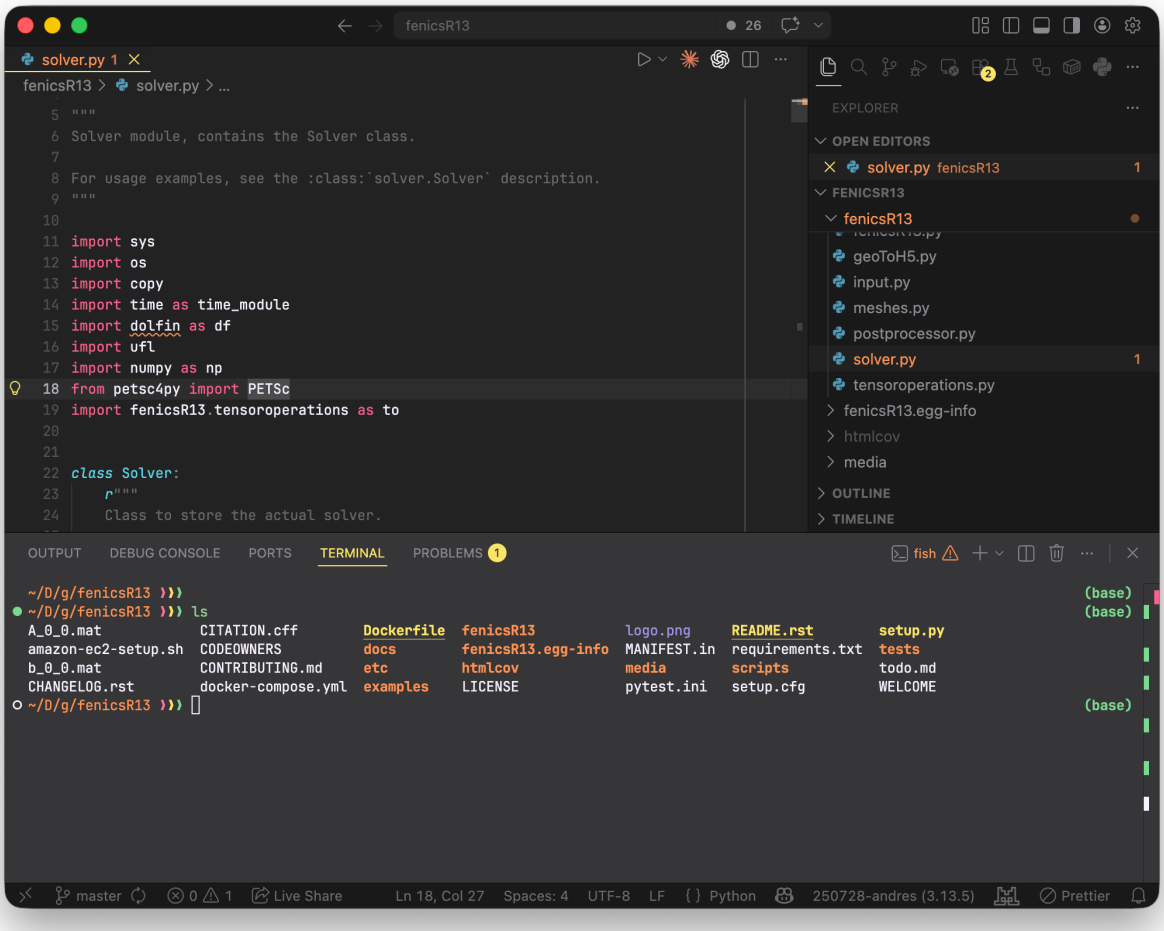


Figure 12.5.: fenicsR13 Solver in VS Code

12.4. References and Further Reading

- MIT: The Missing Semester of Your CS Education
- Bash Guide
- Shell Scripting Tutorial
- Random collection of useful commands by Dr. Georgii Oblapenko

12.5. Exercises

1. Write a shell script that takes a filename as an argument, checks if the file exists, and prints its contents if it does, or an error message if it doesn't.
2. Use `curl` or `wget` to download our course website and use `grep` to extract all dates mentioned on the page.

12.6. Questions

1. What are the advantages of using the shell for running simulations compared to GUIs?
2. How can you automate a simulation workflow using shell scripts?
3. What are some common shell operators and how do they work?
4. How can you manage and navigate the file system using shell commands?

More good exercises can be found in the MIT Missing Semester course.

13. Skills 02 – Git for Simulation Software Development

Version-control habits for numerical projects

13.1. Version Control with Git

Problem: We change files frequently want to store different versions, track who changed what and when, and collaborate with others without overwriting each other's work.

```
solver.py
solver_v1.py
solver_lamberts_changes.py
solver_final.py
solver_FINAL
solver_FINALFINALLLL.py
```

Version control systems like Git track changes to files over time, enabling:

- easier collaboration,
- history tracking,
- branching for experiments,
- safe rollbacks.

> **Note**
>
> Version control is essential for managing the complexity (of simulation software development), where code, case setups, and post-processing scripts evolve together.

13.1.1. Key Features of Git

- **Distributed Version Control**: Each developer has a full copy of the project, including its history, which means work can be done offline and later synced with the central repository.
- **Branching and Merging**: Git allows the creation of branches to experiment with new features. These branches can be merged back into the main branch when the feature is ready.
- **Fast and Lightweight**: Git is designed to be fast and efficient, even for large projects.
- **Data Integrity**: Every file and commit is checksummed, ensuring the integrity of your code.

13.1.2. Basic Git Commands

- `git init`: Initializes a new Git repository.
- `git clone [url]`: Clones a repository from a remote server.
- `git status`: Shows the status of changes in the working directory.

```
git status
```

```
HEAD detached at 930ad28
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ../quarto-1.8.25-linux-amd64.deb

nothing added to commit but untracked files present (use "git add" to track)
```

- `git diff`: Shows the differences between the working directory and the last commit.
- `git add [file]`: Stages a file for commit.
- `git commit -m "[commit message]"`: Commits the staged changes with a message.
- `git push`: Pushes commits to a remote repository.
- `git pull`: Fetches and merges changes from a remote repository.
- `git fetch`: Fetches changes from a remote repository without merging.
- `git branch`: Lists branches in the repository.
- `git checkout [branch]`: Switches to a different branch.
- `git merge [branch]`: Merges a branch into the current branch.
- `git rebase [branch]`: Reapplies commits on top of another base tip.
- … many more

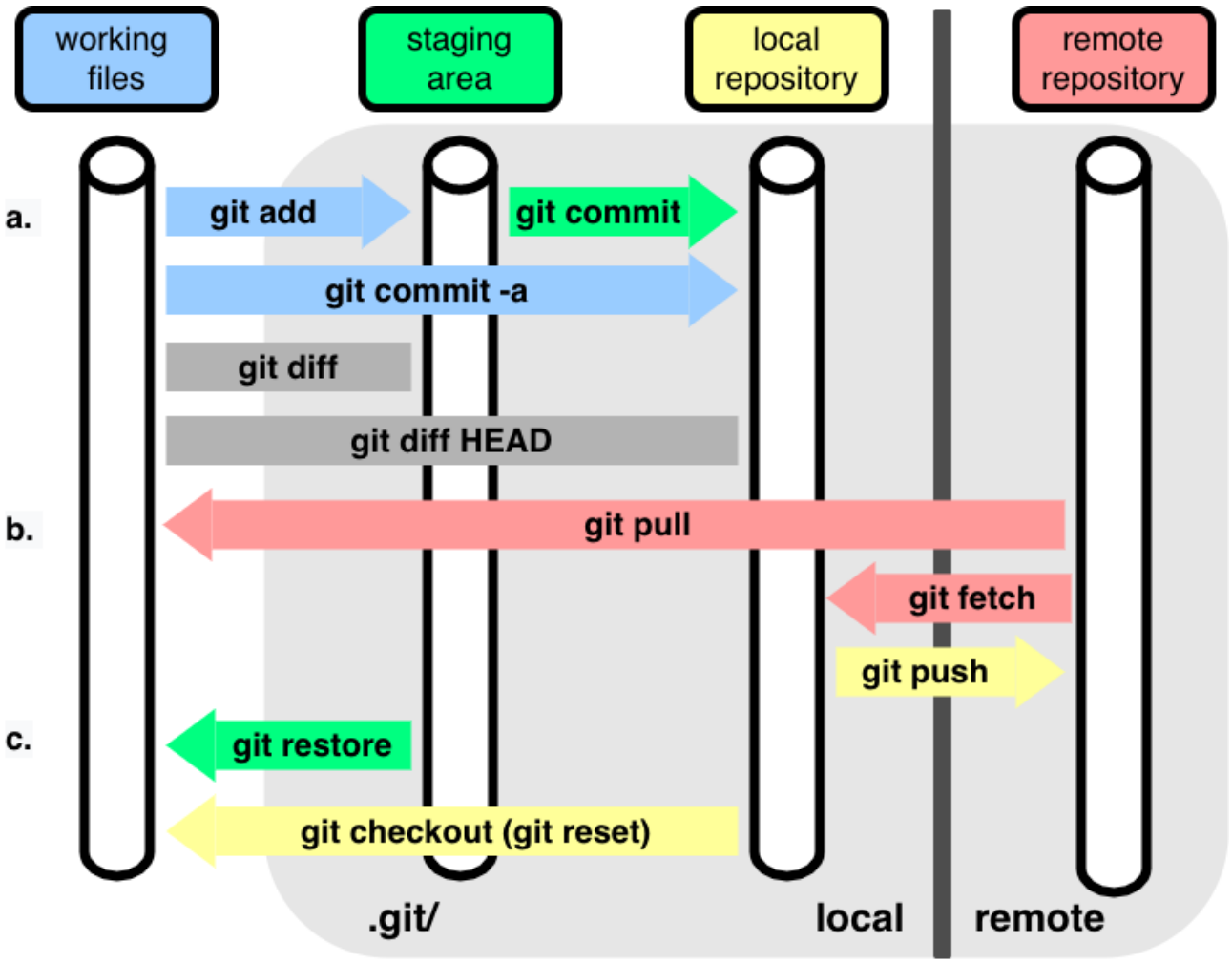


Figure 13.1.: Git Workflow, Source: https://github.com/merely-useful/py-rse

13.1.3. IDE Integration

Most modern IDEs have built-in support for Git, allowing you to perform version control operations without leaving the editor.

> **!** Important
>
> While IDE integration can be convenient, it may not expose all Git features or may abstract away important details. It's absolutely necessary to understand the underlying Git commands and concepts to use version control effectively, especially when resolving conflicts or performing complex operations!

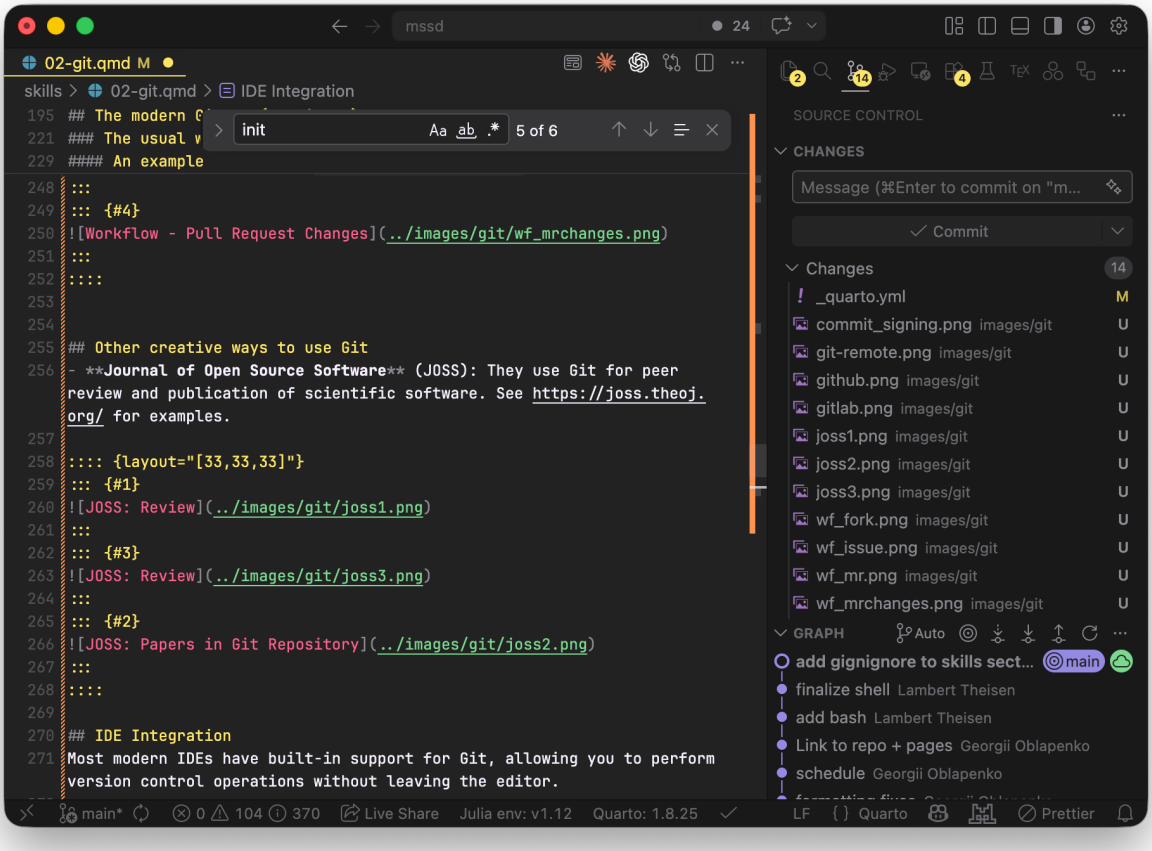


Figure 13.2.: Git in VS Code

13.1.4. Basic Git Workflow

0. **Create**: Create Repo on the Gitlab instance in the Webbrowser (or in the terminal with `git init`).
1. **Clone**: Clone the remote repository to your local machine:

```
git clone git@gitlab.git.nrw:rwth-acom/teaching/mssd.git
```

2. **Edit**: Navigate to the cloned repository directory and create a new file:

```
cd mssd
touch README.md
```

3. **Staging**: Add the new file to the staging area, preparing it for a commit:

```
git add README.md
```

4. **Commit**: Commit the staged file with a descriptive message:

```
git commit -m "Add README.md  "
```

5. **Push**: Push the commit to the remote repository:

```
git push origin main
```

> **Note**
>
> Every repository should have a `README.md` file that provides an overview of the project, instructions for setup and usage, and any other relevant information. The file is usually rendered in the Webbrowser and you can use Markdown syntax to format it nicely.
>
> ```
> # Title
> ## Subtitle
> - Bullet point 1
> - Bullet point 2
> ....
> ```

13.1.5. Commit messages

Write descriptive commit messages: what changed, why it changed, and the expected numerical impact.

```
Use stricter linear solver tolerance in Stokes solve

- switch rtol from 1e-6 to 1e-8
- improve pressure smoothness near outlet
- runtime +12% on reference mesh
```

13.1.6. What usually to track in Git for simulation software development?

- Source code (`.py`, `.jl`, solver scripts)
- Case dictionaries (OpenFOAM) / config files,
- Mesh generation scripts
- Post-processing scripts
- Documentation and run notes
- Environment setup scripts (e.g. `requirements.txt`, `environment.yml`).

Avoid (or think twice before) tracking:

- large (binary) outputs,
- generated visualization files,
- local caches/build artifacts. Put these in `.gitignore` to keep the repository clean and efficient.

`.gitignore-example`:

```
# Ignore all files in the build directory
docs/build/*
# Ignore all .data files
*.data
# Ignore all .log files
*.log
# ...
```

> **Tip**
>
> Use the Git Large File Storage (LFS) extension for large files that need to be versioned, but be mindful of storage limits and costs. See https://git-lfs.github.com/ for more details.

```
# content of .gitignore (dot means hidden file)
docs/build/*  # (star/asterix is placeholder/wildcard to ignore everything in that folder)
```

13.2. Getting Started

13.2.1. Identity Setup

To start using Git, first install it on your machine. Then, set up your user name and email with the following commands:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

> **Note**
>
> In principle, anyone could use your name and email to make commits, see this story for more details. To prevent this, you can sign your commits with GPG keys, which adds an extra layer of security and authenticity to your commits.

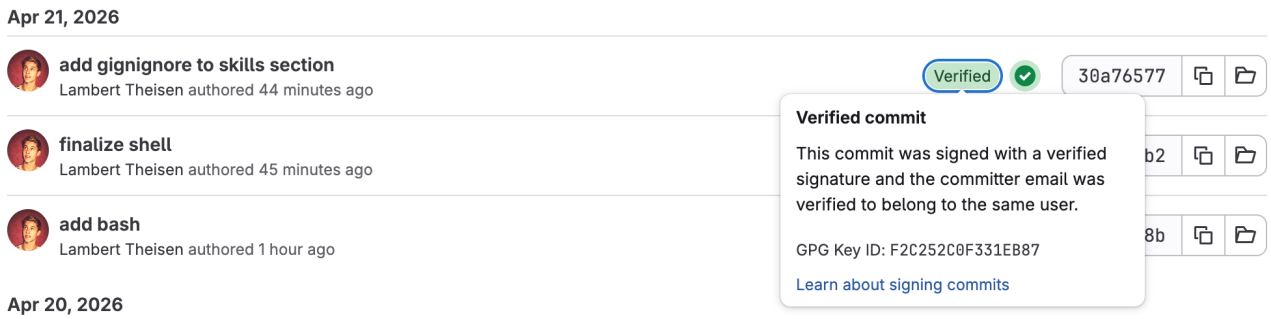


Figure 13.3.: Git commit signing

13.2.2. Authentification on Gitlab/Github platforms

For pushing to remote repositories, you need to authenticate. The easiest way is to use SSH keys:

1. Generate an SSH key pair if you don't have one:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/root/.ssh/id_ed25519):
```

2. Add the public key to your Gitlab/Github account (copy the content of `~/.ssh/id_ed25519.pub` and add it in the SSH keys section of your account settings).

> **Warning**
>
> Please secure your keys with a strong passphrase because access to your private key means that someone could steal your identity and access your repositories!

13.2.3. Branching and resolving conflicts

When working on a new feature or experiment, create a new branch to keep your changes isolated from the main branch. This allows you to work on multiple features simultaneously without affecting the stable codebase or other devs.

Example:

```
rm -rf test_repo
mkdir test_repo
cd test_repo
git init -b main
git config user.name "MSSD Example"
git config user.email "mssd-example@example.com"
git config commit.gpgsign false
echo "Hello, World!" > file.txt
git add file.txt
git commit -m "Initial commit with file.txt"
git log
```

```
Initialized empty Git repository in /builds/rwth-acom/teaching/mssd/skills/test_repo/.git/
[main (root-commit) a0a9be4] Initial commit with file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
commit a0a9be4985c3cf0eadcc9bdce3dbcf1001e3557c
Author: MSSD Example <mssd-example@example.com>
Date:   Thu May 7 09:01:11 2026 +0000

    Initial commit with file.txt
```

```
cd test_repo
git checkout -b feature-branch
echo "This is a new feature." >> file.txt
git add file.txt
git commit -m "Add new feature to file.txt"
```

```
[feature-branch 57cb983] Add new feature to file.txt
 1 file changed, 1 insertion(+)
```

```
Switched to a new branch 'feature-branch'
```

Somebody else changes the `main` branch meanwhile:

```
cd test_repo
git checkout main
echo "This is a change in main." >> file.txt
git add file.txt
git commit -m "Change in main branch"
```

```
[main 37caf0e] Change in main branch
 1 file changed, 1 insertion(+)
```

```
Switched to branch 'main'
```

Now, if you try to merge `feature-branch` into `main`, you will encounter a conflict because both branches have modified the same line in `file.txt`:

```
cd test_repo
git checkout main
git merge feature-branch
```

```
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
Already on 'main'
```

Inspect the merge conflict in `file.txt` and resolve **:

```
cd test_repo
cat file.txt  # show the conflict markers
# Edit file.txt to resolve the conflict, then stage and commit the resolution
echo "This is a change in main." > file.txt
echo "This is a new feature." >> file.txt
git add file.txt
git commit -m "Resolve merge conflict between main and feature-branch"
```

```
Hello, World!
<<<<<<< HEAD
This is a change in main.
=======
This is a new feature.
>>>>>>> feature-branch
[main 506e41f] Resolve merge conflict between main and feature-branch
```

13.2.4. Merging and rebasing

When your feature branch is ready, you can merge it back into the main branch. If there have been changes in the main branch since you created your feature branch, you can either:

1. **Merge** the main branch into your feature branch. After resolving conflicts, you would create a merge commit combining both histories.
    - **Pro**: complete history with feature evolution / context
    - **Con**: more cluttered commit history (merge commits), especially for complex features with many commits.
2. **Rebase** your feature branch onto the latest main branch. This rewrites the history of your feature branch to appear as if you had created it from the latest main branch, which can lead to a cleaner commit history.
    - **Pro**: results in a cleaner, linear commit history without merge commits, making it easier to understand the sequence of changes.
    - **Con**: can be more complex to manage, especially if there are conflicts during the rebase process. It also rewrites history, which can cause issues if the feature branch has already been shared (`pushed`) with others.

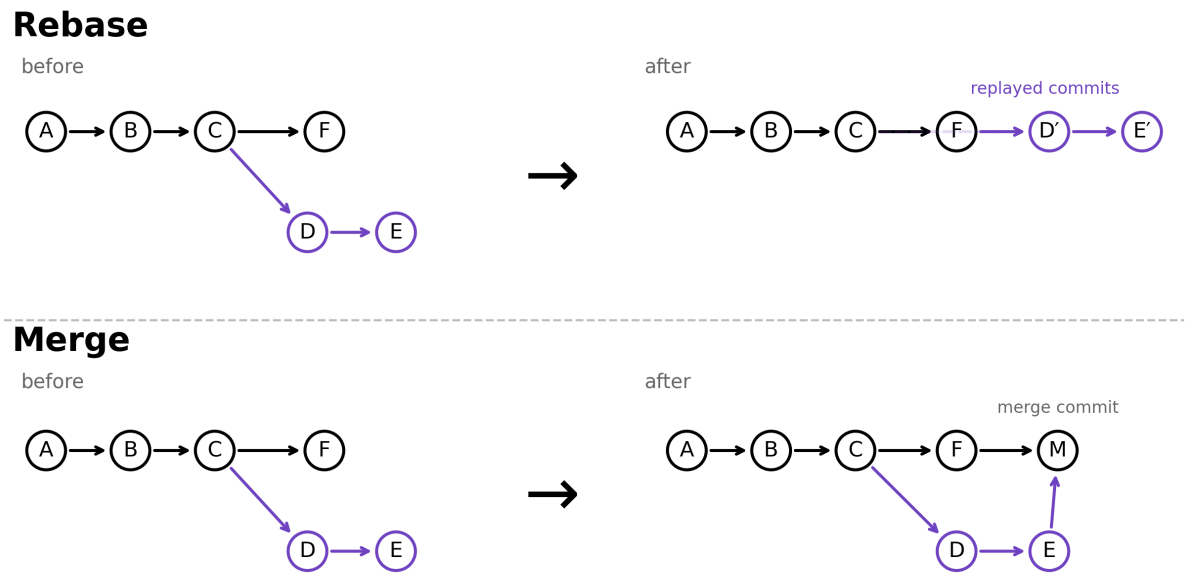


Figure 13.4.: Merge vs. Rebase

> **Warning**
>
> Different projects and teams have different preferences for merging vs. rebasing, and there is no one-size-fits-all answer. Search for a `CONTRIBUTING.md` file in the repository or ask the maintainers for their preferred workflow.

> **Note**
>
> One possible strategy: If the feature branch is already shared with others, prefer merging to avoid rewriting history. If the feature branch is still local and not shared, use rebase.
> Or: Rebase first and then merge to preserve a clean history while still showing the feature branch as a separate line of development.

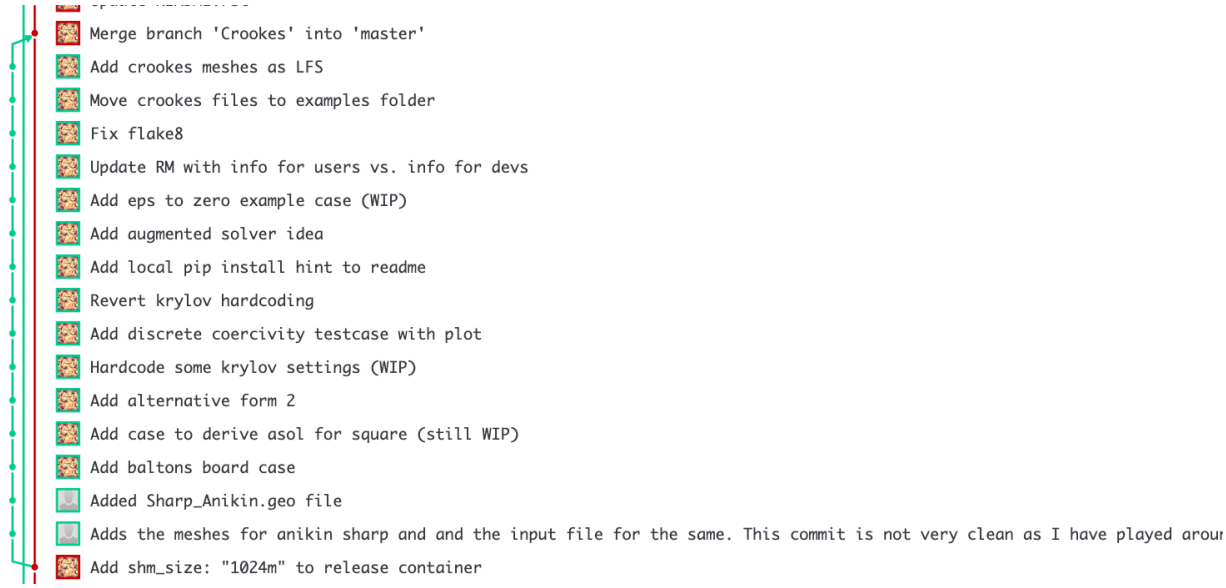


Figure 13.5.: Rebase and merge

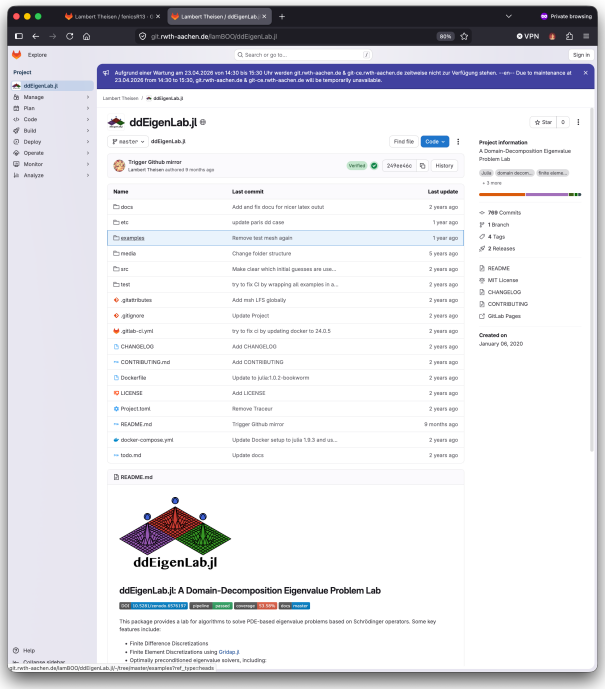
13.3. The Gitlab (or Github) ecosystem

Besides the basic `git` software, there was a rich ecosystem of platforms and tools that enhance collaboration and project management created.

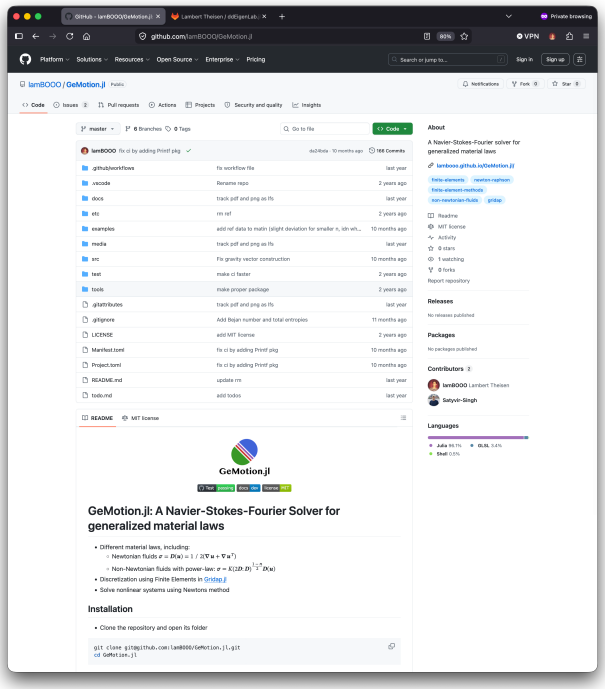
The two major platforms are Gitlab and Github, which offer:

- web-based interfaces for repository management,
- issue tracking,
- pull/merge requests,
- CI/CD pipelines,
- project wikis,
- and more.

> **Note**
>
> The RWTH provides a Gitlab instance for students and staff (see RWTH Gitlab for teaching or RWTH Gitlab CE for non-teaching projects [license differs]), which is ideal for teaching and research projects. Git NRW will (probably) be the new platform for RWTH projects in the future, so we use it for this course to get familiar with it.



(a) Gitlab



(a) Github

13.3.1. The usual workflow in OS projects

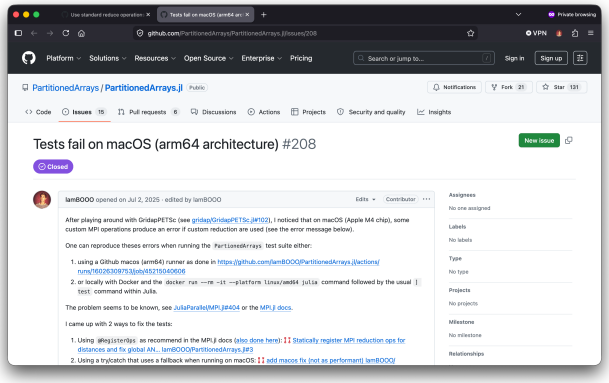
It would not make sense for the maintainers of a OS project to give write access to everyone, so they usually have a "core team" with write access and a "contributor" role for everyone else. Contributors can fork the repository, make changes in their own copy, and then submit a pull/merge request to the main repository. The core team reviews the changes and decides whether to merge them.

> **Note**
>
> A *fork* is a personal copy of someone else's repository. It allows you to freely experiment with changes without affecting the original project. When you're ready to share your changes, you can submit a pull/merge request to the original repository.

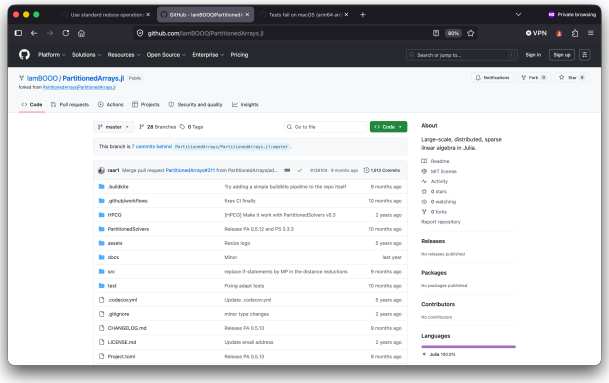
13.3.1.1. An example

See, e.g., https://github.com/PartitionedArrays/PartitionedArrays.jl/pull/209.

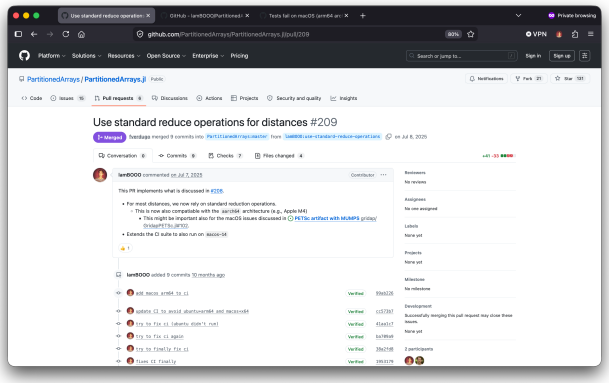
1. Open or find an issue to work on, discuss with the maintainers if necessary.
2. Fork the repository and clone it to your local machine.
3. Add your changes (in a new branch), commit them, and push (the branch) to your fork.
4. Create a pull/merge request from your fork to the original repository, describing your changes and their motivation.
5. Include potential feedback from the maintainers and iterate until the pull/merge request is accepted and merged.



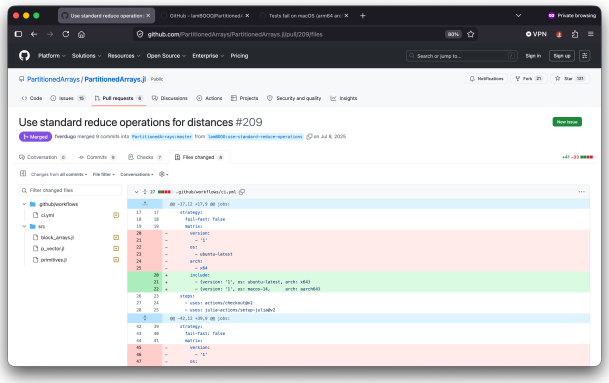
(a) Workflow - Issue



(a) Workflow - Fork



(a) Workflow - Pull Request



(a) Workflow - Pull Request Changes

13.3.2. Gitlab (Github) CI/CD:

Git NRW offers CI/CD (Continuous Integration/Continuous Deployment) features with free *shared runners*. Including:

1. Executing scripts/tests automatically on every push or pull/merge request. Useful for:
    1. Testing code changes (e.g., unit tests, integration tests).
    2. Building documentation.
    3. Running simulations to check for regressions.
2. Saving artifacts (e.g., test results, generated documentation) for later inspection. Useful for:
    1. Sharing results of CI runs with collaborators.
    2. Hosting a static webpage (for, e.g., documentation, personal blog, etc.) using Gitlab Pages or Github Pages.
3. Container registries for reproducable Docker image storage.
4. Tags and Releases: Create software releases with version tags, release notes, and downloadable assets.

Dockerfile example

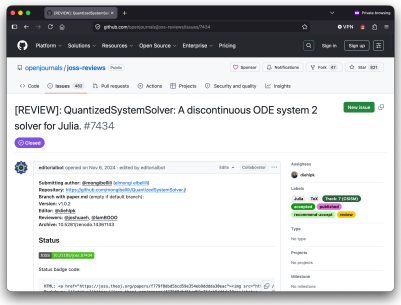
```
# Dockerfile
FROM python:3.13
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 8080

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Note

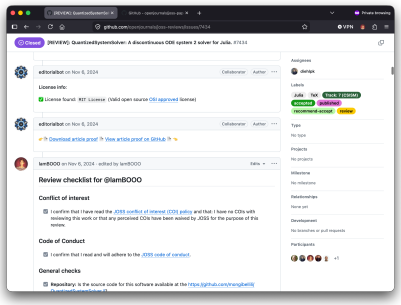
CI/CD features are best understood with an example, see finite difference code later on.

13.3.3. Other creative ways to use Git

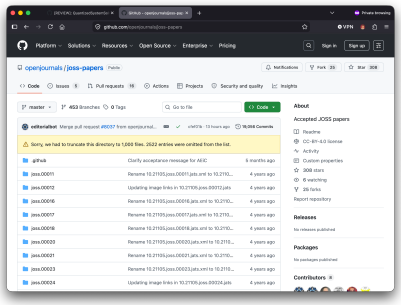
- **Journal of Open Source Software** (JOSS): They use Git for peer review and publication of scientific software. See https://joss.theoj.org/ for examples.



(a) JOSS: Review



(a) JOSS: Review



(a) JOSS: Papers in Git Repository

13.4. Exercises

1. Create a new Git repository for a simple simulation project (e.g., a 2D heat diffusion solver). Initialize the repository, create a README file, and make your first commit.
    - Create a new branch for an experiment (e.g., testing a new solver algorithm).
    - Make some changes in the new branch, commit them, and then switch back to the main branch. Use `git diff` to see the differences between the branches.
    - Create a `.gitignore` file to exclude generated output files from being tracked in Git.
    - If you have access to a Gitlab or Github repository, try pushing your local repository to the remote and creating a pull/merge request.
2. Open an issue in a public open-source repository, this could be an idea for a new feature or a bug report or simply a typo in the documentation. Then, fork the repository, clone it to your local machine, and make a change to address the issue you opened. Finally, submit a pull/merge request with a clear description of your changes and how they address the issue.
3. What is the difference between `git merge` and `git rebase`, and when would you use each one in the context of a simulation software project?

13.5. Questions

1. What are the benefits of using Git for simulation software development?
2. What types of files should you track in Git for a simulation project, and which should you avoid?
3. What are some best practices for writing commit messages in a scientific context?
4. What is a fork, and how does it differ from a branch in Git?
5. What is a typical workflow for contributing to an open-source project on GitHub or GitLab?

13.6. References and Further Reading

- Lambert's talk "Publishing Reproducible Numerics: A Student's Perspective"
- https://third-bit.com/py-rse/git-cmdline.html
- https://aeturrell.github.io/coding-for-economists/wrkflow-version-control.html
- MIT Missing semester: Git

14. Skills 03 - Gmsh Geometry for FEM

Programmatic CAD and boundary tagging for Exercise 4

14.1. Gmsh

For all grid-based methods such as the finite element method, we need to discretize the domain into a mesh. In this exercise, we will use Gmsh.

> **Why Gmsh?**
>
> There are many other mesh generators, also proprietary ones, but Gmsh is free, open-source and academically oriented (see, e.g., [12] or this talk), and it has a Python API that we can use to generate meshes directly from our Python code.

> **By the end of this skill**
>
> You should be able to create simple CAD geometry, assign physical tags for a PDE solver, export a `.msh` file, and check whether the mesh and boundary labels match the intended model.

The basic workflow is:

1. Create a geometrical model of the domain.
2. Define physical groups for boundaries, subdomains, or material regions.
3. Optionally define mesh properties such as element size, element type, or boundary layers.
4. Generate the mesh and write it to disk.

> **Kernel choice**
>
> Gmsh has two kernels: the built-in kernel and the OpenCASCADE (OCC) kernel. The built-in kernel is easier to use, but it is limited in terms of the geometrical features it can handle. The OCC kernel is more powerful and can handle more complex geometries, but it is also more complex to use.

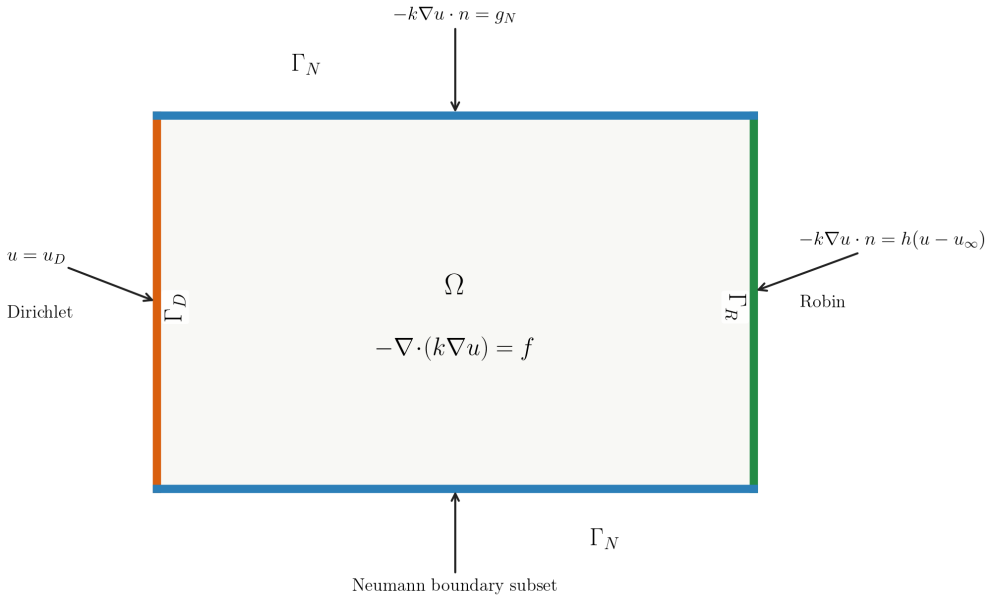


Figure 14.1.: An example geometrical model

14.2. Basics

The first examples use the OCC kernel because it is the better default once geometries involve boolean operations or imported CAD objects.

> **Minimal OCC checklist**
>
> 1. Create primitives (`addRectangle`, `addDisk`, …).
> 2. Combine with boolean operations (`cut`, `fuse`, `fragment`).
> 3. Synchronize via `gmsh.model.occ.synchronize()`.
> 4. Create physical groups for cells and facets.
> 5. Generate the mesh and write `.msh`.

14.2.1. Primitive Construction in Gmsh

The next cell uses Gmsh itself to add a rectangle, a standalone line, and two overlapping disks that are fused into one surface. It prints the entity list before and after `gmsh.model.occ.synchronize()`, then meshes the objects so the notebook output shows what changed visually.

> **Synchronization is not optional**
>
> OCC construction calls create CAD objects inside the OCC kernel first. Gmsh only sees them as model entities after `gmsh.model.occ.synchronize()`.

```
import gmsh
if gmsh.isInitialized():
    gmsh.finalize()
```

```
gmsh.initialize()
gmsh.option.setNumber("General.Terminal", 0)
gmsh.model.add("primitive_demo")

print("Visible model entities before adding primitives:", gmsh.model.getEntities())

# Surface primitive: a rectangle with lower-left corner (x, y, z) and size dx by dy.
rect = gmsh.model.occ.addRectangle(0.0, 0.0, 0.0, 2.0, 1.0)

# Curve primitive: a line needs two point tags first.
p1 = gmsh.model.occ.addPoint(2.55, 0.05, 0.0)
p2 = gmsh.model.occ.addPoint(4.15, 0.95, 0.0)
line = gmsh.model.occ.addLine(p1, p2)

# Boolean primitive operation: fuse two overlapping disks into one surface.
disk_a = gmsh.model.occ.addDisk(5.0, 0.5, 0.0, 0.45, 0.45)
disk_b = gmsh.model.occ.addDisk(5.55, 0.5, 0.0, 0.45, 0.45)
fused_disks, _ = gmsh.model.occ.fuse(
    [(2, disk_a)],
    [(2, disk_b)],
    removeObject=True,
    removeTool=True,
)
fused_disk_surfaces = [tag for dim, tag in fused_disks if dim == 2]

print("OCC calls returned these tags:")
print(f"  rectangle:       dim=2, tag={rect}")
print(f"  line:            dim=1, tag={line}, point tags=({p1}, {p2})")
print(f"  input disks:     dim=2, tags=({disk_a}, {disk_b})")
print(f"  fused disk area: dim=2, tags={fused_disk_surfaces}")
print("Visible model entities before synchronize:", gmsh.model.getEntities())

gmsh.model.occ.synchronize()
print("Visible model entities after synchronize:", gmsh.model.getEntities())

gmsh.option.setNumber("Mesh.MeshSizeMax", 0.12)
gmsh.model.mesh.generate(2)

gmsh.write("primitive_demo.msh")
gmsh.write("primitive_demo.geo_unrolled")
gmsh.write("primitive_demo.brep")

# gmsh.finalize()
```

```
Visible model entities before adding primitives: []
OCC calls returned these tags:
  rectangle:       dim=2, tag=1
  line:            dim=1, tag=5, point tags=(5, 6)
  input disks:     dim=2, tags=(2, 3)
  fused disk area: dim=2, tags=[2]
Visible model entities before synchronize: []
```

```
Visible model entities after synchronize: [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0,
```

```
import os

import matplotlib.pyplot as plt
import matplotlib.tri as mtri
import numpy as np

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

triangles = []
for etype, _, enodes in zip(*gmsh.model.mesh.getElements(2)):
    if etype == 2:  # 3-node triangle
        triangles.extend(np.array([node_index[t] for t in enodes]).reshape(-1, 3))

segments = []
for etype, _, enodes in zip(*gmsh.model.mesh.getElements(1)):
    if etype == 1:  # 2-node line segment
        segments.extend(np.array([node_index[t] for t in enodes]).reshape(-1, 2))

fig, ax = plt.subplots(figsize=(9, 3.4), constrained_layout=True)
if triangles:
    triang = mtri.Triangulation(points[:, 0], points[:, 1], np.array(triangles))
    ax.triplot(triang, lw=0.45, color="0.35")
    ax.tripcolor(triang, facecolors=np.ones(len(triangles)), alpha=0.18, cmap="Blues")

for segment in segments:
    xy = points[segment]
    ax.plot(xy[:, 0], xy[:, 1], color="#9a4d00", lw=1.3)

ax.text(0.15, 0.82, f"rectangle surface tag {rect}", color="#1f5a85", fontsize=10)
ax.text(2.7, 0.8, f"line curve tag {line}", color="#7b3f00", fontsize=10)
ax.text(4.72, 0.5, f"fused disk surface tag(s) {fused_disk_surfaces}", color="#2f6b24", fontsi
ax.set_aspect("equal")
ax.set_title("Gmsh OCC primitives and a fused disk pair after mesh generation")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.grid(True, color="0.9", lw=0.6)
plt.show()
```

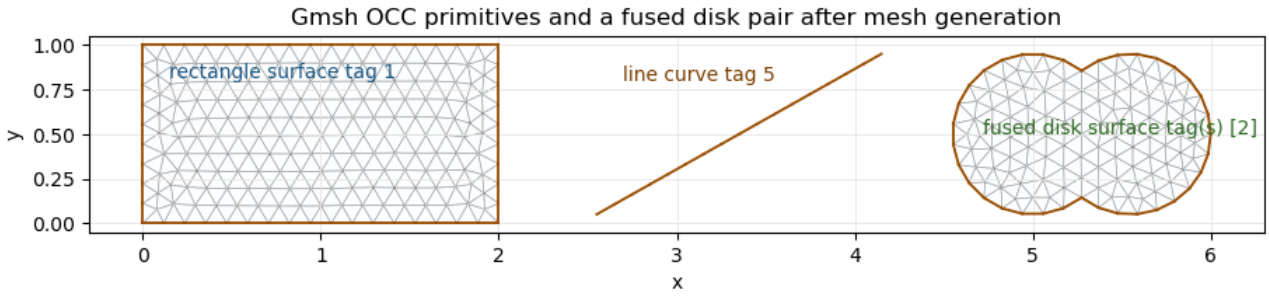


Figure 14.2.: Gmsh OCC primitives and a fused disk pair after mesh generation

14.2.2. Physical Tags for Use with the `.msh` Format

For FEniCSx/DOLFINx, the mesh alone is not enough. Boundary conditions and subdomain data should travel with the mesh as physical tags.

> **!** Tags are the solver contract
>
> `gmsh.model.addPhysicalGroup()` is how we mark cells and facets for later use. The dimension selects the entity type (`0` for points, `1` for curves, `2` for surfaces), the entity tags select the objects, and the optional physical name makes the file readable.

The next cell tags the rectangle as the domain and all boundary curves as one boundary group. Larger projects usually split the boundary into named parts such as `inlet`, `outlet`, and `wall`.

```
import gmsh

gmsh.initialize()
gmsh.model.add("demo")

rect = gmsh.model.occ.addRectangle(0.0, 0.0, 0.0, 2.0, 1.0)
gmsh.model.occ.synchronize()

boundary_curves = [tag for dim, tag in gmsh.model.getBoundary([(2, rect)], oriented=False)]

gmsh.model.addPhysicalGroup(2, [rect], tag=100)
gmsh.model.setPhysicalName(2, 100, "domain")
gmsh.model.addPhysicalGroup(1, boundary_curves, tag=1)
gmsh.model.setPhysicalName(1, 1, "boundary")

gmsh.model.mesh.generate(2)
gmsh.write("demo_rect_occ.msh")
gmsh.write("demo_rect_occ.geo_unrolled")
gmsh.finalize()
```

```
# show that physical groups are part of the msh file
with open("demo_rect_occ.msh", "r") as f:
    for line in f:
        if line.strip() == "$PhysicalNames":
            print("Found PhysicalNames section in msh file:")
```

```
            print(line.strip())
            # Print the next few lines to show the content of the PhysicalNames section
            for _ in range(5):
                print(f.readline().strip())
            break
```

```
Found PhysicalNames section in msh file:
$PhysicalNames
2
1 1 "boundary"
2 100 "domain"
$EndPhysicalNames
$Entities
```

> **Reading the mesh in DOLFINx**
>
> `dolfinx.io.gmshio.read_from_msh()` reads the mesh and returns the DOLFINx mesh together with cell tags and facet tags. Those tag objects are what boundary conditions and marked integrals will use later.

```
# show that FEniCSx can read the mesh and physical tags
import importlib
import numpy as np
from mpi4py import MPI

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")

mesh_data = gmshio.read_from_msh("demo_rect_occ.msh", MPI.COMM_WORLD, gdim=2)
if hasattr(mesh_data, "mesh"):
    msh, cell_tags, facet_tags = mesh_data.mesh, mesh_data.cell_tags, mesh_data.facet_tags
else:
    msh, cell_tags, facet_tags = mesh_data[:3]
print(f"Mesh dimension: {msh.topology.dim}")
print(f"Cell tag ids: {np.unique(cell_tags.values)}")
print(f"Facet tag ids: {np.unique(facet_tags.values)}")
```

```
Info    : Reading 'demo_rect_occ.msh'...
Info    : 9 entities
Info    : 207 nodes
Info    : 412 elements
```

```
Info    : Done reading 'demo_rect_occ.msh'
Mesh dimension: 2
Cell tag ids: [100]
Facet tag ids: [1]
```

14.2.3. Kernels: Built-in vs OCC

Gmsh can construct geometry through the built-in `geo` kernel or through the OpenCASCADE `occ` kernel. Both produce meshes, but their exported geometry text looks quite different.

> **OCC export is not a teaching format**
>
> For OCC models, `geo_unrolled` usually points to a CAD-style `.xao` file. This is useful for reproducibility, but it is not meant to be read like a hand-written `.geo` script.

```
# show geo_unrolled and xao file
with open("demo_rect_occ.geo_unrolled", "r") as f:
    print("Contents of demo_rect_occ.geo_unrolled:")
    for line in f:
        print(line.strip())
with open("demo_rect_occ.geo_unrolled.xao") as f:
    print("\nContents of demo_rect_occ.geo_unrolled.xao:")
    for _ in range(30):
        print(f.readline().strip())
    print("... (xao file continues)")
```

```
Contents of demo_rect_occ.geo_unrolled:
Merge "demo_rect_occ.geo_unrolled.xao";

Contents of demo_rect_occ.geo_unrolled.xao:
<?xml version="1.0" encoding="UTF-8"?>
<XAO version="1.0" author="Gmsh">
<geometry name="demo">
<shape format="BREP"><![CDATA[
CASCADE Topology V1, (c) Matra-Datavision
Locations 0
Curve2ds 0
Curves 4
1 0 0 0 1 0 0
1 2 0 0 0 1 0
1 2 1 0 -1 0 0
1 0 1 0 0 -1 0
Polygon3D 0
PolygonOnTriangulations 0
Surfaces 1
1 1 0.5 0 0 0 1 1 0 -0 -0 1 0
Triangulations 0

TShapes 11
Ve
```

```
1e-07
0 0 0
0 0

0101101
*
Ve
1e-07
2 0 0
0 0
... (xao file continues)
```

14.2.4. Built-in Kernel

The built-in kernel is more explicit: points, lines, curve loops, and plane surfaces are written in a way that maps directly to the `.geo` language.

> **When to use it**
>
> Use `gmsh.model.geo` for small instructional examples and simple planar geometries. Switch to `gmsh.model.occ` when boolean operations or imported CAD geometry become important.

```python
import gmsh

if gmsh.isInitialized():
    gmsh.finalize()

gmsh.initialize()
gmsh.model.add("geo_rectangle_demo")

lc = 0.1  # mesh size at points

p1 = gmsh.model.geo.addPoint(0.0, 0.0, 0.0, lc)
p2 = gmsh.model.geo.addPoint(1.0, 0.0, 0.0, lc)
p3 = gmsh.model.geo.addPoint(1.0, 1.0, 0.0, lc)
p4 = gmsh.model.geo.addPoint(0.0, 1.0, 0.0, lc)

l1 = gmsh.model.geo.addLine(p1, p2)
l2 = gmsh.model.geo.addLine(p2, p3)
l3 = gmsh.model.geo.addLine(p3, p4)
l4 = gmsh.model.geo.addLine(p4, p1)

curve_loop = gmsh.model.geo.addCurveLoop([l1, l2, l3, l4])
surface = gmsh.model.geo.addPlaneSurface([curve_loop])

# Sync CAD kernel data to the Gmsh model before meshing/exporting.
gmsh.model.geo.synchronize()

gmsh.model.addPhysicalGroup(2, [surface], tag=100)
gmsh.model.setPhysicalName(2, 100, "unit_square")
gmsh.model.addPhysicalGroup(1, [l1, l2, l3, l4], tag=1)
```

```
gmsh.model.setPhysicalName(1, 1, "boundary")

gmsh.model.mesh.generate(2)
gmsh.write("demo_rect.msh")
gmsh.write("demo_rect.geo_unrolled")

gmsh.finalize()
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.000139792s, CPU 0.000216s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.00135437s, CPU 0.001951s)
Info    : 142 nodes 286 elements
Info    : Writing 'demo_rect.msh'...
Info    : Done writing 'demo_rect.msh'
Info    : Writing 'demo_rect.geo_unrolled'...
Info    : Done writing 'demo_rect.geo_unrolled'
```

14.2.5. `geo_unrolled` File

With the built-in kernel, the unrolled geometry is much easier to inspect. This makes it a useful bridge between Python API calls and hand-written `.geo` files.

> **Reading the output**
>
> Look for the same construction pattern as in the Python code: characteristic length, points, lines, a curve loop, a plane surface, and physical groups.

```
with open("demo_rect.geo_unrolled", "r") as f:
    geo_content = f.read()
print("\nContents of demo_rect.geo_unrolled:\n")
print(geo_content)
```

```
Contents of demo_rect.geo_unrolled:

cl__1 = 0.1;
Point(1) = {0, 0, 0, cl__1};
Point(2) = {1, 0, 0, cl__1};
Point(3) = {1, 1, 0, cl__1};
Point(4) = {0, 1, 0, cl__1};
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
```

```
Line(4) = {4, 1};
Curve Loop(1) = {1, 2, 3, 4};
Plane Surface(1) = {1};
Physical Curve("boundary") = {1, 2, 3, 4};
Physical Surface("unit_square") = {1};
```

You can also mesh this `geo` or `geo_unrolled` file directly with the `gmsh` command-line tool:

```
gmsh -2 demo_rect.geo_unrolled -o output_filename.msh
```

14.2.6. Open the Gmsh FLTK Viewer (Interactive)

Use this for quick visual inspection of entities, physical groups, and mesh quality.

> Interactive cell
>
> The viewer opens a GUI window and blocks the notebook until the window is closed. Keep the cell disabled for automated rendering and enable it only when working locally.

```
import gmsh

if gmsh.isInitialized():
    gmsh.finalize()

gmsh.initialize()
gmsh.open("bridge_holes.msh")

# Optional display tweaks
gmsh.option.setNumber("Mesh.SurfaceEdges", 1)
gmsh.option.setNumber("Mesh.Lines", 1)

# Blocks until you close the FLTK window
# avoid this in automated workflows, but it's useful for interactive exploration and debugging
# gmsh.fltk.run()
# gmsh.finalize()
```

```
Info    : Reading 'bridge_holes.msh'...
Info    : 15 entities
Info    : 766 nodes
Info    : 1536 elements
Info    : Done reading 'bridge_holes.msh'
```

The interactive viewer should look similar to this:

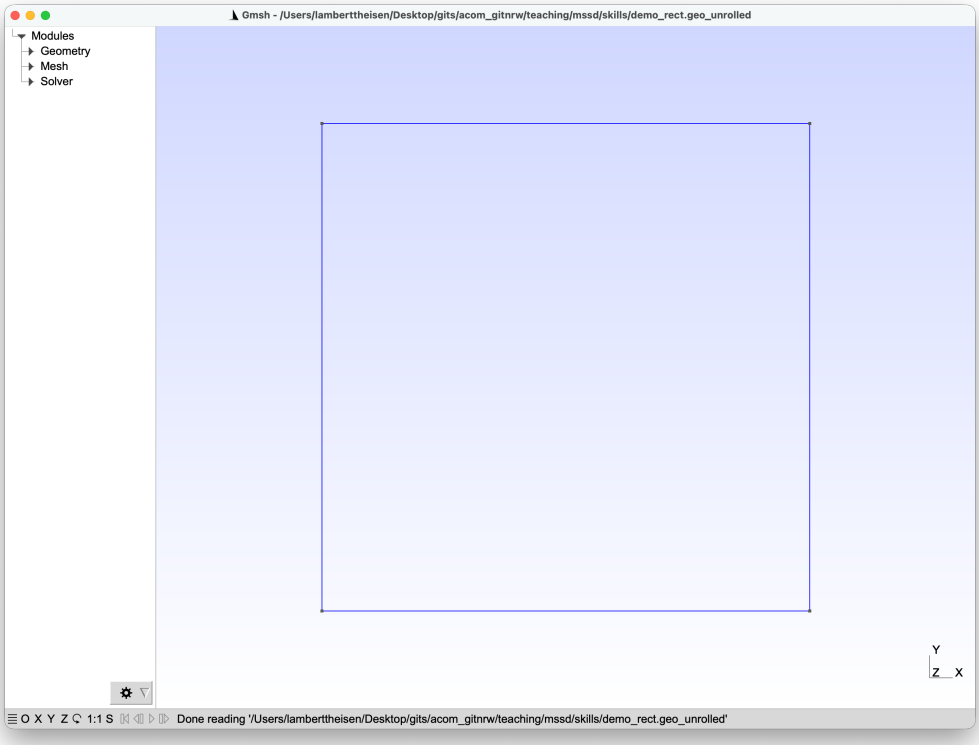


Figure 14.3.: GMSH View

14.3. Advanced Meshing

14.3.1. Reading a `.geo` File with parameters

The Python API is convenient for parametrized scripts, but many Gmsh examples use the native `.geo` language. The ideas are the same: points define geometry, curves connect points, curve loops define closed boundaries, surfaces define the 2D domain, and physical groups carry the tags used later in FEniCSx.

Here we write a small `.geo` file from the notebook so that students can inspect the text and immediately see the mesh it creates.

> **Source of the geometry**
>
> The example `.geo` file comes from the fenicsR13 solver [13]. More mesh examples are available in this repository.

```
from pathlib import Path

geo_text = r'''
// Command line parameter: try p = 0, 1, 2 to refine the mesh.
If(!Exists(p))
  p = 0;
EndIf
```

```
// Settings
res = 100;
Mesh.MeshSizeMax = 1.0 * 2^(-p);
Mesh.MshFileVersion = 2.0;

// Parameters
R1 = 0.5;
R2 = 2.0;

l = 1.0;
L = 2*l;

Point(1) = {0,  0, 0, res};
Point(2) = {l,  0, 0, res};
Point(3) = {-l, 0, 0, res};

Point(1010) = {l,   R1, 0, res};
Point(1011) = {-l,  R1, 0, res};
Point(1012) = {-l, -R1, 0, res};
Point(1013) = {l,  -R1, 0, res};

Point(1020) = {l,   R2, 0, res};
Point(1021) = {-l,  R2, 0, res};
Point(1022) = {-l, -R2, 0, res};
Point(1023) = {l,  -R2, 0, res};

Line(1010) = {1010,1011};
Circle(1011) = {1011,3,1012};
Line(1012) = {1012,1013};
Circle(1013) = {1013,2,1010};

Line(1020) = {1020,1021};
Circle(1021) = {1021,3,1022};
Line(1022) = {1022,1023};
Circle(1023) = {1023,2,1020};

// Physical curve tags are what the PDE code will see.
Physical Curve("it",3010) = {1010};
Physical Curve("il",3011) = {1011};
Physical Curve("ib",3012) = {1012};
Physical Curve("ir",3013) = {1013};

Physical Curve("ot",3020) = {1020};
Physical Curve("ol",3021) = {1021};
Physical Curve("ob",3022) = {1022};
Physical Curve("or",3023) = {1023};

Curve Loop(4010) = {1010,1011,1012,1013};
Curve Loop(4020) = {1020,1021,1022,1023};
Plane Surface(4000) = {4020,4010};
Physical Surface("mesh",4000) = {4000};
```

```
'''

Path("capsule_annulus.geo").write_text(geo_text)
print(geo_text)
```

```
// Command line parameter: try p = 0, 1, 2 to refine the mesh.
If(!Exists(p))
  p = 0;
EndIf

// Settings
res = 100;
Mesh.MeshSizeMax = 1.0 * 2^(-p);
Mesh.MshFileVersion = 2.0;

// Parameters
R1 = 0.5;
R2 = 2.0;

l = 1.0;
L = 2*l;

Point(1) = {0,  0, 0, res};
Point(2) = {l,  0, 0, res};
Point(3) = {-l, 0, 0, res};

Point(1010) = {l,   R1, 0, res};
Point(1011) = {-l,  R1, 0, res};
Point(1012) = {-l, -R1, 0, res};
Point(1013) = {l,  -R1, 0, res};

Point(1020) = {l,   R2, 0, res};
Point(1021) = {-l,  R2, 0, res};
Point(1022) = {-l, -R2, 0, res};
Point(1023) = {l,  -R2, 0, res};

Line(1010) = {1010,1011};
Circle(1011) = {1011,3,1012};
Line(1012) = {1012,1013};
Circle(1013) = {1013,2,1010};

Line(1020) = {1020,1021};
Circle(1021) = {1021,3,1022};
Line(1022) = {1022,1023};
Circle(1023) = {1023,2,1020};

// Physical curve tags are what the PDE code will see.
Physical Curve("it",3010) = {1010};
Physical Curve("il",3011) = {1011};
Physical Curve("ib",3012) = {1012};
```

```
Physical Curve("ir",3013) = {1013};

Physical Curve("ot",3020) = {1020};
Physical Curve("ol",3021) = {1021};
Physical Curve("ob",3022) = {1022};
Physical Curve("or",3023) = {1023};

Curve Loop(4010) = {1010,1011,1012,1013};
Curve Loop(4020) = {1020,1021,1022,1023};
Plane Surface(4000) = {4020,4010};
Physical Surface("mesh",4000) = {4000};
```

Before meshing, open the `.geo` file as geometry and plot the CAD curves. No call to `gmsh.model.mesh.generate(2)` happens in this cell.

> **Geometry before mesh**
>
> This separates two checks that are often mixed up: first verify that the CAD curves and physical names are correct, then generate and inspect the mesh.

```python
def read_triangles_from_msh(filename):
    if gmsh.isInitialized():
        gmsh.finalize()

    gmsh.initialize()
    gmsh.model.add("view_mesh")
    gmsh.merge(filename)

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

    triangles = []
    for _, entity_tag in gmsh.model.getEntities(2):
        element_types, _, element_nodes = gmsh.model.mesh.getElements(2, entity_tag)
        for e_type, e_nodes in zip(element_types, element_nodes):
            # type 2: linear triangles, type 9: quadratic triangles
            if e_type == 2:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 3)
            elif e_type == 9:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 6)[:, :3]
            else:
                continue
            triangles.append(np.vectorize(node_to_local.get)(conn))

    gmsh.finalize()

    if len(triangles) == 0:
        raise RuntimeError(f"No triangular elements found in {filename}")

    tri = np.vstack(triangles)
```

```
    return xy, tri


def plot_msh(ax, filename, title):
    xy, tri = read_triangles_from_msh(filename)
    triang = mtri.Triangulation(xy[:, 0], xy[:, 1], tri)
    ax.triplot(triang, lw=0.35, color="0.2")
    ax.set_aspect("equal")
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
```

```
def sample_gmsh_curve(tag, n=120):
    lower, upper = gmsh.model.getParametrizationBounds(1, tag)
    params = np.linspace(float(lower[0]), float(upper[0]), n)
    points = np.array([gmsh.model.getValue(1, tag, [u]) for u in params])
    return points[:, :2]


def plot_current_gmsh_geometry(ax, title, show_entity_tags=True):
    cmap = plt.get_cmap("tab10")
    physical_curve_to_color = {}
    physical_curve_to_label = {}

    for color_i, (dim, ptag) in enumerate(gmsh.model.getPhysicalGroups(1)):
        name = gmsh.model.getPhysicalName(dim, ptag) or f"physical {ptag}"
        color = cmap(color_i % 10)
        for curve_tag in gmsh.model.getEntitiesForPhysicalGroup(dim, ptag):
            physical_curve_to_color[curve_tag] = color
            physical_curve_to_label[curve_tag] = f"{ptag}: {name}"

    shown_labels = set()
    for _, curve_tag in gmsh.model.getEntities(1):
        xy = sample_gmsh_curve(curve_tag)
        color = physical_curve_to_color.get(curve_tag, "0.25")
        label = physical_curve_to_label.get(curve_tag)
        ax.plot(
            xy[:, 0],
            xy[:, 1],
            lw=2.0,
            color=color,
            label=label if label and label not in shown_labels else None,
        )
        if label:
            shown_labels.add(label)

        if show_entity_tags:
            mid = xy[len(xy) // 2]
            ax.text(mid[0], mid[1], str(curve_tag), fontsize=8, ha="center", va="center")

    point_xy = []
```

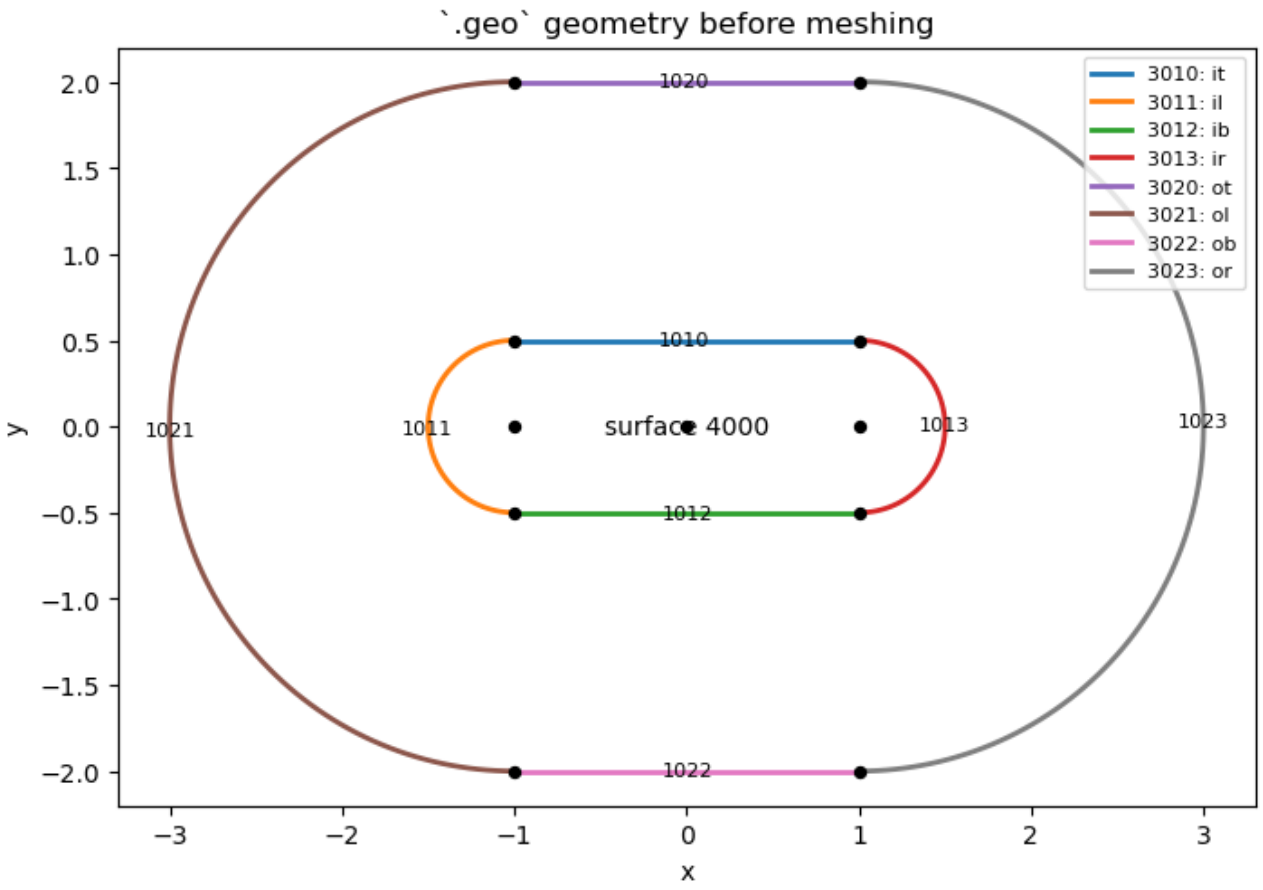
```
    for _, point_tag in gmsh.model.getEntities(0):
        point_xy.append(gmsh.model.getValue(0, point_tag, []))
    if point_xy:
        point_xy = np.array(point_xy)[:, :2]
        ax.scatter(point_xy[:, 0], point_xy[:, 1], s=18, color="black", zorder=3)

    for _, surface_tag in gmsh.model.getEntities(2):
        xmin, ymin, _, xmax, ymax, _ = gmsh.model.getBoundingBox(2, surface_tag)
        x, y = 0.5 * (xmin + xmax), 0.5 * (ymin + ymax)
        ax.text(x, y, f"surface {surface_tag}", fontsize=10, ha="center", va="center")

    ax.set_aspect("equal")
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    if shown_labels:
        ax.legend(loc="upper right", fontsize=8, frameon=True)
```

```
def plot_msh_with_physical_tags(ax, filename, title):
    if gmsh.isInitialized():
        gmsh.finalize()

    gmsh.initialize()
    gmsh.model.add("tag_view")
    gmsh.merge(filename)

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

    triangles = []
    for _, entity_tag in gmsh.model.getEntities(2):
        element_types, _, element_nodes = gmsh.model.mesh.getElements(2, entity_tag)
        for e_type, e_nodes in zip(element_types, element_nodes):
            if e_type == 2:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 3)
            elif e_type == 9:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 6)[:, :3]
            else:
                continue
            triangles.append(np.vectorize(node_to_local.get)(conn))

    if triangles:
        tri = np.vstack(triangles)
        triang = mtri.Triangulation(xy[:, 0], xy[:, 1], tri)
        ax.triplot(triang, lw=0.25, color="0.75")

    cmap = plt.get_cmap("tab10")
    color_i = 0
    for dim, ptag in gmsh.model.getPhysicalGroups():
        if dim != 1:
```

```
            continue
        pname = gmsh.model.getPhysicalName(dim, ptag) or f"tag {ptag}"
        color = cmap(color_i % 10)
        color_i += 1

        shown_label = False
        for ent in gmsh.model.getEntitiesForPhysicalGroup(dim, ptag):
            etypes, _, enodes = gmsh.model.mesh.getElements(1, ent)
            for et, e in zip(etypes, enodes):
                nper = gmsh.model.mesh.getElementProperties(et)[3]
                conn = np.array(e, dtype=np.int64).reshape(-1, nper)
                for row in conn:
                    n0, n1 = int(row[0]), int(row[-1])
                    i0, i1 = node_to_local[n0], node_to_local[n1]
                    ax.plot(
                        [xy[i0, 0], xy[i1, 0]],
                        [xy[i0, 1], xy[i1, 1]],
                        color=color,
                        lw=2.0,
                        label=f"{ptag}: {pname}" if not shown_label else None,
                    )
                    shown_label = True

    gmsh.finalize()

    ax.set_aspect("equal")
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.legend(loc="upper right", fontsize=9, frameon=True)
```

```
if gmsh.isInitialized():
    gmsh.finalize()

gmsh.initialize()
gmsh.open("capsule_annulus.geo")

fig, ax = plt.subplots(figsize=(7, 7), constrained_layout=True)
plot_current_gmsh_geometry(ax, "`.geo` geometry before meshing")
plt.show()

gmsh.finalize()
```

```
Info    : Reading 'capsule_annulus.geo'...
Info    : Done reading 'capsule_annulus.geo'
```



The parameter `p` controls the largest allowed element size. In the command line this would be `gmsh capsule_annulus.geo -2 -setnumber p 1`. In the notebook we read the `.geo` file and then set the same mesh-size option explicitly, because this makes the visual comparison independent of command-line execution.

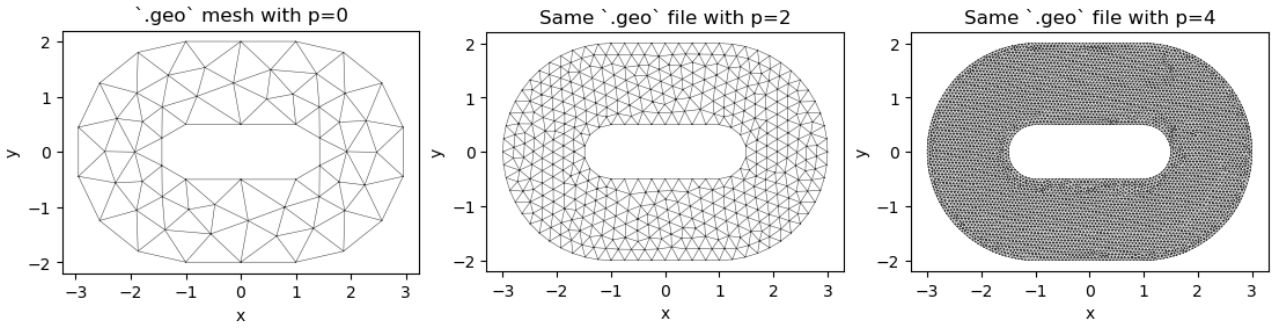
> **Refinement convention**
>
> Here larger `p` means smaller elements because the notebook uses `1.0 * 2 ** (-p)` as the maximum mesh size. The three plots below show the same geometry at increasing resolution.

```
def mesh_from_geo(filename, p=0, msh_filename=None):
    if gmsh.isInitialized():
        gmsh.finalize()

    if msh_filename is None:
        msh_filename = f"{Path(filename).stem}_p{p}.msh"

    gmsh.initialize()
    gmsh.open(filename)
    gmsh.option.setNumber("Mesh.MeshSizeMax", 1.0 * 2 ** (-p))
    gmsh.model.mesh.generate(2)
    gmsh.write(msh_filename)
    gmsh.finalize()
    return msh_filename
```

```
geo_mesh_p0 = mesh_from_geo("capsule_annulus.geo", p=0)
geo_mesh_p2 = mesh_from_geo("capsule_annulus.geo", p=2)
geo_mesh_p4 = mesh_from_geo("capsule_annulus.geo", p=4)

fig, axes = plt.subplots(1, 3, figsize=(11, 5), constrained_layout=True)
plot_msh(axes[0], geo_mesh_p0, "`.geo` mesh with p=0")
plot_msh(axes[1], geo_mesh_p2, "Same `.geo` file with p=2")
plot_msh(axes[2], geo_mesh_p4, "Same `.geo` file with p=4")
plt.show()
```

```
Info    : Reading 'capsule_annulus.geo'...
Info    : Done reading 'capsule_annulus.geo'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1010 (Line)
Info    : [ 20%] Meshing curve 1011 (Circle)
Info    : [ 30%] Meshing curve 1012 (Line)
Info    : [ 40%] Meshing curve 1013 (Circle)
Info    : [ 60%] Meshing curve 1020 (Line)
Info    : [ 70%] Meshing curve 1021 (Circle)
Info    : [ 80%] Meshing curve 1022 (Line)
Info    : [ 90%] Meshing curve 1023 (Circle)
Info    : Done meshing 1D (Wall 0.000333792s, CPU 0.000442s)
Info    : Meshing 2D...
Info    : Meshing surface 4000 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.000708208s, CPU 0.001054s)
Info    : 67 nodes 139 elements
Info    : Writing 'capsule_annulus_p0.msh'...
Info    : Done writing 'capsule_annulus_p0.msh'
Info    : Reading 'capsule_annulus.geo'...
Info    : Done reading 'capsule_annulus.geo'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1010 (Line)
Info    : [ 20%] Meshing curve 1011 (Circle)
Info    : [ 30%] Meshing curve 1012 (Line)
Info    : [ 40%] Meshing curve 1013 (Circle)
Info    : [ 60%] Meshing curve 1020 (Line)
Info    : [ 70%] Meshing curve 1021 (Circle)
Info    : [ 80%] Meshing curve 1022 (Line)
Info    : [ 90%] Meshing curve 1023 (Circle)
Info    : Done meshing 1D (Wall 0.000259791s, CPU 0.000353s)
Info    : Meshing 2D...
Info    : Meshing surface 4000 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.00462625s, CPU 0.007416s)
Info    : 445 nodes 895 elements
Info    : Writing 'capsule_annulus_p2.msh'...
Info    : Done writing 'capsule_annulus_p2.msh'
Info    : Reading 'capsule_annulus.geo'...
Info    : Done reading 'capsule_annulus.geo'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1010 (Line)
```

```
Info    : [ 20%] Meshing curve 1011 (Circle)
Info    : [ 30%] Meshing curve 1012 (Line)
Info    : [ 40%] Meshing curve 1013 (Circle)
Info    : [ 60%] Meshing curve 1020 (Line)
Info    : [ 70%] Meshing curve 1021 (Circle)
Info    : [ 80%] Meshing curve 1022 (Line)
Info    : [ 90%] Meshing curve 1023 (Circle)
Info    : Done meshing 1D (Wall 0.0003335s, CPU 0.000589s)
Info    : Meshing 2D...
Info    : Meshing surface 4000 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.0655314s, CPU 0.106388s)
Info    : 5647 nodes 11299 elements
Info    : Writing 'capsule_annulus_p4.msh'...
Info    : Done writing 'capsule_annulus_p4.msh'
Info    : Reading 'capsule_annulus_p0.msh'...
Info    : 64 nodes
Info    : 128 elements
Info    : Done reading 'capsule_annulus_p0.msh'
Info    : Reading 'capsule_annulus_p2.msh'...
Info    : 442 nodes
Info    : 884 elements
Info    : Done reading 'capsule_annulus_p2.msh'
Info    : Reading 'capsule_annulus_p4.msh'...
Info    : 5644 nodes
Info    : 11288 elements
Info    : Done reading 'capsule_annulus_p4.msh'
```



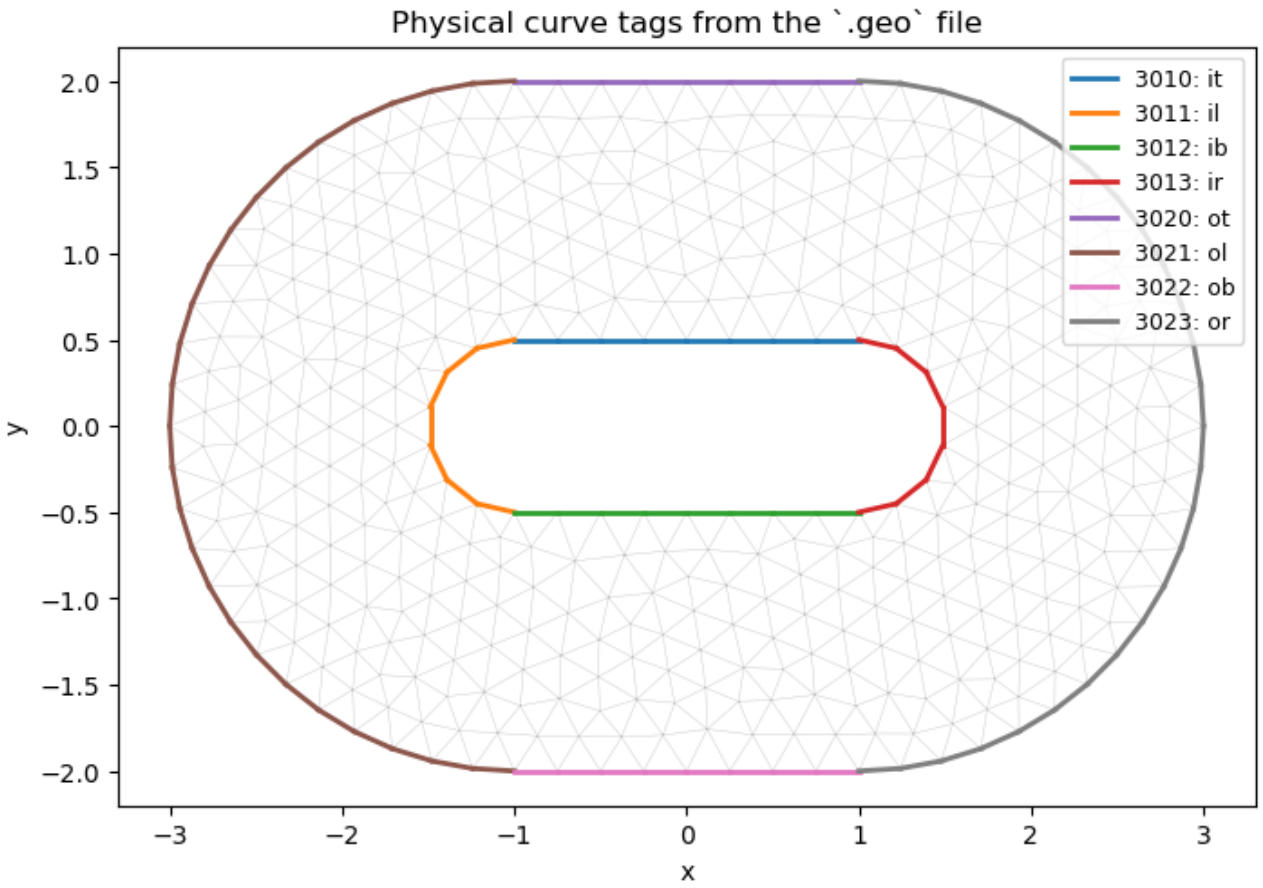
The plot below overlays the physical curve names from the `.geo` file. This is the important check before using the mesh in a PDE solver: the labels must match the intended boundary conditions.

> **Inspecting tags interactively**
>
> Alternatively, you can hover over the curves in the Gmsh FLTK viewer to see the tags and physical group names.

```
fig, ax = plt.subplots(figsize=(7, 7), constrained_layout=True)
plot_msh_with_physical_tags(ax, geo_mesh_p2, "Physical curve tags from the `.geo` file")
plt.show()
```

```
Info    : Reading 'capsule_annulus_p2.msh'...
```

```
Info    : 442 nodes
Info    : 884 elements
Info    : Done reading 'capsule_annulus_p2.msh'
```



14.3.2. Mesh-Size Fields in the Python API

Global mesh sizes are often too blunt. A mesh-size field lets us ask Gmsh for smaller elements near selected entities and larger elements farther away. The next function refines near the boundary of a square using a `Distance` field followed by a `Threshold` field.

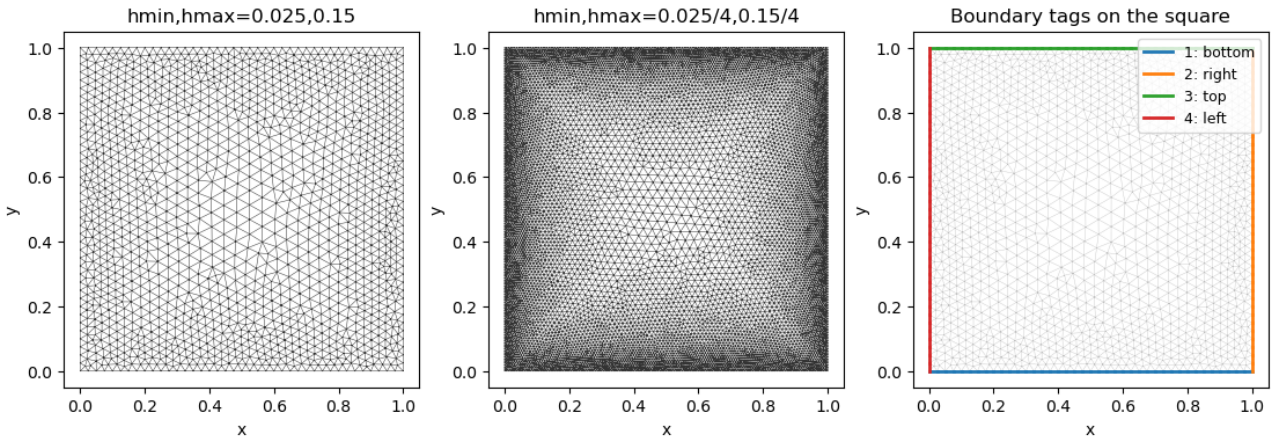
> **Field recipe**
>
> Important details to notice:
>
> - `CurvesList` selects the boundary curves that generate the distance field.
> - `DistMin` and `DistMax` describe the transition zone from fine to coarse mesh.
> - `MeshSizeExtendFromBoundary`, `MeshSizeFromPoints`, and `MeshSizeFromCurvature` are turned off so that the background field is the main size rule.

```
def create_square_with_boundary_refinement(filename="square_boundary_refined.msh", hmin=0.025,
    if gmsh.isInitialized():
        gmsh.finalize()

    gmsh.initialize()
```

```
    gmsh.model.add("square_boundary_refined")

    rect = gmsh.model.occ.addRectangle(0, 0, 0, 1, 1)
    gmsh.model.occ.synchronize()

    boundary_lines = gmsh.model.getBoundary([(2, rect)], oriented=False)
    boundary_line_tags = [tag for dim, tag in boundary_lines if dim == 1]

    gmsh.model.mesh.field.add("Distance", 1)
    gmsh.model.mesh.field.setNumbers(1, "CurvesList", boundary_line_tags)
    gmsh.model.mesh.field.setNumber(1, "Sampling", max(2.0 / hmin, 100))

    gmsh.model.mesh.field.add("Threshold", 2)
    gmsh.model.mesh.field.setNumber(2, "InField", 1)
    gmsh.model.mesh.field.setNumber(2, "SizeMin", hmin)
    gmsh.model.mesh.field.setNumber(2, "SizeMax", hmax)
    gmsh.model.mesh.field.setNumber(2, "DistMin", 3 * hmin)
    gmsh.model.mesh.field.setNumber(2, "DistMax", 16 * hmax)
    gmsh.model.mesh.field.setAsBackgroundMesh(2)

    gmsh.option.setNumber("Mesh.MeshSizeExtendFromBoundary", 0)
    gmsh.option.setNumber("Mesh.MeshSizeFromPoints", 0)
    gmsh.option.setNumber("Mesh.MeshSizeFromCurvature", 0)

    gmsh.model.addPhysicalGroup(2, [rect], tag=1)
    gmsh.model.setPhysicalName(2, 1, "Omega")

    labels = {"bottom": 1, "right": 2, "top": 3, "left": 4}
    for dim, tag in boundary_lines:
        x, y, _ = gmsh.model.occ.getCenterOfMass(dim, tag)
        if abs(y - 0.0) < 1e-12:
            name = "bottom"
        elif abs(x - 1.0) < 1e-12:
            name = "right"
        elif abs(y - 1.0) < 1e-12:
            name = "top"
        else:
            name = "left"
        gmsh.model.addPhysicalGroup(1, [tag], tag=labels[name])
        gmsh.model.setPhysicalName(1, labels[name], name)

    gmsh.model.mesh.generate(2)
    gmsh.write(filename)
    gmsh.finalize()
    return filename


square_field_mesh = create_square_with_boundary_refinement(filename="square_field_mesh.msh", h
square_field_mesh2 = create_square_with_boundary_refinement(filename="square_field_mesh2.msh",

fig, axes = plt.subplots(1, 3, figsize=(11, 5), constrained_layout=True)
```

```
plot_msh(axes[0], square_field_mesh, "hmin,hmax=0.025,0.15")
plot_msh(axes[1], square_field_mesh2, "hmin,hmax=0.025/4,0.15/4")
plot_msh_with_physical_tags(axes[2], square_field_mesh, "Boundary tags on the square")
plt.show()
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.0001985s, CPU 0.000335s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.02453s, CPU 0.036586s)
Info    : 1483 nodes 2968 elements
Info    : Writing 'square_field_mesh.msh'...
Info    : Done writing 'square_field_mesh.msh'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.000312417s, CPU 0.000447s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.184211s, CPU 0.298308s)
Info    : 10843 nodes 21688 elements
Info    : Writing 'square_field_mesh2.msh'...
Info    : Done writing 'square_field_mesh2.msh'
Info    : Reading 'square_field_mesh.msh'...
Info    : 9 entities
Info    : 1483 nodes
Info    : 2964 elements
Info    : Done reading 'square_field_mesh.msh'
Info    : Reading 'square_field_mesh2.msh'...
Info    : 9 entities
Info    : 10843 nodes
Info    : 21684 elements
Info    : Done reading 'square_field_mesh2.msh'
Info    : Reading 'square_field_mesh.msh'...
Info    : 9 entities
Info    : 1483 nodes
Info    : 2964 elements
Info    : Done reading 'square_field_mesh.msh'
```



14.3.3. Quadrilateral and Structured Meshes

Gmsh is not limited to triangular meshes. For many rectangular or logically rectangular geometries, quadrilateral elements are also possible. The usual Gmsh word for converting triangles into quads is **recombination**.

> Two separate ideas
>
> `setRecombine()` asks Gmsh for quadrilateral elements. `setTransfiniteCurve()` and `setTransfiniteSurface()` additionally impose a structured, grid-like node layout.

The next example compares the same square as an unstructured triangular mesh, a recombined quadrilateral mesh, a structured transfinite quadrilateral mesh, and a structured transfinite triangular mesh.

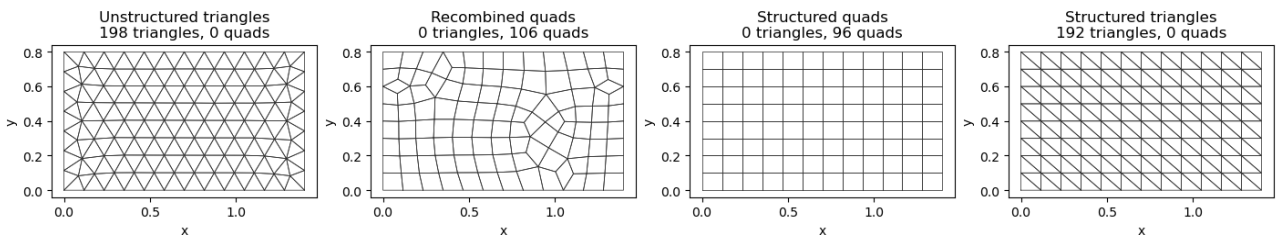
```
def create_square_mesh(filename, recombine=False, structured=False, nx=13, ny=9, lc=0.12):
    if gmsh.isInitialized():
        gmsh.finalize()

    gmsh.initialize()
    gmsh.model.add(Path(filename).stem)

    p1 = gmsh.model.geo.addPoint(0.0, 0.0, 0.0, lc)
    p2 = gmsh.model.geo.addPoint(1.4, 0.0, 0.0, lc)
    p3 = gmsh.model.geo.addPoint(1.4, 0.8, 0.0, lc)
    p4 = gmsh.model.geo.addPoint(0.0, 0.8, 0.0, lc)

    l1 = gmsh.model.geo.addLine(p1, p2)
    l2 = gmsh.model.geo.addLine(p2, p3)
    l3 = gmsh.model.geo.addLine(p3, p4)
    l4 = gmsh.model.geo.addLine(p4, p1)
    loop = gmsh.model.geo.addCurveLoop([l1, l2, l3, l4])
    surface = gmsh.model.geo.addPlaneSurface([loop])

    if structured:
        gmsh.model.geo.mesh.setTransfiniteCurve(l1, nx)
        gmsh.model.geo.mesh.setTransfiniteCurve(l3, nx)
```

```
        gmsh.model.geo.mesh.setTransfiniteCurve(l2, ny)
        gmsh.model.geo.mesh.setTransfiniteCurve(l4, ny)
        gmsh.model.geo.mesh.setTransfiniteSurface(surface)

    if recombine:
        gmsh.model.geo.mesh.setRecombine(2, surface)

    gmsh.model.geo.synchronize()
    gmsh.model.addPhysicalGroup(2, [surface], tag=1)
    gmsh.model.setPhysicalName(2, 1, "Omega")
    gmsh.model.addPhysicalGroup(1, [l1, l2, l3, l4], tag=2)
    gmsh.model.setPhysicalName(1, 2, "boundary")

    gmsh.option.setNumber("Mesh.RecombinationAlgorithm", 1)
    gmsh.model.mesh.generate(2)
    gmsh.write(filename)
    gmsh.finalize()
    return filename


def read_triangles_and_quads_from_msh(filename):
    if gmsh.isInitialized():
        gmsh.finalize()

    gmsh.initialize()
    gmsh.model.add("view_quads")
    gmsh.merge(filename)

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

    cells = []
    counts = {"triangles": 0, "quads": 0}
    for _, entity_tag in gmsh.model.getEntities(2):
        element_types, _, element_nodes = gmsh.model.mesh.getElements(2, entity_tag)
        for e_type, e_nodes in zip(element_types, element_nodes):
            if e_type == 2:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 3)
                counts["triangles"] += len(conn)
            elif e_type == 3:
                conn = np.array(e_nodes, dtype=np.int64).reshape(-1, 4)
                counts["quads"] += len(conn)
            else:
                continue
            cells.extend(np.vectorize(node_to_local.get)(conn))

    gmsh.finalize()
    return xy, cells, counts
```

```
def plot_cells(ax, filename, title):
    xy, cells, counts = read_triangles_and_quads_from_msh(filename)
    for cell in cells:
        polygon = xy[np.r_[cell, cell[0]]]
        ax.plot(polygon[:, 0], polygon[:, 1], lw=0.55, color="0.2")
    ax.set_aspect("equal")
    ax.set_title(f"{title}\n{counts['triangles']} triangles, {counts['quads']} quads")
    ax.set_xlabel("x")
    ax.set_ylabel("y")


tri_square = create_square_mesh("square_triangles.msh")
quad_square = create_square_mesh("square_recombined_quads.msh", recombine=True)
structured_square = create_square_mesh("square_structured_quads.msh", structured=True, recombi
structured_triangles = create_square_mesh("square_structured_triangles.msh", structured=True)

fig, axes = plt.subplots(1, 4, figsize=(14, 4.2), constrained_layout=True)
plot_cells(axes[0], tri_square, "Unstructured triangles")
plot_cells(axes[1], quad_square, "Recombined quads")
plot_cells(axes[2], structured_square, "Structured quads")
plot_cells(axes[3], structured_triangles, "Structured triangles")
plt.show()
```

```
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.000110375s, CPU 0.000209s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.00108654s, CPU 0.001635s)
Info    : 119 nodes 240 elements
Info    : Writing 'square_triangles.msh'...
Info    : Done writing 'square_triangles.msh'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.000107666s, CPU 0.000161s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Blossom: 301 internal 40 closed
Info    : Blossom recombination completed (Wall 0.000842375s, CPU 0.00121s): 106 quads, 0 tria
Info    : Done meshing 2D (Wall 0.00201304s, CPU 0.002895s)
Info    : 127 nodes 150 elements
Info    : Writing 'square_recombined_quads.msh'...
Info    : Done writing 'square_recombined_quads.msh'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
```

```
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 8.4542e-05s, CPU 0.00011s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Transfinite)
Info    : Done meshing 2D (Wall 2.3541e-05s, CPU 3.1e-05s)
Info    : 117 nodes 140 elements
Info    : Writing 'square_structured_quads.msh'...
Info    : Done writing 'square_structured_quads.msh'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 8.075e-05s, CPU 0.000119s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Transfinite)
Info    : Done meshing 2D (Wall 2.0167e-05s, CPU 3.1e-05s)
Info    : 117 nodes 236 elements
Info    : Writing 'square_structured_triangles.msh'...
Info    : Done writing 'square_structured_triangles.msh'
Info    : Reading 'square_triangles.msh'...
Info    : 9 entities
Info    : 119 nodes
Info    : 236 elements
Info    : Done reading 'square_triangles.msh'
Info    : Reading 'square_recombined_quads.msh'...
Info    : 9 entities
Info    : 127 nodes
Info    : 146 elements
Info    : Done reading 'square_recombined_quads.msh'
Info    : Reading 'square_structured_quads.msh'...
Info    : 9 entities
Info    : 117 nodes
Info    : 136 elements
Info    : Done reading 'square_structured_quads.msh'
Info    : Reading 'square_structured_triangles.msh'...
Info    : 9 entities
Info    : 117 nodes
Info    : 232 elements
Info    : Done reading 'square_structured_triangles.msh'
```



14.3.4. 3D Example

The same `.geo` language can describe 3D construction. This example sweeps two cross-sections along a spline wire to create a curved pipe-like geometry.

> **From 2D to 3D**
>
> The important new idea is `Extrude ... Using Wire`: instead of filling a plane surface, Gmsh moves a surface along a path and creates a volume-like object.

```
from pathlib import Path

geo_text = r'''
// from https://gitlab.onelab.info/gmsh/gmsh/-/blob/master/examples/boolean/pipe.geo
SetFactory("OpenCASCADE");

Mesh.MeshSizeMin = 0.1;
Mesh.MeshSizeMax = 0.1;
Geometry.NumSubEdges = 100; // nicer display of curve

nturns = DefineNumber[ 1, Min 0.1, Max 1, Step 0.01, Name "Parameters/Turn" ];
npts = 20;
r = 1;
rd = 0.1;
h = 1 * nturns;

For i In {0:npts-1}
  theta = i * 2*Pi*nturns/npts;
  Point(i + 1) = {r * Cos(theta), r * Sin(theta), i * h/npts};
EndFor

Spline(1) = {1:npts};
Wire(1) = {1};

Disk(1) = {1,0,0, rd};

Rectangle(2) = {1+2*rd,-rd,0, 2*rd,2*rd,rd/5};
Rotate {{1, 0, 0}, {0, 0, 0}, Pi/2} { Surface{1,2}; }

Extrude { Surface{1,2}; } Using Wire {1}
Delete{ Surface{1,2}; }
'''

Path("pipe.geo").write_text(geo_text)
print(geo_text)
```

```
// from https://gitlab.onelab.info/gmsh/gmsh/-/blob/master/examples/boolean/pipe.geo
SetFactory("OpenCASCADE");

Mesh.MeshSizeMin = 0.1;
```

```
Mesh.MeshSizeMax = 0.1;
Geometry.NumSubEdges = 100; // nicer display of curve

nturns = DefineNumber[ 1, Min 0.1, Max 1, Step 0.01, Name "Parameters/Turn" ];
npts = 20;
r = 1;
rd = 0.1;
h = 1 * nturns;

For i In {0:npts-1}
  theta = i * 2*Pi*nturns/npts;
  Point(i + 1) = {r * Cos(theta), r * Sin(theta), i * h/npts};
EndFor

Spline(1) = {1:npts};
Wire(1) = {1};

Disk(1) = {1,0,0, rd};

Rectangle(2) = {1+2*rd,-rd,0, 2*rd,2*rd,rd/5};
Rotate {{1, 0, 0}, {0, 0, 0}, Pi/2} { Surface{1,2}; }

Extrude { Surface{1,2}; } Using Wire {1}
Delete{ Surface{1,2}; }
```

The rendered geometry looks like this:

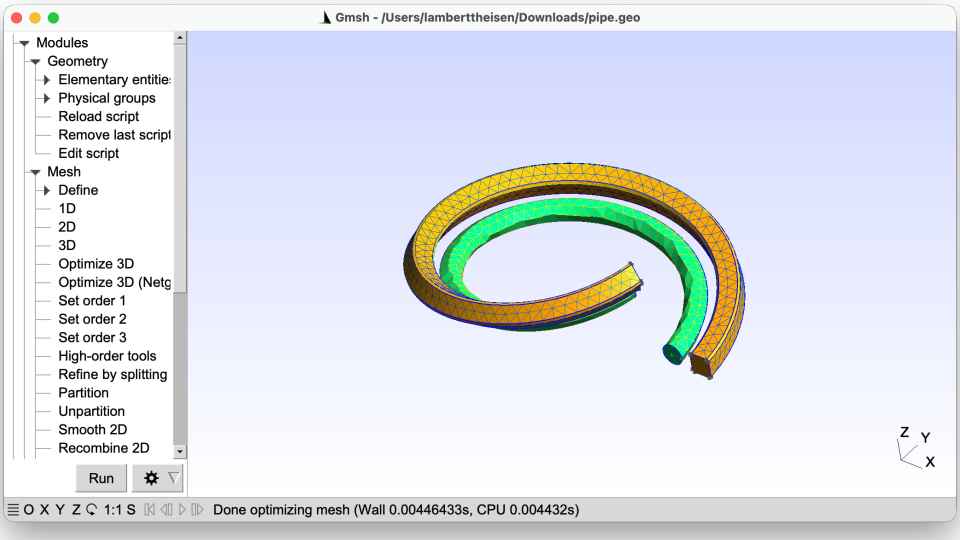


Figure 14.4.: 3D Mesh in Gmsh

14.4. Further Reading

Where to go next

Use the official documentation for API details, then compare with domain-specific tutorials when importing tagged meshes into FEniCSx/DOLFINx.

- Gmsh Documentation
- Geo-file example from Gmsh Git
- Geo-files from F. Cuvelier
- Talk by C. Geuzaine and J.-F. Remacle
- Course by C. Geuzaine and J.-F. Remacle
- nice tutorial by J. Dokken

15. Skills 04 - ParaView for FEM Output

Visual inspection, pipeline thinking, and reproducible figures

15.1. Introduction

Finite-element simulations do not end when the linear system has been solved. Before we interpret numbers, export figures, or compare designs, we need to inspect the computed field on the actual mesh. ParaView is a standard tool for this step: it is free, open-source, scriptable, and can handle data sets that are much larger than what we want to load into a notebook.

In this course, ParaView has three jobs:

1. Check whether the mesh and boundary tags look like the problem we meant to solve.
2. Inspect scalar and vector fields without hiding discretization artifacts.
3. Produce reproducible screenshots, line plots, animations, and exported data for reports (with `pvpython`).

> **Note**
>
> ParaView is not a replacement for quantitative verification. It is a fast way to see mistakes that would otherwise survive for a long time: swapped boundaries, wrong units, clipped ranges, discontinuities, bad refinement zones, or a solution plotted on the wrong data association.

15.2. Topics

After this notebook you should be able to:

1. Open `.vtu`, `.pvd`, and `.xdmf` output from a FEM code.
2. Explain the difference between the **Pipeline Browser**, **Properties**, and **Render View**.
3. Use `Apply`, visibility, coloring, rescaling, and representation modes intentionally.
4. Build common filters: `Calculator`, `Contour`, `Warp By Scalar`, `Extract Surface`, `Feature Edges`, `Plot Over Line`, and `Probe Location`.
5. Export a figure, a CSV line sample, or a saved `.pvsm` state in a reproducible way.

15.3. Create a Small FEM-like Data Set

The next cell creates a tiny triangular mesh and writes a time series that ParaView can open. This is not a PDE solve; it is a controlled visualization data set with:

- `temperature` as point data,
- `heat_flux` as vector point data,
- `material_id` as cell data,
- one `.pvd` file that collects several `.vtu` time steps.

Run the cell, then open `paraview_output/heat_demo.pvd` in ParaView.

> **Note**
>
> This is a minimal example. In practice, you create the file as a result of a PDE solve. This might only be used later for time series data.

```
from pathlib import Path
import numpy as np

out_dir = Path("paraview_output")
out_dir.mkdir(exist_ok=True)

def vtk_values(values):
    values = np.asarray(values)
    if values.ndim == 1:
        return " ".join(map(str, values.tolist()))
    return " ".join(map(str, values.reshape(-1).tolist()))

def data_array(name, values, vtk_type="Float64", components=None):
    component_attr = f' NumberOfComponents="{components}"' if components else ""
    return (
        f'        <DataArray type="{vtk_type}" Name="{name}"{component_attr} format="ascii">\n
        f'          {vtk_values(values)}\n'
        f'        </DataArray>'
    )

def write_vtu(filename, points, triangles, temperature, heat_flux, material_id):
    n_cells = len(triangles)
    connectivity = triangles.reshape(-1)
    offsets = 3 * np.arange(1, n_cells + 1, dtype=np.int64)
    cell_types = np.full(n_cells, 5, dtype=np.uint8)  # VTK_TRIANGLE
    lines = [
        '<?xml version="1.0"?>',
        '<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">',
        '  <UnstructuredGrid>',
        f'    <Piece NumberOfPoints="{len(points)}" NumberOfCells="{n_cells}">',
        '      <PointData Scalars="temperature" Vectors="heat_flux">',
        data_array("temperature", temperature),
        data_array("heat_flux", heat_flux, components=3),
        '      </PointData>',
```

```
        '      <CellData Scalars="material_id">',
        data_array("material_id", material_id, vtk_type="Int32"),
        '      </CellData>',
        '      <Points>',
        data_array("Points", points, components=3),
        '      </Points>',
        '      <Cells>',
        data_array("connectivity", connectivity, vtk_type="Int64"),
        data_array("offsets", offsets, vtk_type="Int64"),
        data_array("types", cell_types, vtk_type="UInt8"),
        '      </Cells>',
        '    </Piece>',
        '  </UnstructuredGrid>',
        '</VTKFile>',
    ]
    filename.write_text("\n".join(lines) + "\n")

nx, ny = 42, 24
xs = np.linspace(0.0, 2.5, nx + 1)
ys = np.linspace(0.0, 1.0, ny + 1)
points_2d = np.array([(x, y) for y in ys for x in xs])
points = np.column_stack([points_2d, np.zeros(len(points_2d))])

def pid(i, j):
    return j * (nx + 1) + i

triangles = []
cell_centers = []
for j in range(ny):
    for i in range(nx):
        quads = [pid(i, j), pid(i + 1, j), pid(i + 1, j + 1), pid(i, j + 1)]
        xmid = xs[i : i + 2].mean()
        ymid = ys[j : j + 2].mean()
        # Leave a small circular void so that surface extraction and edges are visible.
        if (xmid - 1.2) ** 2 + (ymid - 0.5) ** 2 < 0.16 ** 2:
            continue
        triangles.extend([[quads[0], quads[1], quads[2]], [quads[0], quads[2], quads[3]]])
        cell_centers.extend([(xmid, ymid), (xmid, ymid)])

triangles = np.array(triangles, dtype=np.int64)

x = points[:, 0]
y = points[:, 1]
base_temperature = 1.0 - x / x.max() + 0.18 * np.exp(-45.0 * ((x - 1.2) ** 2 + (y - 0.5) ** 2)
material_id = np.array([1 if cx < 1.25 else 2 for cx, _ in cell_centers], dtype=np.int32)

written = []
for step, time in enumerate(np.linspace(0.0, 1.0, 6)):
    oscillation = 0.08 * np.sin(2.0 * np.pi * time) * np.sin(np.pi * x / x.max()) * np.sin(np.
    temperature = base_temperature + oscillation
    dtdx = -1.0 / x.max() + oscillation * (np.pi / x.max()) * np.cos(np.pi * x / x.max())
```

```
    dtdy = oscillation * np.pi * np.cos(np.pi * y)
    heat_flux = np.column_stack([-dtdx, -dtdy, np.zeros_like(dtdx)])
    filename = out_dir / f"heat_demo_{step:03d}.vtu"
    write_vtu(filename, points, triangles, temperature, heat_flux, material_id)
    written.append((time, filename.name))

pvd_lines = [
    '<?xml version="1.0"?>',
    '<VTKFile type="Collection" version="0.1" byte_order="LittleEndian">',
    '  <Collection>',
]
for time, filename in written:
    pvd_lines.append(f'    <DataSet timestep="{time:.6f}" group="" part="0" file="{filename}"/
pvd_lines.extend(["  </Collection>", "</VTKFile>"])
(out_dir / "heat_demo.pvd").write_text("\n".join(pvd_lines) + "\n")

print(f"Open this file in ParaView: {out_dir / 'heat_demo.pvd'}")
print(f"Wrote {len(written)} time steps with {len(points)} points and {len(triangles)} cells."
```

```
Open this file in ParaView: paraview_output/heat_demo.pvd
Wrote 6 time steps with 1075 points and 1948 cells.
```

```
from xml.etree import ElementTree as ET
import matplotlib.pyplot as plt
import matplotlib.tri as mtri

root = ET.parse("paraview_output/heat_demo_000.vtu").getroot()

def read_array(name, dtype=float):
    node = root.find(f".//DataArray[@Name='{name}']")
    return np.fromstring(node.text, sep=" ", dtype=dtype)

preview_points = read_array("Points").reshape(-1, 3)[:, :2]
preview_temperature = read_array("temperature")
preview_triangles = read_array("connectivity", dtype=np.int64).reshape(-1, 3)

triangulation = mtri.Triangulation(
    preview_points[:, 0], preview_points[:, 1], preview_triangles
)

fig, ax = plt.subplots(figsize=(9, 3.8), constrained_layout=True)
field = ax.tripcolor(triangulation, preview_temperature, shading="gouraud", cmap="coolwarm")
ax.triplot(triangulation, color="black", linewidth=0.18, alpha=0.35)
ax.set_aspect("equal")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Generated ParaView demo data: temperature on triangular mesh")
fig.colorbar(field, ax=ax, label="temperature")
plt.show()
```

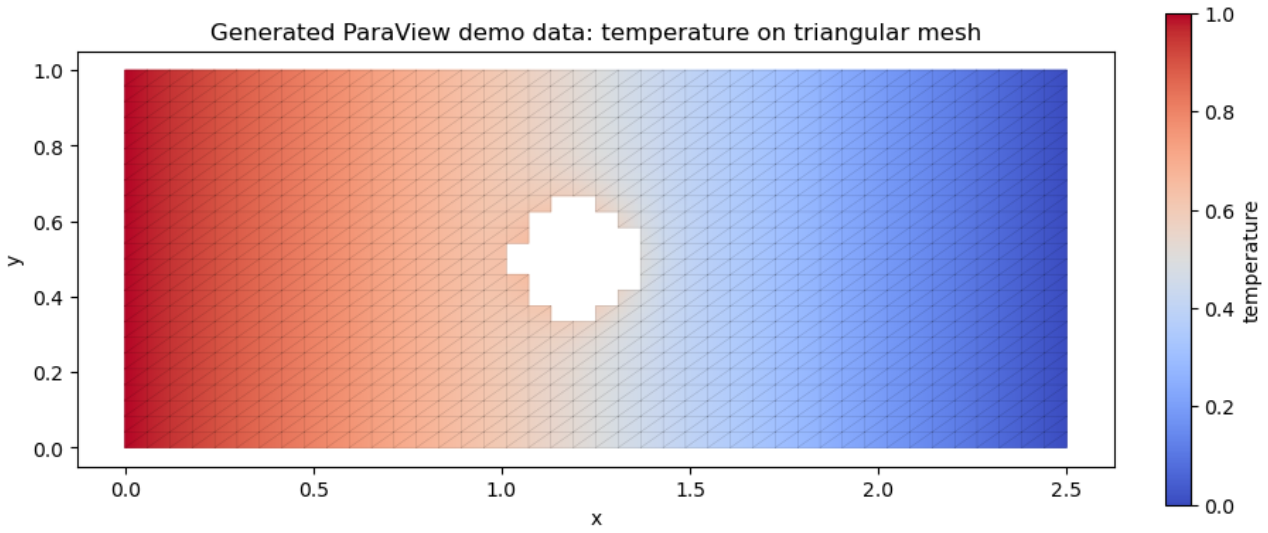


Figure 15.1.: Generated demo data for the ParaView workflow: temperature on a triangular mesh written to VTK.

15.4. ParaView's Mental Model

A ParaView session is a data-flow graph. Every object in the **Pipeline Browser** is either a data source or a filter derived from another object.

Three interface habits matter immediately:

1. Select the object you want to edit in the **Pipeline Browser**.
2. Change parameters in **Properties**.
3. Press **Apply**.

ParaView deliberately does not recompute every time you touch a property. This is useful for large data, but it also means that many beginners think nothing happened. If the view does not update, first ask: did I select the right pipeline object, and did I press **Apply**?

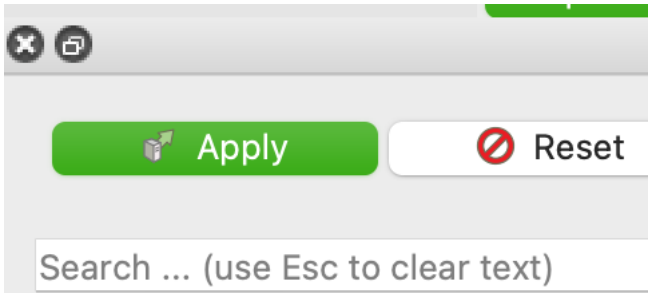
The eye icon controls visibility. It does not delete data. This is important when you compare a raw mesh, a contour result, and a warped result in the same view.

The gear icon exposes advanced properties. Use it when a filter has more options than the default panel shows.

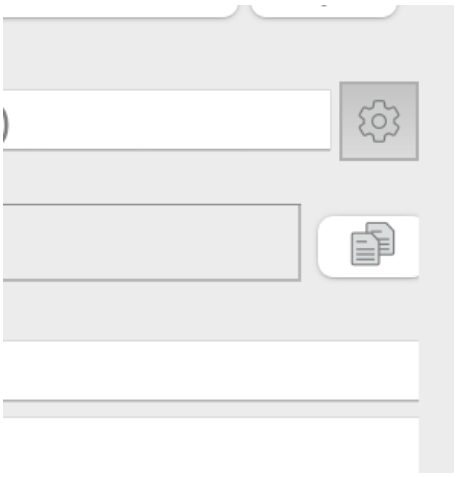
15.5. First Inspection Workflow

Open `paraview_output/heat_demo.pvd` and use this checklist:

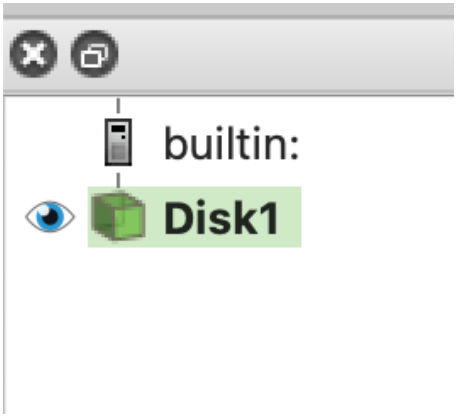
> **Lecture 03 output files**
>
> The coupled bracket example from Lecture 03 can also be opened in ParaView. Download the XDMF file together with its paired HDF5 file and keep both files in the same directory:
>
> - Temperature: `bracket_advective_heat_T.xdmf`, `bracket_advective_heat_T.h5`
> - Pressure: `bracket_advective_heat_p.xdmf`, `bracket_advective_heat_p.h5`



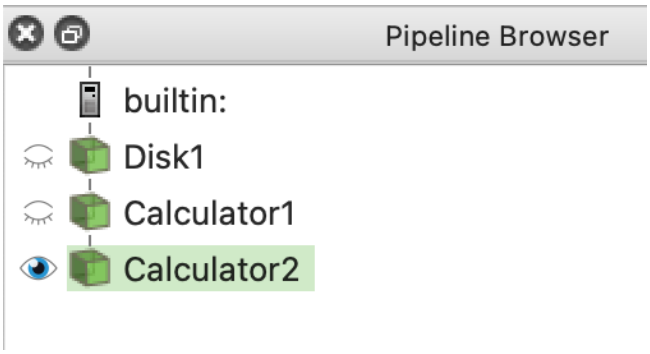
(a) Apply button in the Properties panel.



(b) Gear icon for advanced properties.



(c) Visibility eye in the Pipeline Browser.



(d) Pipeline Browser with a filter chain.

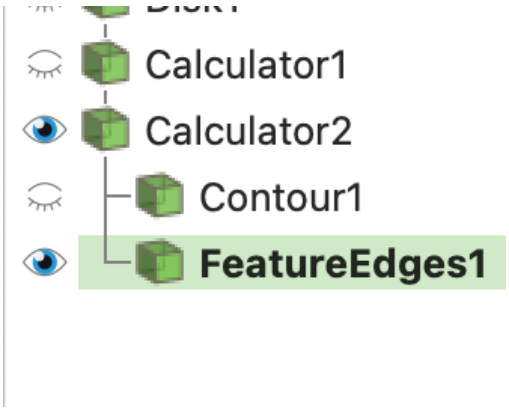
Figure 15.2.: Core ParaView interface controls used throughout the tutorial: Apply, advanced properties, object visibility, and the pipeline browser.

- Velocity: `bracket_advective_heat_u.xdmf`, `bracket_advective_heat_u.h5`

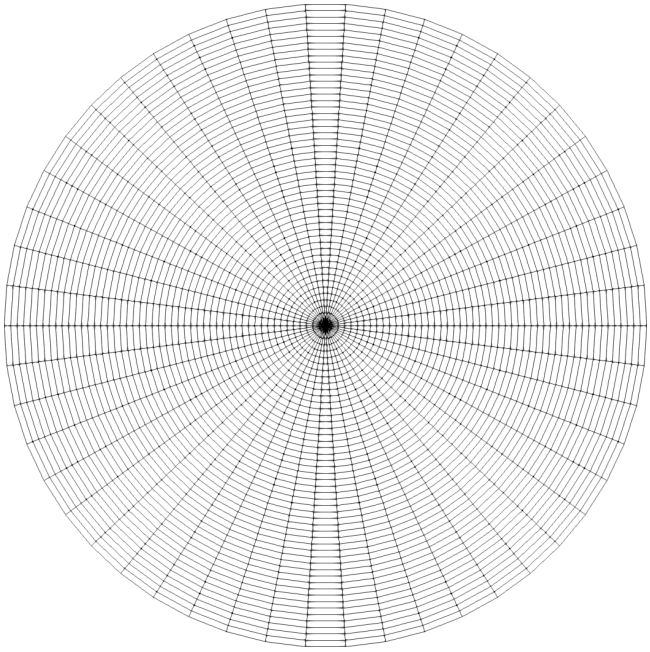
Open the `.xdmf` file in ParaView; it will read the corresponding `.h5` data file automatically.

1. Press **Apply** in the Properties panel.
2. Set representation to `Surface With Edges`.
3. Color by `temperature`.
4. Rescale the color range to the current data range.
5. Move through time with the time controls.
6. Switch coloring to `material_id` and notice that this is cell data, not point data.

A first plot should answer boring but essential questions: is the domain correct, is the void visible, are the elements plausible, does the field live on the expected part of the mesh, and does the time slider actually change the data?



(a) Representation menu with mesh-edge controls.



(b) Surface With Edges view for checking the mesh and scalar field together.

Figure 15.3.: First inspection view: enable mesh edges, color by the target field, and verify the geometry before interpreting the solution.

Tip

For debugging, `Surface With Edges` is often better than a beautiful smooth surface. It reveals element size, holes, boundaries, and interpolation artifacts.

15.6. Point Data vs Cell Data

FEM output can attach arrays to different geometric objects:

- **Point data** lives on mesh vertices and is interpolated inside cells.
- **Cell data** is one value per element.

- **Field data** is global metadata and is not directly drawn on the mesh.

This distinction changes what a plot means. Continuous P_1 temperature fields are naturally point data. Material labels, subdomain ids, and many DG quantities are naturally cell data. If ParaView offers both point and cell versions of an array, read the label carefully before interpreting discontinuities or smoothness.

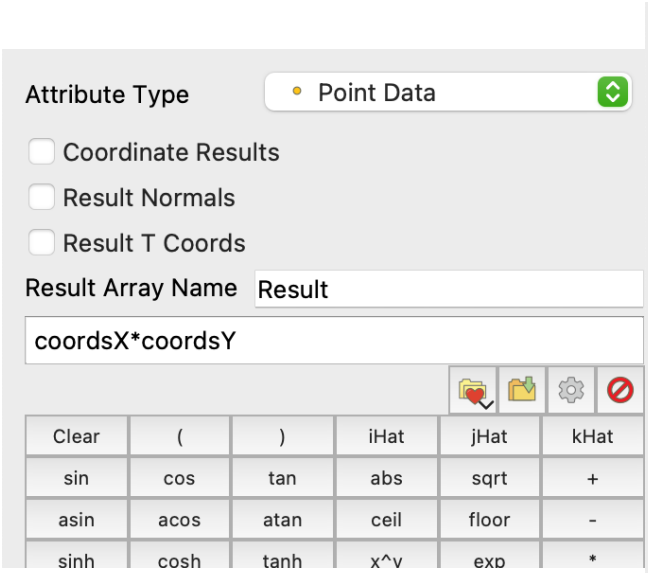
15.7. Calculator Filter

`Filters > Alphabetical > Calculator` creates a new array from existing arrays and coordinates. Typical uses are unit conversion, derived magnitudes, nondimensional quantities, and quick sanity checks.

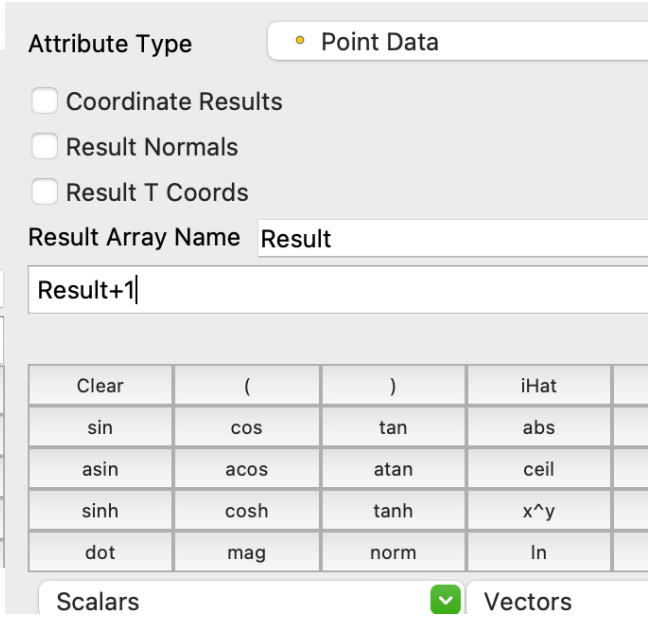
Examples for this data set:

- `temperature - 273.15` if a field was exported in Kelvin and should be inspected in Celsius.
- `mag(heat_flux)` to inspect the flux magnitude.
- `coordsX*coordsY` as a simple coordinate-dependent test field.

Use a clear **Result Array Name** such as `heat_flux_magnitude`. Then press **Apply**, color by the new array, and rescale the color range.



(a) Calculator expression and result-array name.



(b) New calculator output in the pipeline.

Figure 15.4.: Calculator workflow: define a derived array, apply the filter, and inspect the generated pipeline object.

> **Warning**
>
> Calculator expressions act on the selected data association. If the input array is point data, the result is point data. If you need cell-wise results, convert or select the correct association first.

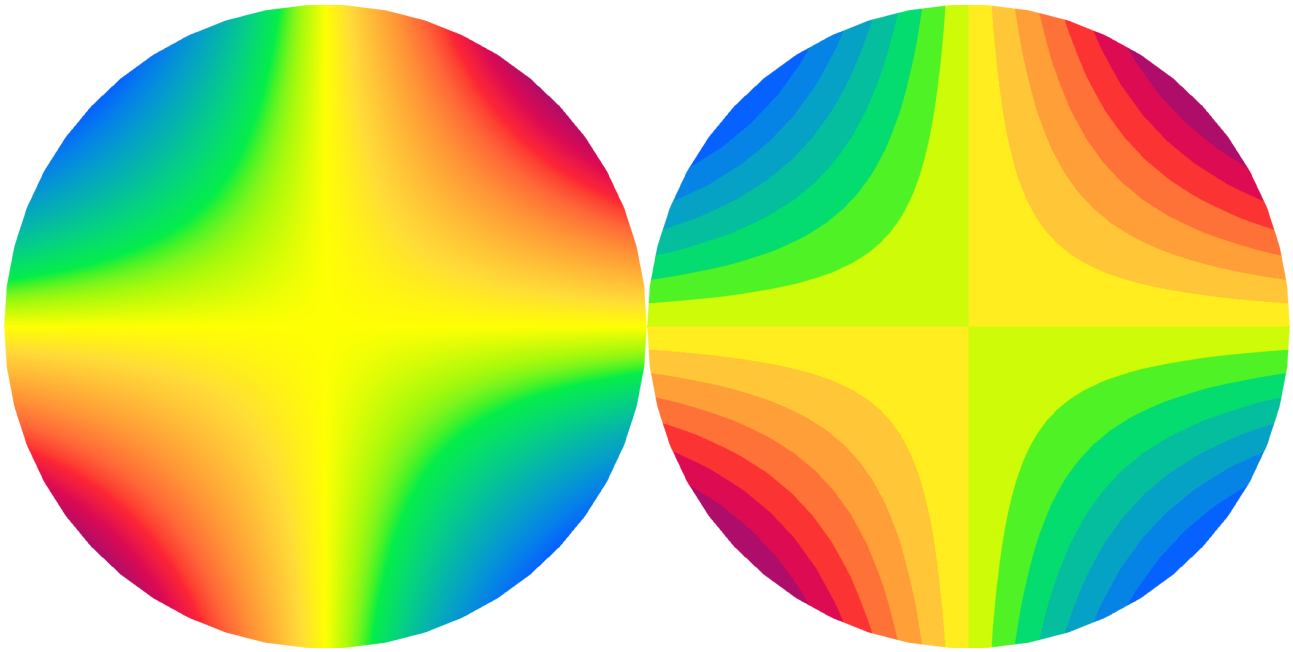
15.8. Color Ranges and Legends

Color is part of the numerical argument. The same field can look stable, unstable, smooth, or broken depending on the chosen range.

Useful buttons and settings:

1. **Rescale to Data Range** for one static snapshot.
2. **Rescale to Temporal Range** for time series comparisons.
3. **Use Discrete Colors** when levels should correspond to contour lines or material ids.
4. **Edit Color Legend Properties** to set titles, number formatting, font size, and labels.

Figure 15.5.: Color-range rescaling controls for static and temporal data ranges.



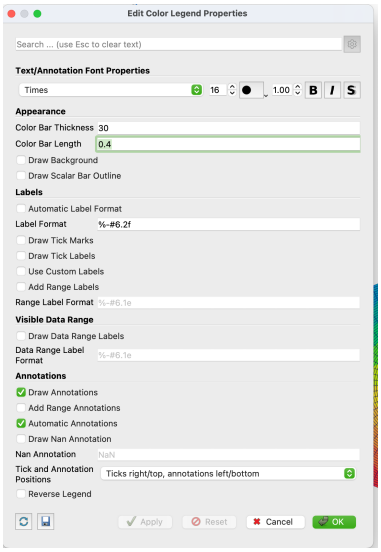
(a) Continuous color map for smooth scalar fields. (b) Discrete color map for levels or labeled regions.

Figure 15.6.: Continuous and discrete color maps communicate different numerical assumptions about the same data.

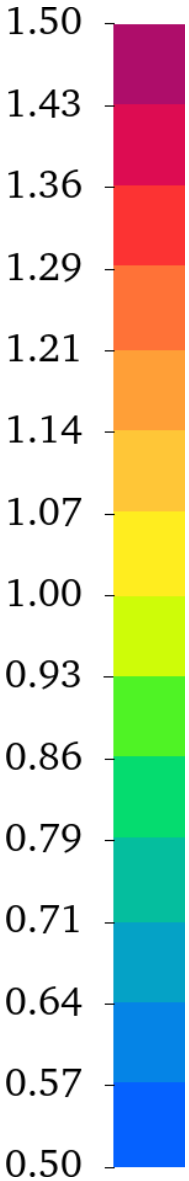
For reports, do not rely on ParaView defaults blindly. Use a readable legend title with units, set a range that is comparable across cases, and document if the range was clipped.

15.9. Contour and Isolines

`Contour` extracts curves or surfaces where a scalar field has prescribed values. In 2D heat conduction this gives isotherms.



(a) Legend settings for titles, labels, and fonts.



(b) Color legend in the render view.

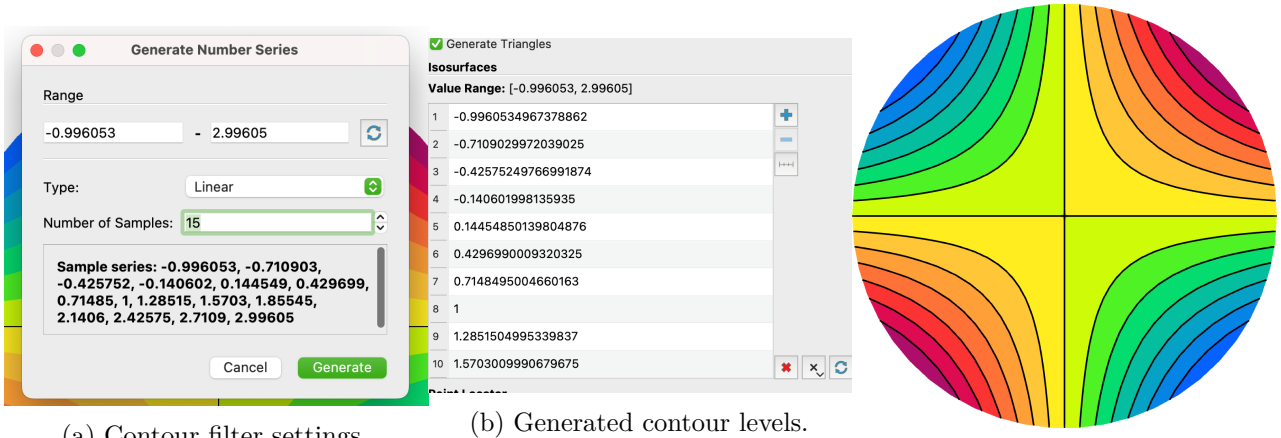
(c) Compact custom legend for report figures.

Figure 15.7.: Legend controls and output: set readable labels and units, then check how the legend appears in the final render.

Exercise:

1. Select the loaded data set.
2. Add `Contour`.
3. Choose `temperature` as the contour variable.
4. Generate around 8 to 12 values.
5. Set contour coloring to `Solid Color`, for example black.
6. Increase line width if you export a high-resolution image.

Contours are easiest to read when the underlying surface uses a continuous color map and the contour lines are a contrasting solid color.



(a) Contour filter settings.

(b) Generated contour levels.

(c) Contour lines over the scalar field.

Figure 15.8.: Contour setup and result: choose the scalar, define readable levels, and draw the isolines over the field.

15.10. Mesh Edges, Surfaces, and Boundaries

For FEM output, it is often useful to separate geometry from field values:

- `Surface With Edges` shows elements directly in the active representation.
- `Extract Surface` extracts the outer surface of a 3D data set.
- `Feature Edges` extracts sharp or boundary edges and can make hidden boundary structure visible.
- `Extract Selection` can isolate selected cells, points, or thresholded regions.

For DG methods or multi-material meshes, inspect cell boundaries before smoothing or resampling. A nice continuous-looking surface may hide an important discontinuity.

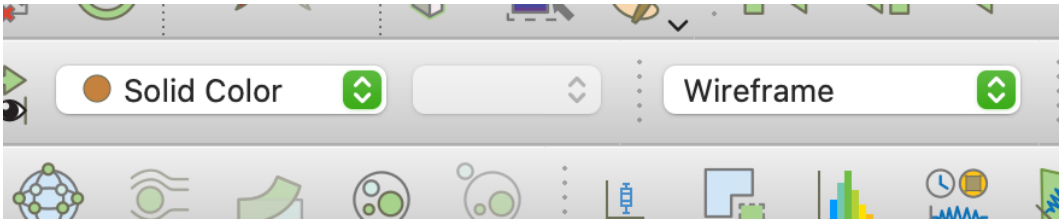


Figure 15.9.: Feature Edges filter controls for extracting boundary and sharp edges.

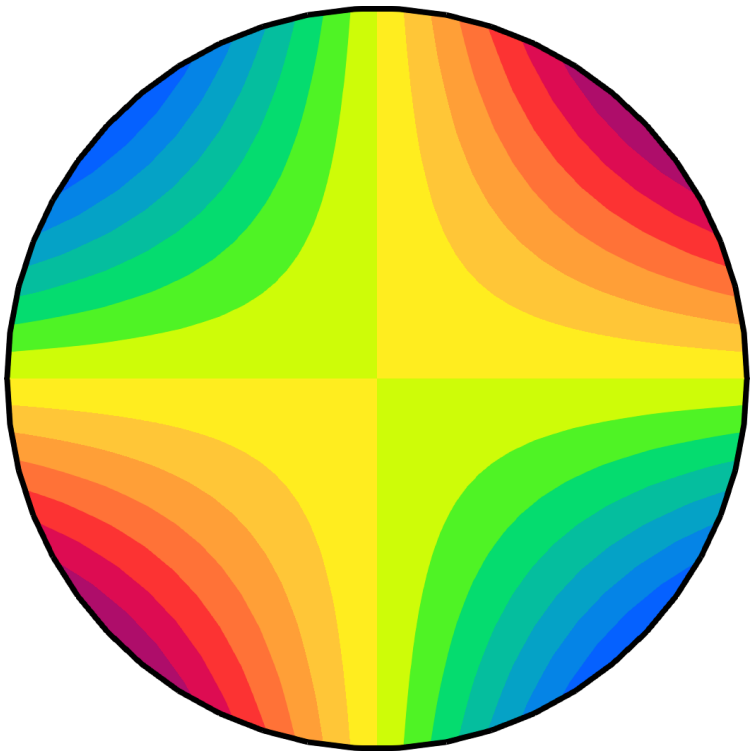


Figure 15.10.: Extracted boundary and feature edges shown without the full scalar surface.

15.11. Warp By Scalar

`Warp By Scalar` turns a scalar field into vertical displacement. For 2D scalar fields this can be a useful qualitative plot, but it changes the geometry, so it should be used deliberately.

Exercise:

1. Select the loaded data set.
2. Add `Warp By Scalar`.
3. Use `temperature` as the scalar.
4. Start with a small scale factor such as `0.25`.
5. Use the 2D/3D camera buttons to switch between flat and oblique views.

Warping is a visualization transform, not a physical deformation unless the scalar really is a displacement component.

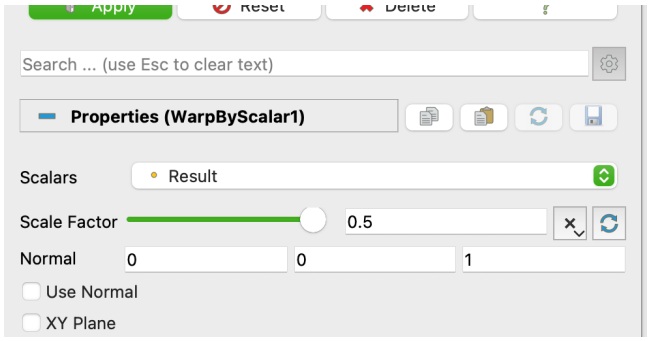
15.12. Probe and Plot Over Line

A visual plot becomes more useful when we can extract numbers from it.

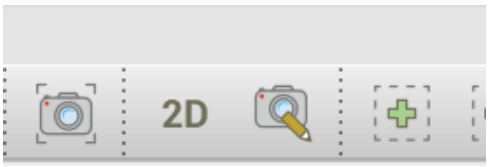
Use these tools:

- Hover over the view to inspect approximate point values.
- `Find Data` to query cells or points satisfying a condition.
- `Probe Location` to sample at one coordinate.
- `Plot Over Line` to sample a field along a segment.

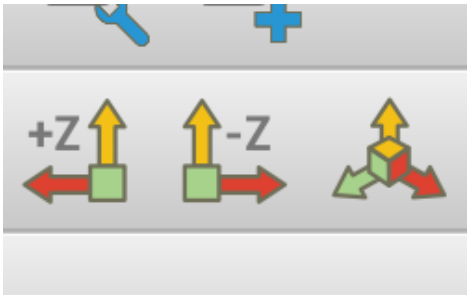
Exercise with `Plot Over Line`:



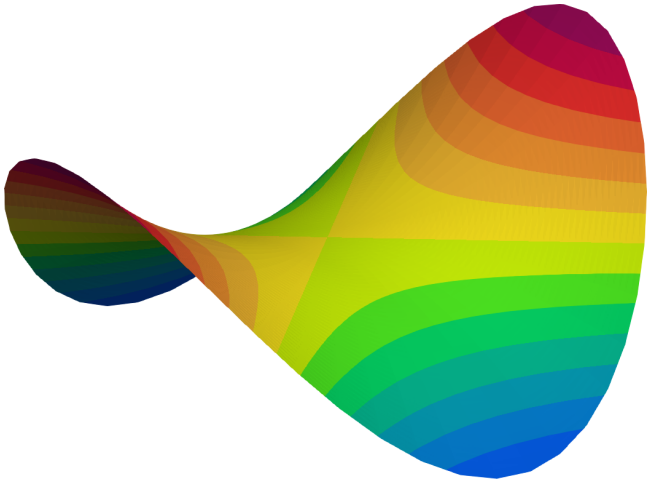
(a) Warp By Scalar filter settings.



(b) 2D and 3D camera buttons.



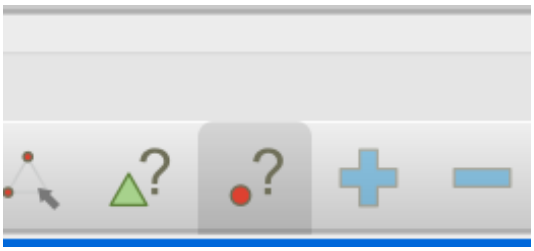
(c) Camera alignment controls.



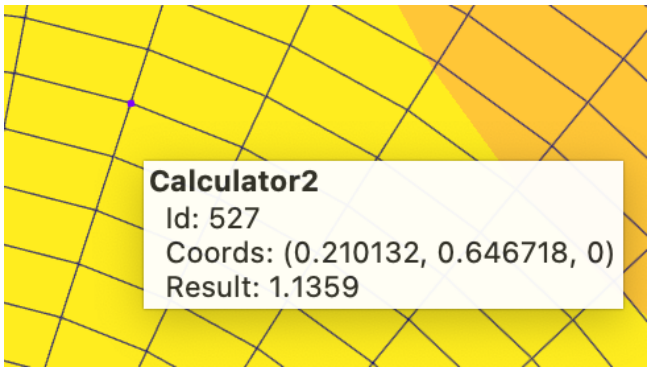
(d) Warped scalar field in an oblique view.

Figure 15.11.: Warp By Scalar workflow: set a conservative scale factor, use camera controls deliberately, and compare the warped view with the flat field.

1. Select the loaded data set.
2. Add `Plot Over Line`.
3. Set point 1 to `(0, 0.5, 0)` and point 2 to `(2.5, 0.5, 0)`.
4. Press **Apply**.
5. Export the spreadsheet view as CSV if you need the values in Python, Julia, or LaTeX.

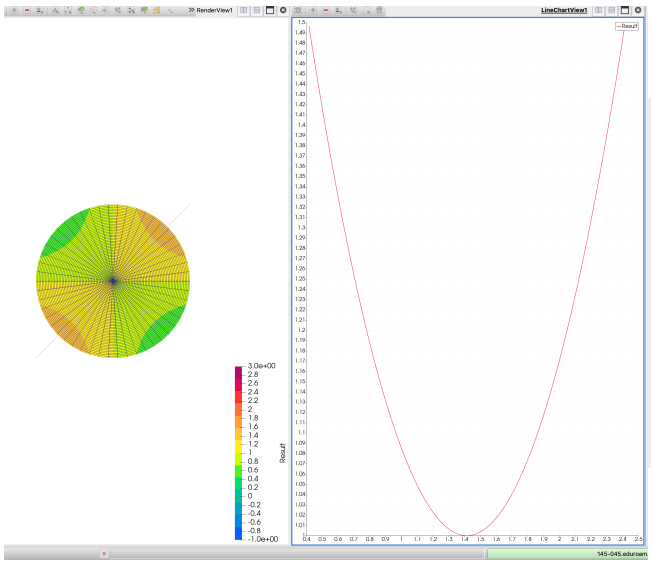


(a) Point-value inspection in the render view.

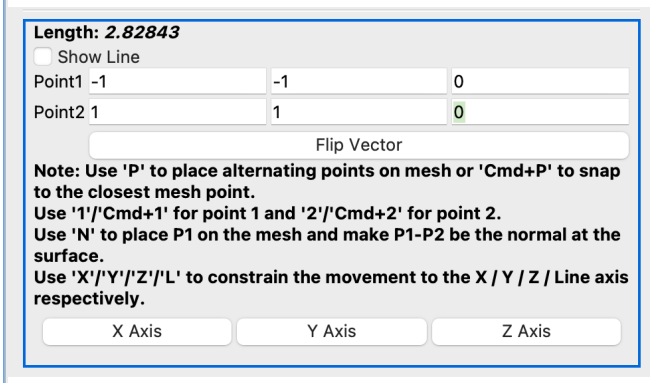


(b) Find Data dialog for querying cells and points.

Figure 15.12.: Point and cell queries turn a visual inspection into a numerical check.



(a) Plot Over Line sampling segment.



(b) Line plot output for the sampled field.

Figure 15.13.: Plot Over Line samples field values along a chosen segment and exposes the result as a plot or table.

15.13. Time Series and Animation

The `.pvd` file created above is a collection file. ParaView reads it as one time-dependent data source, even though the actual fields live in separate `.vtu` files.

For animations:

1. Open the time controls and step through the solution manually first.
2. Use a fixed color range over the full temporal range.
3. Save the state before exporting.

4. Use `File > Save Animation` only after checking the first and last frame.

A common mistake is to rescale the color map at every time step. That can hide amplitude changes completely.

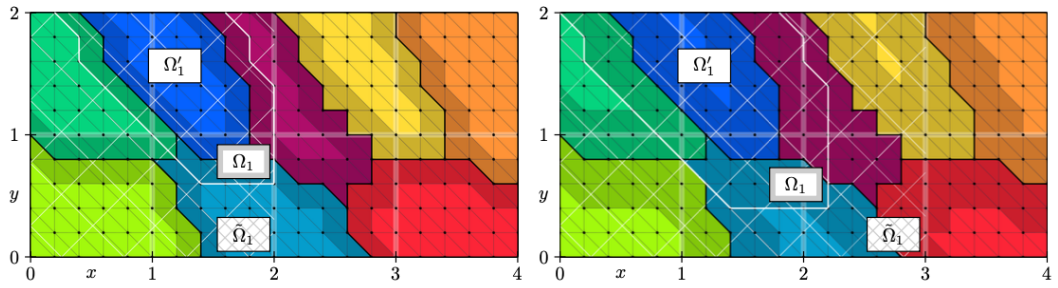
15.14. Exporting Figures and States

A reproducible ParaView figure should be recoverable from files, not only from memory.

Recommended export workflow:

1. Set background to white for print figures.
2. Hide orientation axes unless they carry information.
3. Use a fixed camera position and document it by saving state.
4. Set the image size explicitly, for example `2000 x 1400` pixels.
5. Save the state via `File > Save State` as `.pvsm`.
6. Export PNG/JPEG for quick use, and CSV for quantitative line plots.

For final papers, many groups export the main field image and the color legend separately. This gives more control over placement in LaTeX or vector graphics tools.



(a) $\delta = 1$. (b) $\delta = 2$.

FIG. 3. *Sketch of the non-overlapping* $\{\Omega'_i\}_{i=1}^8$ *(black border), overlapping* $\{\Omega_i\}_{i=1}^8$ *(white border), and periodic neighborhood decomposition* $\{\tilde{\Omega}_i\}_{i=1}^8$ *(cross-hatch) of* $\Omega_4 := (0,4) \times (0,2)$ *for an overlap (dark shades) of* **(a)** $\delta = 1$ *and* **(b)** $\delta = 2$ *layers of elements. An increase of the periodic neighborhood from* $\tilde{\Omega}_1 = (0,2) \times (0,2)$ *for* $\delta = 1$ *to* $\tilde{\Omega}_1 = (0,3) \times (0,2)$ *for* $\delta = 2$ *can be observed.*

Figure 15.14.: Advanced layer composition example with the main field and legend arranged for publication.

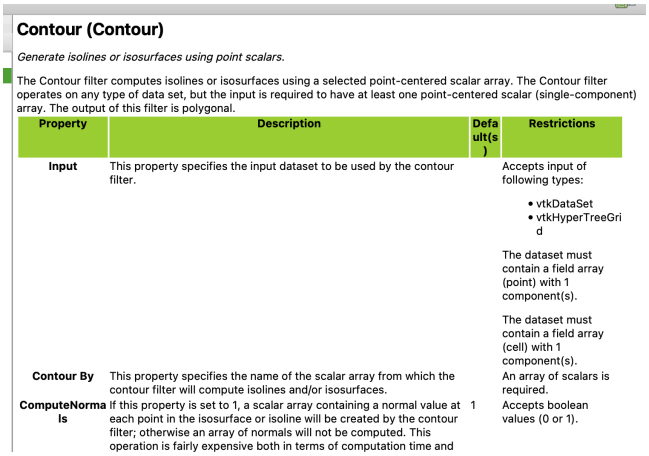
15.15. Help and Reproducibility

Every filter has built-in help through the `?` button in the Properties panel. Use it when a parameter is unclear; many filters have options that are hard to guess from the name alone.

For reproducibility, remember the following files:

- `.vtu`: one VTK unstructured-grid data set.
- `.pvd`: a collection file, often used for time series.
- `.xdmf` plus `.h5`: common FEniCSx output pair for mesh and field data.
- `.pvsm`: ParaView state file containing the visualization pipeline.
- `.py`: ParaView Python trace or script for automated rendering.

(a) Help button in a filter properties panel.



Contour (Contour)

Generate isolines or isosurfaces using point scalars.

The Contour filter computes isolines or isosurfaces using a selected point-centered scalar array. The Contour filter operates on any type of data set, but the input is required to have at least one point-centered scalar (single-component) array. The output of this filter is polygonal.

Property	Description	Default(s)	Restrictions
Input	This property specifies the input dataset to be used by the contour filter.		Accepts input of following types: • vtkDataSet • vtkHyperTreeGrid The dataset must contain a field array (point) with 1 component(s). The dataset must contain a field array (cell) with 1 component(s).
Contour By	This property specifies the name of the scalar array from which the contour filter will compute isolines and/or isosurfaces.		An array of scalars is required.
ComputeNormals	If this property is set to 1, a scalar array containing a normal value at each point in the isosurface or isoline will be created by the contour filter; otherwise an array of normals will not be computed. This operation is fairly expensive both in terms of computation time and	1	Accepts boolean values (0 or 1).

(b) Filter documentation opened inside ParaView.

Figure 15.15.: Built-in filter help is the fastest way to check unfamiliar parameters while keeping the current pipeline open.

A good submission can include raw simulation output plus a `.pvsm` state. That lets someone reopen the exact pipeline, inspect the filters, and regenerate the figure.

15.16. Typical FEniCSx Output Pattern

In FEniCSx, the most common path is to write mesh and functions through XDMF or VTK output. The exact API can vary by version, but the visualization principle is stable: write the mesh, write named fields, and keep time values explicit.

```
# Typical idea in a FEniCSx solver script, not executed here:
#
# from dolfinx import io
#
# with io.XDMFFile(mesh.comm, "solution.xdmf", "w") as xdmf:
#     xdmf.write_mesh(mesh)
#     u.name = "temperature"
#     xdmf.write_function(u, t=0.0)
#
# For a time-dependent solve, call write_function(u, t) after each accepted time step.
# Open solution.xdmf in ParaView and check that the time selector appears.
```

15.17. Further Reading

- ParaView Reference
- ParaView Ressources

[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.

[8] 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/

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

[10] 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.

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

[12] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities," *Numerical Meth Engineering*, vol. 79, no. 11, pp. 1309–1331, Sep. 2009, doi: 10.1002/nme.2579.

[13] L. Theisen and M. Torrilhon, "fenicsR13: A Tensorial Mixed Finite Element Solver for the Linear R13 Equations Using the FEniCS Computing Platform," *ACM Trans. Math. Softw.*, vol. 47, no. 2, pp. 17:1–17:29, Apr. 2021, doi: 10.1145/3442378.