ONESIMULATION

Notebook Author’s Guide

Physics models, numerical methods, and reproducible results

PYTHON API MANUAL

Source checkout based on 7bf3ce9b, inspected 10 November 2026

Companion to the OneSimulation example notebooks

How to use this guide

The workflows in Part I follow the same sequence as the example notebooks. State the physical question and units, assemble the model, choose numerical methods, solve, check the solution, and build the figure from the checked data. Start with the lumped cooling example to learn the execution path and then choose a spatial example. Part II gives a starting model, numerical route, and verification target for every physics package. Part III explains the role of each numerical layer and how to combine declarations. The appendices give the import names, signatures and contracts for the canonical exports. Chapter 6 expands the physics and service examples. Chapter 9 provides controlled method comparisons, and Chapter 13 locates each numerical symbol in the workflows or source tests.

The reference covers `onesim.physics`, its sixteen domain packages, `onesim.numerics` and its nine method packages. Publicly exposed operator namespaces are identified separately. The reference boundary is the application-facing package exports. Backend implementation modules, compatibility aliases, commercial product services and internal helper functions have separate ownership and can change with the runtime. An exported configuration, protocol or result adapter is a contract. Its presence says nothing about which physical model, mesh and solver combinations the installed runtime can execute. The examples and cited source tests specify concrete combinations.

Contents

Part I From model to notebook 8

1 Set up the notebook . . . 9

1.1 Install and select the kernel . . . 9

1.2 Separate the scientific choices . . . 10

1.3 Make the notebook tell a scientific story . . . 10

2 Transient radiation and vibration . . . 12

2.1 Radiative cooling of one thermal mass . . . 12

2.2 Reuse the transient workflow for a damped oscillator . . . 14

3 Three spatial studies . . . 16

3.1 Three-dimensional cantilever . . . 16

3.2 Heat transport with prescribed recirculation . . . 20

3.3 Magnetostatics of a homogenized solenoid . . . 23

4 Read, visualize and export results . . . 26

4.1 Choose a physical view . . . 26

4.2 Use a built-in viewer or customize the panel . . . 26

4.3 Keep verification and presentation together . . . 27

Part II Physics model selection 28

5 Build a first case in each physics package . . . 29

5.1 Reusable periodic-line setup . . . 29

5.2 Acoustics . . . 29

5.3 Biophysics . . . 30

5.4 Chemistry . . . 31

5.5 Electromagnetics . . . 32

5.6 Electrostatics . . . 33

5.7 Geotechnics . . . 35

5.8 Heat transfer . . . 35

5.9 Hydrodynamics . . . 35

5.10 Magnetohydrodynamics . . . 37

5.11 Magnetostatics . . . 38

5.12 Multibody dynamics . . . 38

5.13 Multiphysics . . . 39

5.14 Optics . . . 41

5.15 Plasma physics . . . 43
5.16 Solid mechanics . . . 44
5.17 Systems and networks . . . 44

6 Worked physics and service recipes . . . 46

6.1 Shared cell for lumped histories . . . 46
6.2 Shared cell for one-dimensional profiles . . . 47
6.3 Shared cell for facade configurations . . . 49
6.4 Acoustics recipes . . . 50
6.5 Biophysics recipes . . . 60
6.6 Chemistry recipes . . . 70
6.7 Electromagnetics recipes . . . 73
6.8 Electrostatics recipes . . . 88
6.9 Geotechnics recipes . . . 88
6.10 Heat transfer recipes . . . 99
6.11 Hydrodynamics recipes . . . 101
6.12 Magnetohydrodynamics recipes . . . 112
6.13 Magnetostatics recipes . . . 125
6.14 Multibody dynamics recipes . . . 126
6.15 Multiphysics recipes . . . 136
6.16 Optics and photonics recipes . . . 149
6.17 Plasma physics recipes . . . 153
6.18 Solid mechanics recipes . . . 170
6.19 Systems and networks recipes . . . 174

Part III Select and combine numerical methods 188

7 Numerical configuration and spatial operators . . . 189

7.1 Discretization and validation . . . 189
7.2 Finite differences . . . 189
7.3 Finite volumes and finite integration . . . 190
7.4 Finite elements . . . 190
7.5 Change one numerical choice at a time . . . 191

8 Solvers, time integration and coupled fields . . . 194

8.1 Linear solvers and preconditioners . . . 194
8.2 Nonlinear solvers . . . 194
8.3 Time integration, events and continuation . . . 195
8.4 Transfer between fields . . . 196
8.5 Coupled systems . . . 198

9 Worked numerical comparisons . . . 199

9.1 Discretization and initialization lab . . . 199

9.2 Finite-difference lab . . . 201

9.3 Finite-volume lab . . . 202

9.4 Finite-element lab . . . 205

9.5 Linear algebra lab . . . 209

9.6 Nonlinear solver lab . . . 211

9.7 Time integration and continuation lab . . . 213

9.8 Field transfer lab . . . 217

9.9 Coupled execution lab . . . 226

10 Executable numerical recipes . . . 229

10.1 Declarations and execution contracts . . . 229

10.2 Coupled execution and iterative algebra . . . 233

11 Define and solve your own equation . . . 275

11.1 Start with a well-posed mathematical problem . . . 275

11.2 Derive a verification target and its forcing . . . 276

11.3 Declare the field, coefficients and boundary values . . . 276

11.4 Choose the discretization, solve and verify . . . 277

11.5 Read the measured refinement results . . . 278

11.6 Plot the checked solution and its error . . . 280

11.7 Adapt the example within its execution contract . . . 281

12 Diagnose a notebook result . . . 282

12.1 Repository examples . . . 282

13 Coverage and navigation by symbol . . . 283

13.1 Physics model and service coverage . . . 283

13.2 discretization selection index . . . 285

13.3 fdm selection index . . . 285

13.4 fvm selection index . . . 286

13.5 fem selection index . . . 288

13.6 linear selection index . . . 290

13.7 nonlinear selection index . . . 290

13.8 time selection index . . . 291

13.9 field_transfer selection index . . . 291

13.10 coupled_systems selection index . . . 292

A Physics API reference 293

A.1 onesim.physics . . . 293

A.2 onesim.physics.acoustics . . . 302
A.3 onesim.physics.biophysics . . . 309
A.4 onesim.physics.chemistry . . . 314
A.5 onesim.physics.electromagnetics . . . 317
A.6 onesim.physics.electrostatics . . . 319
A.7 onesim.physics.geotechnics . . . 319
A.8 onesim.physics.heat_transfer . . . 328
A.9 onesim.physics.hydrodynamics . . . 331
A.10 onesim.physics.magnetohydrodynamics . . . 337
A.11 onesim.physics.magnetostatics . . . 342
A.12 onesim.physics.multibody . . . 343
A.13 onesim.physics.multiphysics . . . 352
A.14 onesim.physics.optics . . . 362
A.15 onesim.physics.plasma . . . 365
A.16 onesim.physics.solid_mechanics . . . 370
A.17 onesim.physics.systems . . . 373

B Numerical API reference 387

B.1 onesim.numerics . . . 387
B.2 onesim.numerics.coupled_systems . . . 388
B.3 onesim.numerics.discretization . . . 390
B.4 onesim.numerics.fdm . . . 393
B.5 onesim.numerics.fem . . . 396
B.6 onesim.numerics.field_transfer . . . 403
B.7 onesim.numerics.fvm . . . 407
B.8 onesim.numerics.linear . . . 416

B.9 onesim.numerics.nonlinear 418

B.10 onesim.numerics.time 420

C Exposed operator and service namespaces 423

C.1 onesim.numerics.field_transfer.operators 423

C.2 onesim.physics.electromagnetics.operators 425

C.3 onesim.physics.plasma.operators 427

C.4 onesim.physics.heat_transfer.advanced_thermal 428

C.5 onesim.physics.heat_transfer.configuration 429

C.6 onesim.physics.heat_transfer.errors 430

C.7 onesim.physics.heat_transfer.heat_conduction 430

C.8 onesim.physics.heat_transfer.heat_transport 431

C.9 onesim.physics.heat_transfer.public_api 432

C.10 onesim.physics.heat_transfer.validation_report 433

PART I

From model to notebook

CHAPTER 1

Set up the notebook

1.1 Install and select the kernel

Use the interpreter and wheel specified by the release you are running. The first example notebooks were written for Runtime Alpha 0.2.1a2. The source checkout declares Python 3.13 or later in `implementation/pyproject.toml` and has its own package metadata. The distribution is `onesimulation`; Python imports use `onesim`. In a development checkout, use the repository root as the working directory for these shell commands.

```
python3.13 -m venv .venv
source .venv/bin/activate
python -m pip install -e './implementation[notebook]'
python -m ipykernel install --user --name onesim
python -m jupyter lab
```

For a delivered wheel, install the platform-specific wheel with the extras described in its accompanying `README_DE.md`. Select that environment as the notebook kernel. The notebook extra includes JupyterLab, plotting libraries, widgets and the tetrahedral mesher for the source installation. Record the actual interpreter and distribution version near the top of your notebook.

```
import sys
from importlib.metadata import version

print(sys.executable)
print("OneSimulation", version("onesimulation"))
```

The plot examples load the shared Matplotlib theme from `onesimulation_plot_theme.py`. Keep this file beside your notebook and call `theme.use_theme()` before creating figures.

1.2 Separate the scientific choices

Object	Choose or declare	Inspect
Geometry and selections	Dimensions, regions and named boundaries.	World coordinates, boundary ownership and mesh markers.
`Problem`	Models, materials, loads, initial conditions, constraints and field links.	Units, unknowns, source signs and conditions for uniqueness.
Mesh and `Numerics`	Spatial carrier, schemes, integration, solvers and tolerances.	Actual mesh size, admissible options, convergence and time-step limits.
`Simulation`	Bind the problem, mesh and numerics. Choose a stationary solve or a transient run.	Validation issues, result fields, solver report and accepted output times.
Figure and data	Samples, components, units, color scale, view and exports.	Axes, physical support, reference errors and reproducibility.

Use `LumpedDomain().system` for global balances, `Line(...).line` for the supplied one-dimensional models and `Box(...).volume` or `Cylinder(...).volume` for volume models. A line is embedded in world space. Its local arclength can differ from the world x coordinate. To combine boundary selections, use union `|`, intersection `&` and difference `-` on selections from the same geometry. Generate the mesh from the geometry that owns those selections. Recreate the model and its conditions together when making a parameter variant that changes geometry or physics ownership.

A model provides typed references such as `model.temperature`. Use `model.fields()` to inspect the declared fields and their `field_id`, `unit` and `role`. Differential fields require a consistent initial state for transient runs. Algebraic fields satisfy a constraint or a stationary equation. Derived fields are computed from the solution. Coupling targets are model input references. Use `InitialValue(ref, value)` to set an initial condition. For stationary nonlinear iteration, `Numerics(initial_guess=...)` serves a separate purpose. Check the contract of each callable source or condition. The volume examples below use vectorized `f(x, y, z)`, while some boundary conditions use `f(time, coordinate)`. A numerical array must match the carrier shape and component order expected by the receiving field.

1.3 Make the notebook tell a scientific story

Open with one question, a diagram or description of the geometry, assumptions, and the measured output. Use a short, editable parameter cell with units. Put the complete solve in a function when you will compare meshes or parameter settings. Keep the extraction cell separate so its outputs can drive a plot, a CSV table and a verification summary. Restart the kernel and run all cells in order before sharing the notebook.

Start the verification cell with a check that would fail if you plotted the wrong result. Examples include the sign of the deflection, the units and shape of a field, the final accepted output time or the total integrated source. Then compare with an independent solution, a balance, a symmetry, or a refinement run. State the expected approximation error and use a tolerance scaled to the observable. A solver residual measures how well the algebraic equations were solved. A refinement comparison measures sensitivity to the chosen mesh or step size.

The recirculation notebook exposes a fast profile and a finer profile. The cantilever compares two meshes in one run. Record the profile, mesh cells, element order, step size, solver and tolerances. Save native values separately from display conversions. A table headed `stress_Pa` stays in pascals even if the plot colorbar displays MPa. Use the same color limits when comparing two plots of one quantity. Center a difference plot's diverging scale at zero. For field slices, label the physical plane, maintain equal spatial aspect, and identify any visual deformation scale. A denser display grid samples the same solution. A finer solver mesh requires a second solve.

CHAPTER 2

Transient radiation and vibration

2.1 Radiative cooling of one thermal mass

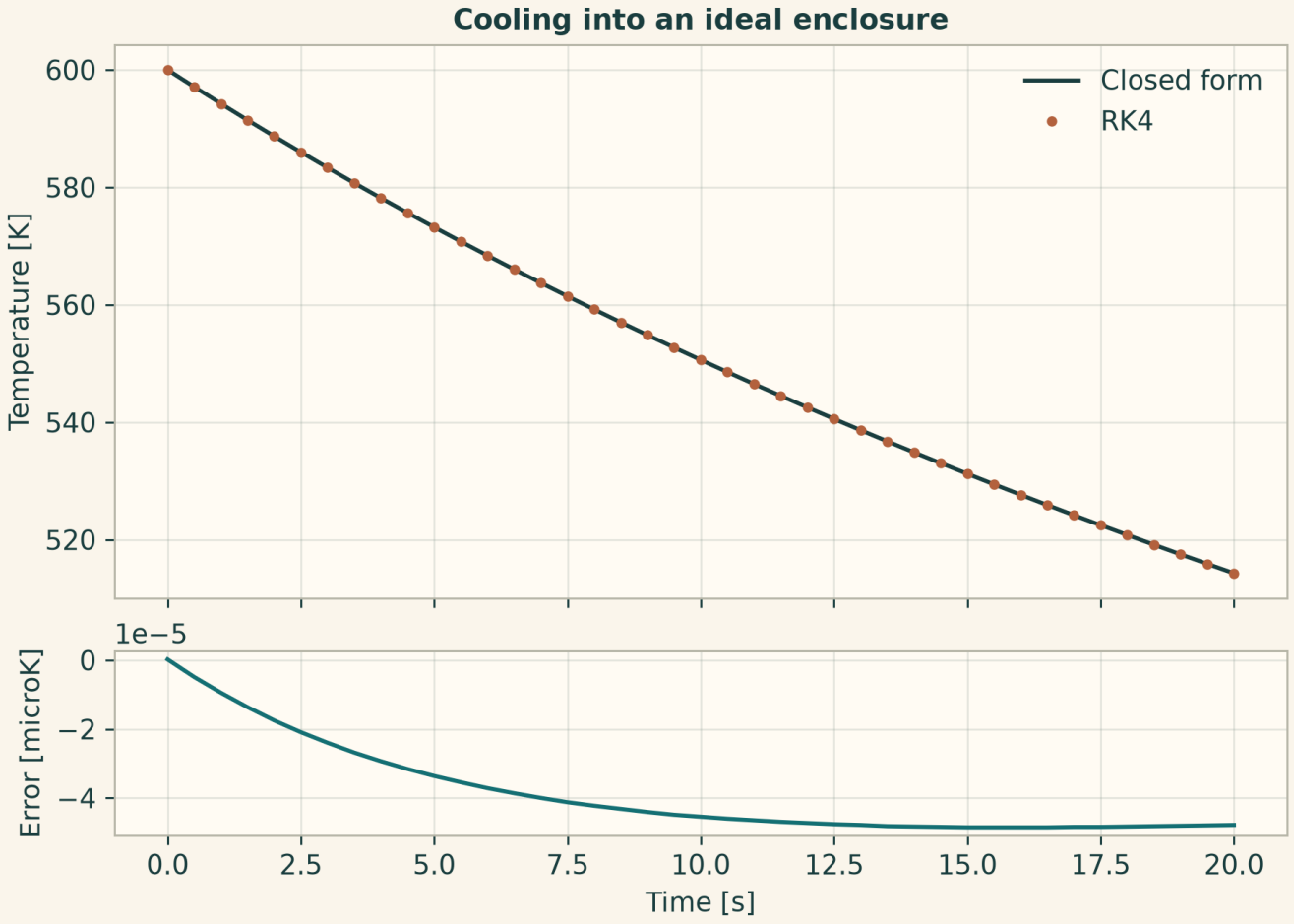
The physical state is a single temperature in kelvin. The surface radiates $\varepsilon\sigma A(T^4 - T_\infty^4)$ watts into an enclosure. The thermal capacity C is the *total* heat capacity in J/K. With zero heating power and $T_\infty = 0$, separation of variables gives

$$T(t) = \left(T_0^{-3} + 3\varepsilon\sigma A t / C\right)^{-1/3}.$$

The ambient value of zero kelvin is an idealized reference case. For a physical enclosure supply its absolute temperature and choose a reference appropriate to that model.

```
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.heat_transfer import (
    RadiationCooling, ThermalGreyDiffuseRadiationBoundary,
)
from onesim.results.storage import MemoryOutputSink, Output
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

capacity, area, emissivity = 100.0, 0.1, 0.8  # J/K, m^2, 1
T0, ambient, end, dt = 600.0, 0.0, 20.0, 0.25  # K, K, s, s

def run_lumped(domain, model, initial, fields, *, end, dt, interval):
    problem = Problem(
        name=type(model).__name__, geometry=domain, physics=(model,),
        initial_conditions=tuple(InitialValue(ref, v) for ref, v in initial),
    )
    sim = Simulation(
        problem=problem, mesh=LumpedMesh(domain),
        numerics=Numerics(
            spatial=LumpedParameters(), time_integrator=ClassicalRungeKutta4(),
        ),
    )
    sim.validate().require_valid()
    sink = MemoryOutputSink()
    result = sim.run(
        start_time=0.0, end_time=end, time_step=dt,
```

```
        output=Output(fields=fields, interval=interval, sink=sink),
    )
    times = np.array([float(s.time) for s in sink.snapshots])
    assert times.size > 1 and np.isclose(times[-1], end)
    return result, sink, times

def scalar_series(sink, ref):
    return np.array([
        np.asarray(s.fields[ref.field_id].values).item() for s in sink.snapshots
    ])

domain = LumpedDomain()
model = RadiationCooling(
    domain=domain.system, heat_capacity=capacity, radiating_area=area,
    radiation=ThermalGreyDiffuseRadiationBoundary(
        condition_id="cooling", emissivity=emissivity,
        ambient_temperature=ambient,
    ),
)
result, sink, t = run_lumped(
    domain, model, ((model.temperature, T0),),
    (model.temperature, model.radiated_power), end=end, dt=dt, interval=0.5,
)
T = scalar_series(sink, model.temperature)
P = scalar_series(sink, model.radiated_power)
sigma = 5.670374419e-8  # W/(m^2 K^4)
exact = (T0**(-3) + 3 * emissivity * sigma * area * t / capacity)**(-1 / 3)
np.testing.assert_allclose(T, exact, rtol=0, atol=1e-6)
assert np.all(np.diff(T) < 0) and np.all(P >= 0)

out = Path("onesim-output")
out.mkdir(exist_ok=True)
np.savetxt(
    out / "cooling.csv", np.column_stack((t, T, exact, P)), delimiter=",",
    header="time_s,temperature_K,reference_K,power_W", comments="",
)
theme.use_theme()
fig, (ax, err) = plt.subplots(
    2, 1, sharex=True, figsize=(7, 5), layout="constrained",
    gridspec_kw={"height_ratios": [3, 1]},
)
ax.plot(t, exact, color=theme.INK, label="Closed form")
ax.plot(t, T, "o", ms=3, color=theme.COPPER, label="RK4")
ax.set(ylabel="Temperature [K]", title="Cooling into an ideal enclosure")
ax.legend(frameon=False)
err.plot(t, 1e6 * (T - exact), color=theme.TEAL)
err.set(xlabel="Time [s]", ylabel="Error [microK]")
for panel in (ax, err):
    panel.grid(alpha=0.22)
fig.savefig(out / "cooling.svg")
plt.show()
```



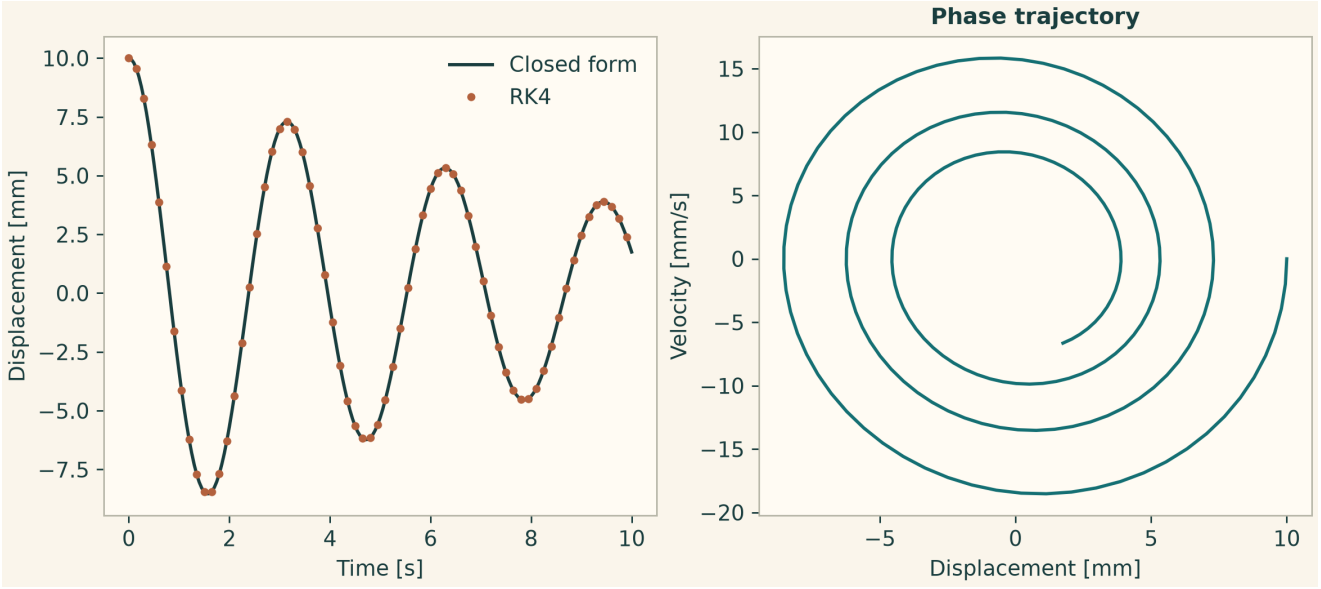
`time_step` controls integration. `Output(interval=...)` controls which times are written, and the actual times live on the snapshots. Here the interval is an integer multiple of the step and the code checks the end time. `scalar_series` deliberately requires exactly one value per field and snapshot. Spatial and vector results need a different extractor. The error panel exposes discrepancies that are invisible in the overlaid temperature curves.

2.2 Reuse the transient workflow for a damped oscillator

Keep the imports and helpers from the preceding example. An oscillator adds an independent velocity initial condition. With mass m, stiffness k and damping c, the unforced motion satisfies $m\ddot{x} + c\dot{x} + kx = 0$. In the underdamped regime, $\alpha = c/(2m)$ and $\omega = \sqrt{k/m - \alpha^2}$. Check displacement and velocity independently. Mechanical energy gives an additional check even when the phase of the motion is hard to see in a plot.

```
from onesim.physics.solid_mechanics import StructuralOscillator

mass, stiffness, damping = 1.0, 4.0, 0.2  # kg, N/m, N s/m
x0, v0 = 0.01, 0.0  # m, m/s
domain = LumpedDomain()
osc = StructuralOscillator(
    domain=domain.system, mass=mass, stiffness=stiffness, damping=damping,
)
result, sink, t = run_lumped(
    domain, osc, ((osc.displacement, x0), (osc.velocity, v0)),
    (osc.displacement, osc.velocity, osc.mechanical_energy),
```

```
    end=10.0, dt=0.01, interval=0.05,
)
x = scalar_series(sink, osc.displacement)
v = scalar_series(sink, osc.velocity)
energy = scalar_series(sink, osc.mechanical_energy)
alpha = damping / (2 * mass)
omega = np.sqrt(stiffness / mass - alpha**2)
b = (v0 + alpha * x0) / omega
shape = x0 * np.cos(omega * t) + b * np.sin(omega * t)
exact_x = np.exp(-alpha * t) * shape
exact_v = np.exp(-alpha * t) * (
    -alpha * shape - x0 * omega * np.sin(omega * t)
    + b * omega * np.cos(omega * t)
)
np.testing.assert_allclose(x, exact_x, rtol=0, atol=2e-8)
np.testing.assert_allclose(v, exact_v, rtol=0, atol=5e-8)
np.testing.assert_allclose(energy, (mass * v**2 + stiffness * x**2) / 2)
assert np.max(np.diff(energy)) <= 1e-12
assert energy[-1] < energy[0]

fig, (ax, phase) = plt.subplots(1, 2, figsize=(9, 4), layout="constrained")
ax.plot(t, 1e3 * exact_x, color=theme.INK, label="Closed form")
ax.plot(t[::3], 1e3 * x[::3], "o", ms=3, color=theme.COPPER, label="RK4")
ax.set(xlabel="Time [s]", ylabel="Displacement [mm]")
ax.legend(frameon=False)
phase.plot(1e3 * x, 1e3 * v, color=theme.TEAL)
phase.set(xlabel="Displacement [mm]", ylabel="Velocity [mm/s]",
          title="Phase trajectory")
fig.savefig(out / "oscillator.svg")
plt.show()
```



Change the damping and plot a second run with the same axes. When $k/m - \alpha^2$ is zero or negative, replace the reference formula with the critically damped or overdamped solution. The numerical model still receives the physical parameters in SI units.

CHAPTER 3

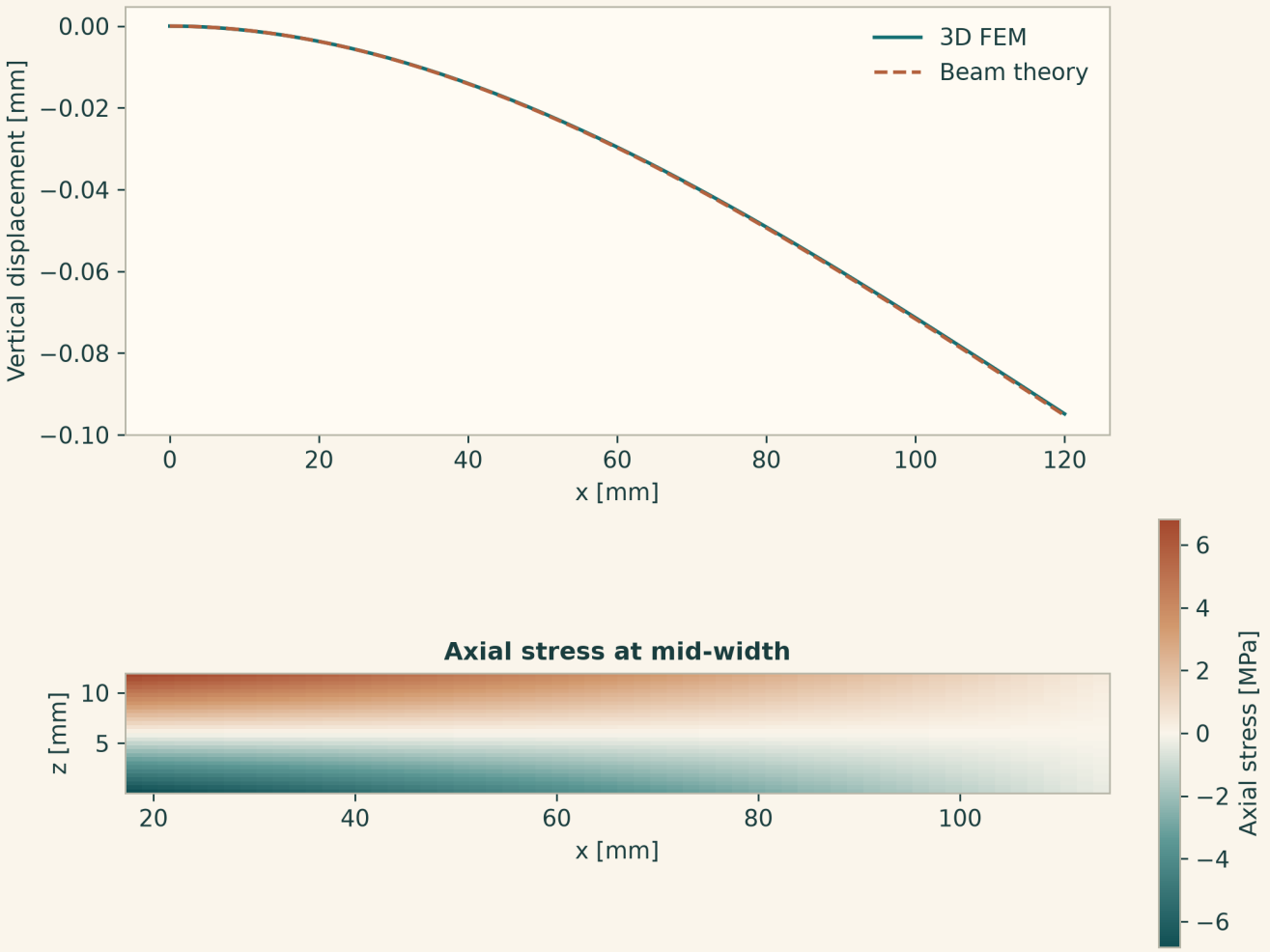
Three spatial studies

3.1 Three-dimensional cantilever

A rectangular solid is fixed at $x = 0$ and loaded on the opposite face. The force is distributed as a surface traction in Pa, with $t_z = -F/(wh)$. Q2 elements use a second-degree tensor-product displacement basis on the structured hexahedra. A nodal field carries three displacement components. Stress is a derived field with its own evaluation rule. Sample the centerline and a section inside the body in world coordinates.

```
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

from onesim.cad.d3 import Box
from onesim.materials import IsotropicSolid
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.linear import SparseDirect
from onesim.physics.solid_mechanics import (
    FixedDisplacement, LinearElasticity, Traction, TractionFree,
)
from onesim.simulation import Problem, Simulation

L, W, H = 0.120, 0.018, 0.012  # m
E, nu, force = 70e9, 0.30, 30.0  # Pa, 1, N
material = IsotropicSolid(
    density=2700.0, specific_heat_capacity=900.0, thermal_conductivity=200.0,
    youngs_modulus=E, poisson_ratio=nu, thermal_expansion_coefficient=23e-6,
)

def solve_beam(cells):
    box = Box(length=L, width=W, height=H)
    model = LinearElasticity(domain=box.volume, material=material)
    b = box.boundaries
    problem = Problem(
        name="Cantilever", geometry=box, physics=(model,),
        boundary_conditions=(
            FixedDisplacement(boundary=b.x_min, physics=model, value=(0, 0, 0)),
            Traction(boundary=b.x_max, physics=model, value=(0, 0, -force/(W*H))),
            TractionFree(boundary=b.all - b.x_min - b.x_max, physics=model),
        ),
    )
    mesh = StructuredHexahedralMesher(cells=cells).generate(box)
    sim = Simulation(
        problem=problem, mesh=mesh,
```

```
        numerics=Numerics(
            spatial=FiniteElement(element=Lagrange(order=2)),
            linear_solver=SparseDirect(),
        ),
    )
    sim.validate().require_valid()
    return model, sim.solve_stationary()

x = np.linspace(0, L, 81)
points = np.column_stack((x, np.full_like(x, W/2), np.full_like(x, H/2)))
coarse_model, coarse = solve_beam((10, 3, 2))
model, result = solve_beam((20, 4, 3))
displacement = result.field(model.displacement)
u = displacement.evaluate(points)
coarse_tip = coarse.field(coarse_model.displacement).evaluate(points[-1:])[0, 2]
inertia = W * H**3 / 12
reference = -force * x**2 * (3*L - x) / (6*E*inertia)
assert np.isfinite(u).all() and u[-1, 2] < 0
assert np.max(np.abs(u[0])) < 1e-12
assert abs(u[-1, 2] / reference[-1] - 1) < 0.02
assert abs(coarse_tip / u[-1, 2] - 1) < 0.01

xs = np.linspace(0.15*L, 0.95*L, 60)
zs = np.linspace(0.02*H, 0.98*H, 30)
xx, zz = np.meshgrid(xs, zs)
section = np.column_stack((xx.ravel(), np.full(xx.size, W/2), zz.ravel()))
stress = result.field(model.stress).evaluate(section)
sxx = stress[:, 0].reshape(xx.shape)
assert np.isfinite(sxx).all()
limit = np.max(np.abs(sxx)) / 1e6

theme.use_theme()
fig, (ax, cut) = plt.subplots(2, 1, figsize=(8, 6), layout="constrained")
ax.plot(1e3*x, 1e3*u[:, 2], color=theme.TEAL, label="3D FEM")
ax.plot(1e3*x, 1e3*reference, "--", color=theme.COPPER, label="Beam theory")
ax.set(xlabel="x [mm]", ylabel="Vertical displacement [mm]")
ax.legend(frameon=False)
im = cut.pcolormesh(1e3*xx, 1e3*zz, sxx/1e6, shading="auto",
                    cmap=theme.DIVERGING, vmin=-limit, vmax=limit)
cut.set(xlabel="x [mm]", ylabel="z [mm]", title="Axial stress at mid-width")
cut.set_aspect("equal")
fig.colorbar(im, ax=cut, label="Axial stress [MPa]")
out = Path("onesim-output")
out.mkdir(exist_ok=True)
fig.savefig(out / "cantilever.svg")
plt.show()
```



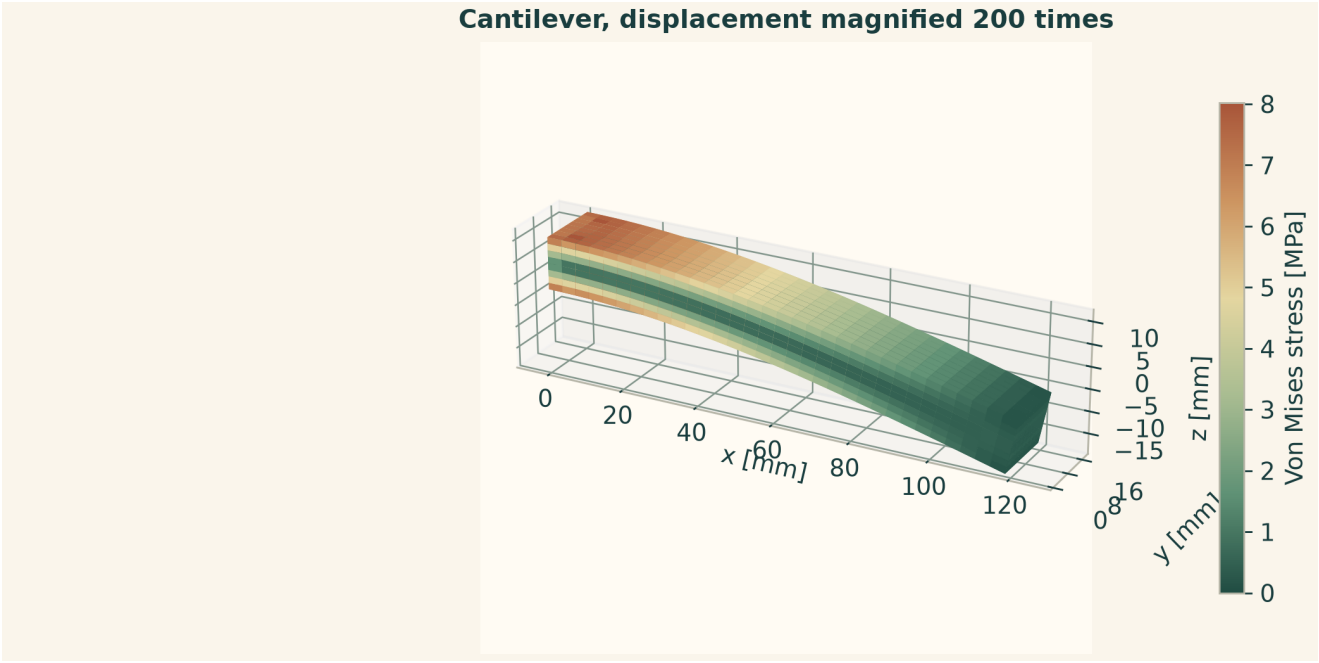
The Euler–Bernoulli curve is an independent approximation for the centerline of a slender beam. Three-dimensional effects near the clamp and load face are expected. In the example notebook the load is also integrated from the assembled nodal forces, the centerline symmetry is checked, and surface stress is evaluated before drawing the deformed geometry. To reproduce its surface view, tessellate each face for display, sample stress at patch centers and displacement at patch corners, then color the displaced patches using the sampled stress. State the displacement magnification in the figure.

3.1.1 Draw the deformed surface

Run this cell directly after the cantilever example. Sample displacement at the corners of each display patch and von Mises stress at its center. Deform the corner positions with a stated visual multiplier and color them using a separate stress scale. The sampling positions are in the original configuration. The colorbar retains the original stress values after conversion to MPa.

```
from matplotlib.colors import Normalize
from matplotlib.ticker import MaxNLocator
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

sizes = (L, W, H)
resolution = (32, 8, 8)
quads = []
for normal in range(3):
    a, b = [axis for axis in range(3) if axis != normal]
```

```
    aa = np.linspace(0, sizes[a], resolution[a] + 1)
    bb = np.linspace(0, sizes[b], resolution[b] + 1)
    for side in (0.0, sizes[normal]):
        for a0, a1 in zip(aa[:-1], aa[1:]):
            for b0, b1 in zip(bb[:-1], bb[1:]):
                q = np.empty((4, 3))
                q[:, normal] = side
                q[:, a] = (a0, a1, a1, a0)
                q[:, b] = (b0, b0, b1, b1)
                quads.append(q)
quads = np.asarray(quads)
shape = quads.shape
surface_u = displacement.evaluate(quads.reshape(-1, 3)).reshape(shape)
mises = result.field(model.von_mises_stress).evaluate(quads.mean(axis=1))
assert np.isfinite(surface_u).all() and np.isfinite(mises).all()
assert np.min(mises) >= 0 and np.max(mises) > 0

magnification = 200.0
display_quads = 1e3 * (quads + magnification * surface_u)
fig, ax = plt.subplots(
    figsize=(8, 4), subplot_kw={"projection": "3d"}, layout="constrained",
)
norm = Normalize(vmin=0, vmax=np.max(mises) / 1e6)
surface = Poly3DCollection(display_quads, cmap=theme.FIELD, norm=norm)
surface.set_array(np.asarray(mises).ravel() / 1e6)
ax.add_collection3d(surface)
ax.auto_scale_xyz(*(display_quads.reshape(-1, 3).T))
ax.set_box_aspect(np.ptp(display_quads.reshape(-1, 3), axis=0))
ax.view_init(elev=24, azim=-62)
ax.yaxis.set_major_locator(MaxNLocator(nbins=3))
ax.xaxis.labelpad = 10
ax.yaxis.labelpad = 8
ax.set(xlabel="x [mm]", ylabel="y [mm]", zlabel="z [mm]",
       title=f"Cantilever, displacement magnified {magnification:g} times")
fig.colorbar(surface, ax=ax, label="Von Mises stress [MPa]", shrink=0.8)
fig.savefig(out / "cantilever-surface.png", dpi=220)
plt.show()
```



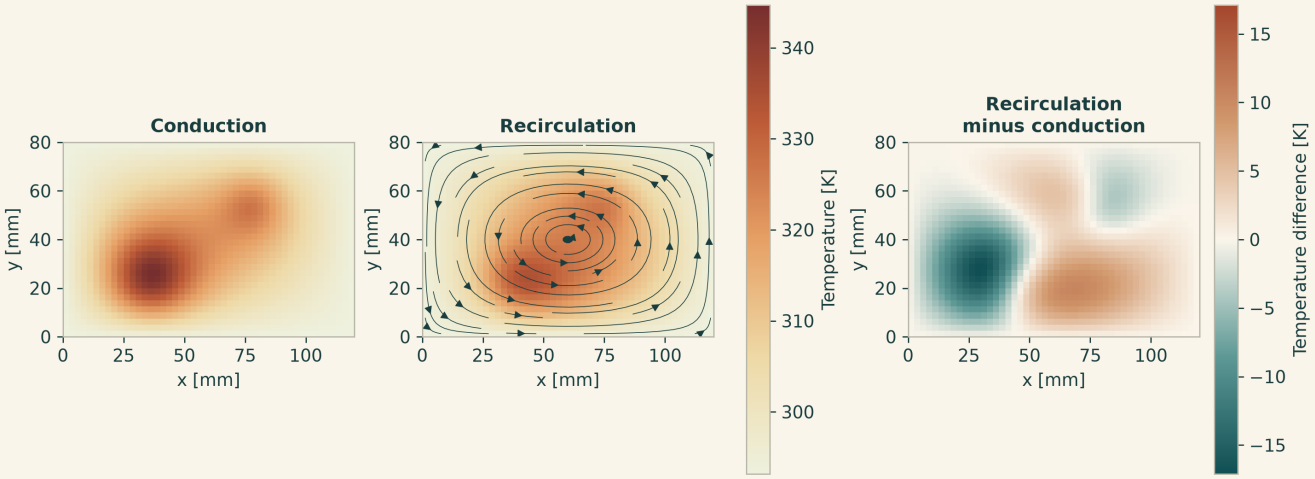
3.2 Heat transport with prescribed recirculation

This model solves the stationary advection–diffusion balance for temperature in a thin volume. Use a velocity tangent to the walls with zero divergence and no z component. Compare the same sources and boundary conditions with pure conduction and with circulation. Use constant ρ, c_p and k. `IsotropicSolid` supplies the coefficients to the thermal model. The velocity is a prescribed field in this study. In the recirculation panel, dark streamlines show the input velocity and the colors show the solved temperature.

```
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from scipy.special import erf

from onesim.cad.d3 import Box
from onesim.materials import IsotropicSolid
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fvm import (
    CentralDifference, FiniteVolume, LeastSquaresGradient, LinearUpwind,
)
from onesim.numerics.linear import SparseDirect
from onesim.physics.heat_transfer import (
    ConvectiveHeatTransfer, FixedTemperature, HeatConduction, HeatFlux,
)
from onesim.simulation import Problem, Simulation

L, W, H = 0.120, 0.080, 0.002  # m
ambient, speed, conductivity = 293.15, 0.0006, 2.0  # K, m/s, W/(m K)
sources = ((0.036, 0.024, 0.008, 1e6), (0.078, 0.054, 0.006, 8e5))

def velocity(x, y, z):
```

```
    sx, sy = np.pi*x/L, np.pi*y/W
    return np.column_stack((
        speed*np.sin(sx)*np.cos(sy),
        -speed*(W/L)*np.cos(sx)*np.sin(sy),
        np.zeros_like(x),
    ))

def heating(x, y, z):
    return sum(a*np.exp(-((x-x0)**2 + (y-y0)**2)/(2*s**2))
               for x0, y0, s, a in sources)

def gaussian_integral(length, center, width):
    d = np.sqrt(2) * width
    return width*np.sqrt(np.pi/2) * (erf((length-center)/d) + erf(center/d))

source_power = H * sum(
    a*gaussian_integral(L, x0, s)*gaussian_integral(W, y0, s)
    for x0, y0, s, a in sources
)

def solve_heat(cells, *, circulating):
    box = Box(length=L, width=W, height=H)
    material = IsotropicSolid(
        density=1000.0, specific_heat_capacity=1000.0,
        thermal_conductivity=conductivity, youngs_modulus=1e9,
        poisson_ratio=0.3, thermal_expansion_coefficient=0.0,
    )
    args = dict(domain=box.volume, material=material, heating=heating)
    model = (ConvectiveHeatTransfer(velocity=velocity, **args)
             if circulating else HeatConduction(**args))
    b = box.boundaries
    problem = Problem(
        name="Recirculation" if circulating else "Conduction",
        geometry=box, physics=(model,),
        boundary_conditions=(
            FixedTemperature(physics=model, value=ambient,
                             boundary=b.x_min | b.x_max | b.y_min | b.y_max),
            HeatFlux(physics=model, value=0.0, boundary=b.z_min | b.z_max),
        ),
    )
    mesh = StructuredHexahedralMesher(cells=cells).generate(box)
    sim = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(
            spatial=FiniteVolume(
                gradient=LeastSquaresGradient(), diffusion=CentralDifference(),
                convection=LinearUpwind() if circulating else None,
            ),
            linear_solver=SparseDirect(),
        ),
    )
```

```
    sim.validate().require_valid()
    result = sim.solve_stationary()
    field = result.field(model.temperature)
    xyz = np.asarray(field.space.dof_coordinates)
    T = np.asarray(field.values)
    assert np.isfinite(T).all() and T.min() >= ambient - 1e-8

    # Bound FV operator used for a discrete power balance.
    op = result.operators[model.model_id]
    face_power = op.conservative_face_fluxes(T)
    outgoing = float(np.sum(op.geometry.face_incidence @ face_power))
    assert abs(outgoing/source_power - 1) < 2e-4

    order = np.lexsort((xyz[:, 0], xyz[:, 1]))
    shape = (cells[1], cells[0])  # one cell in z
    X = xyz[order, 0].reshape(shape)
    Y = xyz[order, 1].reshape(shape)
    return X, Y, T[order].reshape(shape)

X0, Y0, still = solve_heat((48, 32, 1), circulating=False)
X, Y, moving = solve_heat((48, 32, 1), circulating=True)
np.testing.assert_allclose(X, X0)
np.testing.assert_allclose(Y, Y0)
difference = moving - still
assert np.max(np.abs(difference)) > 5.0
limits = (min(still.min(), moving.min()), max(still.max(), moving.max()))
theme.use_theme()
fig, axes = plt.subplots(1, 3, figsize=(11, 4), layout="constrained")
for ax, T, title in zip(axes[:2], (still, moving), ("Conduction", "Recirculation")):
    im = ax.pcolormesh(1e3*X, 1e3*Y, T, shading="nearest", cmap=theme.THERMAL,
                       vmin=limits[0], vmax=limits[1])
    ax.set_title(title)
vx, vy, _ = velocity(X.ravel(), Y.ravel(), np.zeros(X.size)).T
axes[1].streamplot(1e3*X[0], 1e3*Y[:, 0], vx.reshape(X.shape),
                   vy.reshape(Y.shape), color=theme.INK, linewidth=0.45,
                   density=0.65)
fig.colorbar(im, ax=axes[:2].tolist(), label="Temperature [K]")
limit = np.max(np.abs(difference))
im = axes[2].pcolormesh(1e3*X, 1e3*Y, difference, shading="nearest",
                        cmap=theme.DIVERGING, vmin=-limit, vmax=limit)
axes[2].set_title("Recirculation\nminus conduction")
fig.colorbar(im, ax=axes[2], label="Temperature difference [K]")
for ax in axes:
    ax.set(xlabel="x [mm]", ylabel="y [mm]", aspect="equal")
out = Path("onesim-output")
out.mkdir(exist_ok=True)
fig.savefig(out / "recirculation.svg")
plt.show()
```



For a refinement check, call the same solver at a second resolution. Interpolate the fine temperature at the coarse centers inside its support before computing RMS and maximum differences. For a conductive flux arrow, evaluate `result.field(model.heat_flux)` at the physical sample points. A conservative balance uses the assembled face powers. Summing their cell incidences cancels interior faces. The face powers include the transport terms used by the FV equation. The sampled Fourier flux has units W/m^2 and describes conduction. They answer different physical questions. Access to `result.operators` is specific to the bound discretization and belongs in a clearly identified diagnostic cell.

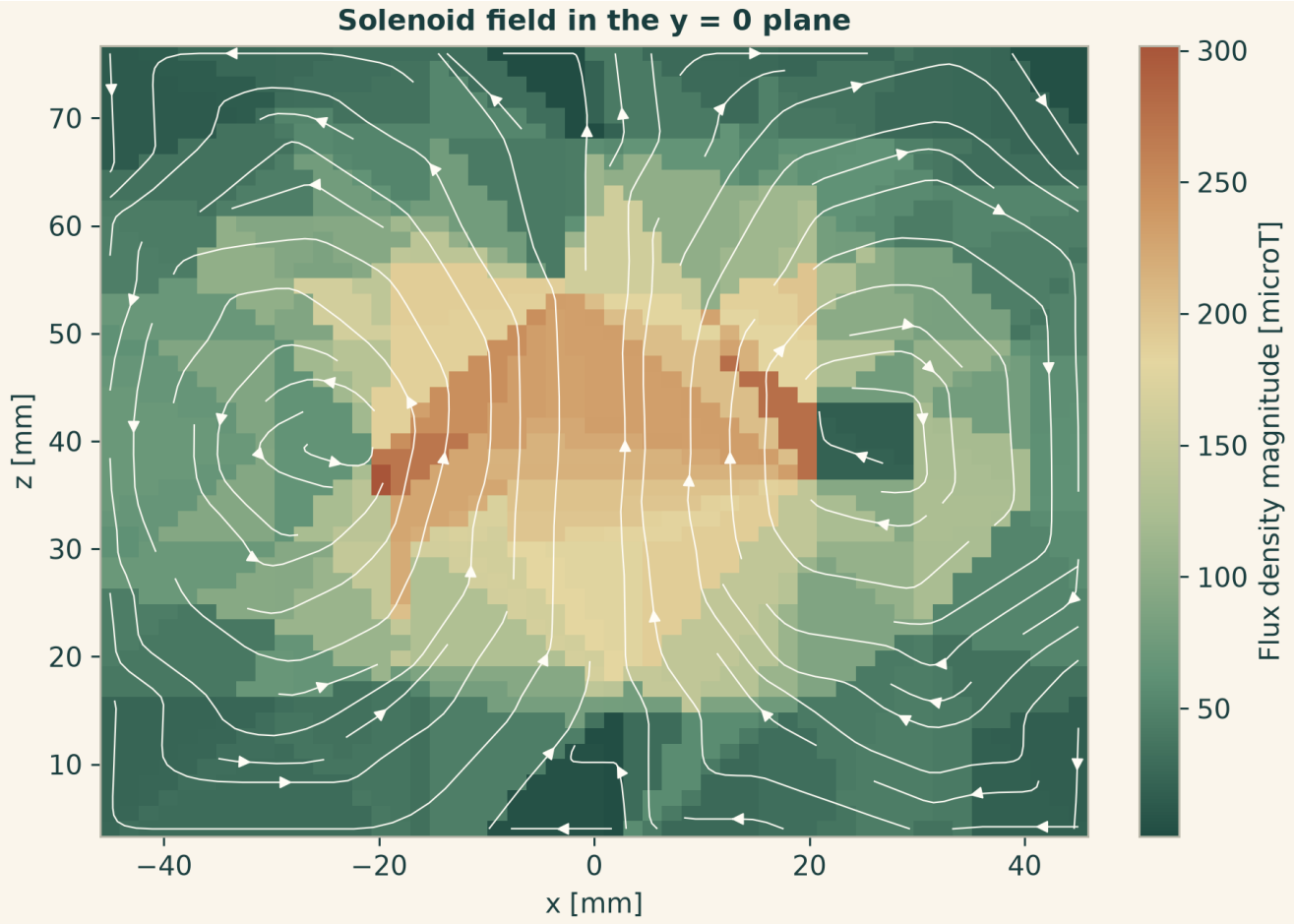
3.3 Magnetostatics of a homogenized solenoid

The winding is a resolved annular volume with an azimuthal impressed current density. The exterior cylinder approximates a finite surrounding domain. The H(curl) basis gives a vector potential **A** whose curl yields **B**. The weak Coulomb gauge supplies a constraint. Use an explicit magnetic boundary, since domain truncation affects the field. The independent checks below use $\mathbf{B} = \mu\mathbf{H}$ and linear current scaling under constant permeability.

```
from dataclasses import replace
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

from onesim.cad.d3 import Cylinder
from onesim.materials import VacuumPermeability
from onesim.meshing import LocalRefinement, TetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Nedelec
from onesim.numerics.linear import SparseDirect
from onesim.physics.magnetostatics import (
    CoulombGauge, MagneticInsulation, MagneticVectorPotential,
    Magnetostatics, SolenoidCurrent,
)
from onesim.simulation import Problem, Simulation

geometry = Cylinder(radius=0.05, length=0.08)
coil = SolenoidCurrent(
```

```
    inner_radius=0.02, outer_radius=0.03, length=0.02,
    center=(0, 0, 0.04), turns=10, current=1.0,
)
mesh = TetrahedralMesher(
    target_size=0.017, embedded_regions=(coil.support,),
    refinements=(LocalRefinement(region=coil.support, target_size=0.008),),
).generate(geometry)
mu = VacuumPermeability()

def solve_coil(source):
    model = Magnetostatics(
        domain=geometry.volume, permeability=mu, current_sources=(source,),
        formulation=MagneticVectorPotential(gauge=CoulombGauge()),
    )
    sim = Simulation(
        problem=Problem(
            name="Homogenized solenoid", geometry=geometry, physics=(model,),
            boundary_conditions=(
                MagneticInsulation(boundary=geometry.boundaries.all, physics=model),
            ),
        ),
        mesh=mesh,
        numerics=Numerics(spatial=FiniteElement(element=Nedelec(order=1)),
                          linear_solver=SparseDirect()),
    )
    sim.validate().require_valid()
    return model, sim.solve_stationary()

model, result = solve_coil(coil)
double_model, double_result = solve_coil(replace(coil, current=2.0))
B_field = result.field(model.magnetic_flux_density)
H_field = result.field(model.magnetic_field_strength)
test_points = np.array([[0, 0, 0.04], [0.01, 0.005, 0.045], [0.035, 0, 0.06]])
B0 = B_field.evaluate(test_points)
H0 = H_field.evaluate(test_points)
B2 = double_result.field(double_model.magnetic_flux_density).evaluate(test_points)
assert 0 < len(mesh.region_cells(coil.support)) < mesh.cell_count
assert B0[0, 2] > 0
assert np.linalg.norm(B0 - float(mu.evaluate())*H0) / np.linalg.norm(B0) < 1e-12
assert np.linalg.norm(B2 - 2*B0) / np.linalg.norm(2*B0) < 1e-8

x, z = np.linspace(-0.045, 0.045, 51), np.linspace(0.004, 0.076, 51)
X, Z = np.meshgrid(x, z)
points = np.column_stack((X.ravel(), np.zeros(X.size), Z.ravel()))
B = B_field.evaluate(points).reshape(X.shape + (3,))
assert np.isfinite(B).all()
theme.use_theme()
fig, ax = plt.subplots(figsize=(7, 5), layout="constrained")
im = ax.pcolormesh(1e3*X, 1e3*Z, 1e6*np.linalg.norm(B, axis=-1),
                   shading="auto", cmap=theme.FIELD)
ax.streamplot(1e3*x, 1e3*z, B[..., 0], B[..., 2],
```

```
              density=0.8, color=theme.STREAMLINE, linewidth=0.6)
ax.set(xlabel="x [mm]", ylabel="z [mm]", aspect="equal",
       title="Solenoid field in the y = 0 plane")
fig.colorbar(im, ax=ax, label="Flux density magnitude [microT]")
out = Path("onesim-output")
out.mkdir(exist_ok=True)
fig.savefig(out / "solenoid.svg")
plt.show()
```



The streamlines display the projection onto the cut plane and the colors show the magnitude of all three vector components. Mark the projected winding cross-sections in a publication figure. The example notebook adds a weak gauge residual check and exports all three vector components with the source-region cell indices. Reuse a mesh for a current-only change with the same source support. If you move or resize the winding, regenerate the mesh with the new embedded region. Vary the outer cylinder and the local mesh size separately to distinguish truncation error from discretization error.

CHAPTER 4

Read, visualize and export results

4.1 Choose a physical view

`result.field(ref)` accepts the reference belonging to the model instance used in the solve. A `SolvedField` exposes its native `space`, `values`, `coefficients` and `unit`. The values of an FV scalar are usually cell-supported and the coefficients of a Nedelec field are edge moments. Use `evaluate(points)` for physical values at explicit world points. Use `at(time)` or `evaluate(points, time=...)` when selecting a retained state. Inspect `result.times` or the recorded output times before choosing frames. For line profiles, `field.sample(along=..., points=..., time=...)` and `result.sample(field=..., along=..., points=..., time=...)` use the sampling carrier rather than a reshaped coefficient vector.

`field.integrate(over=..., time=...)` is appropriate when the field and its support supply native integration. Use `region=` or `boundary=` for a named subset, and specify `component=` or `normal_component=True` when the physical integral is a scalar component or outward flux. Verify support, sign, and geometric measure. For example, a temperature integrated over a volume has units K m^3. Dividing by the volume gives an average in K. A boundary flux integral can be a power, provided the flux is defined on that support with the correct outward normal. For discrete FV conservation, use the executed conservative face fluxes as in the thermal study.

4.2 Use a built-in viewer or customize the panel

`field.plot(...)` offers an application-facing plot entry point. The visualization package also takes the `result.result_set` and `ref.field_id`. For example, `onesim.visualization.line_plot` accepts explicit `start=` and `end=` points. `slice_plot` accepts a `plane=` and can render the cut using the native carrier, with controls for time, component, and complex representation. `result_widget` gives a notebook viewer when the relevant optional dependencies are installed. Import the geometry and animation functions from their owning subpackages because the parent attributes with the same names refer to the packages.

For a custom panel, first build arrays with well-defined shape and support. A regular cut can use `meshgrid`, one batched `evaluate` call, and a reshape that matches the grid order. For an unstructured surface, use its actual connectivity or native cut topology. Keep discontinuous cell values piecewise constant unless you explicitly describe the reconstruction used for display. Use an aligned coordinate and color scale for comparisons. For phasors choose real part, magnitude or phase and state the convention. For a transient animation, use a fixed color normalization and label the time of every frame.

4.3 Keep verification and presentation together

Save vector SVG or PDF for clean line art and text. Use PNG when the figure is dense or the target viewer requires a raster image. Save raw arrays with coordinates and units in CSV or NPZ, and include the runtime version, mesh and time parameters, metrics and export filenames in a small JSON record. Check those metrics before recording a successful run. `HDF5OutputSink` supplies a storage option for supported larger outputs. The memory sink holds selected snapshots in process memory. For long runs choose an explicit `ExecutionOptions` retention policy and verify that the requested fields and times remain available. Use the checkpoint API when continuation requires numerical history, for example after a multistep time method. A CSV of sampled fields represents observables and usually lacks the state and history needed for restart.

```
# Run after the cantilever example.
import json
from importlib.metadata import version

summary = {
    "runtime": version("onesimulation"),
    "fine_cells": [20, 4, 3],
    "element": "Q2",
    "tip_m": float(u[-1, 2]),
    "relative_mesh_change": float(abs(coarse_tip / u[-1, 2] - 1)),
    "figure": "cantilever.svg",
}
(out / "cantilever-verification.json").write_text(
    json.dumps(summary, indent=2) + "\n", encoding="utf-8"
)
```

PART II

Physics model selection

CHAPTER 5

Build a first case in each physics package

The following sections use the canonical import path shown under each heading. Paths beginning with `test_` refer to `implementation/tests/api_target/` in the source checkout. They give specific model and method combinations with verification criteria. The configuration, provider, state and result types in several packages also support a domain service route. Start from the model declaration when it provides the field-based workflow used in this guide. For the service route, construct the matching configuration and use the facade's validation and execution methods. Pass state and result objects through the corresponding provider and adapter family. Appendix A identifies those types and their arguments.

5.1 Reusable periodic-line setup

Several reference cases use periodic endpoints to isolate wave propagation or diffusion from boundary effects. Construct the model on the same line passed to the helper. The returned fields remain on the selected numerical carrier. When comparing against a Fourier mode, use point values for nodal or sample data and cell averages for FV data. For a uniform mesh of N cells per wavelength, the cell mean of the sinusoidal part is the center value multiplied by $\operatorname{sinc}(1/N)$ in NumPy's normalized convention.

```
import numpy as np
from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics import Numerics
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

def run_periodic_line(line, model, initial, numerics, *, cells, end, dt):
    mesh = StructuredLineMesher(cells=cells).generate(line)
    problem = Problem(
        name=type(model).__name__, geometry=line, physics=(model,),
        initial_conditions=tuple(InitialValue(ref, value) for ref, value in initial),
        boundary_conditions=(PeriodicBoundary(
            physics=model, pair=PeriodicPair(line.boundaries.start,
            line.boundaries.end),
        ),),
    )
    sim = Simulation(problem=problem, mesh=mesh, numerics=numerics)
    sim.validate().require_valid()
    return sim.run(start_time=0.0, end_time=end, time_step=dt)
```

5.2 Acoustics

`onesim.physics.acoustics`

Choose `LinearAcoustics` for linear pressure and axial particle velocity on a line. Supply density in kg/m^3, sound speed in m/s, damping if needed and the initial pressure and velocity. A traveling sinusoid uses $u = p/(\rho c)$ for the positive propagation direction. A reflecting wall, a periodic pair and a prescribed flux represent different physical experiments. Verified routes include FD with a polynomial derivative, FV with a Rusanov flux and MUSCL reconstruction, and DG with a Rusanov face flux. Start with `test_acoustic_method_flexibility.py` for the line case and plot pressure versus arclength at successive times.

Choose `PressureDrivenMode` for a single mechanical mode with force equal to effective area times applied pressure. `RadiatingMode` adds a constant radiation resistance. `UltrasonicAcoustics` uses the linear damped wave model. The structural-acoustics, vibroacoustics and ultrasound facades have their own provider contracts.

```
# Use run_lumped and scalar_series from the cooling chapter.
from onesim.physics.acoustics import PressureDrivenMode

domain = LumpedDomain()
mode = PressureDrivenMode(
    domain=domain.system, mass=2.0, stiffness=3.0, area=0.5,
    damping=0.4, pressure_loading=4.0,
)
_, samples, time = run_lumped(
    domain, mode, ((mode.displacement, 0.0), (mode.velocity, 0.0)),
    (mode.displacement, mode.velocity), end=1.0, dt=0.01, interval=0.05,
)
displacement = scalar_series(samples, mode.displacement)
assert displacement[-1] > 0
```

5.3 Biophysics

`onesim.physics.biophysics`

`PassiveMembrane` models an electrical compartment, `OrganClearance` a well-mixed amount and elimination balance, `CompliantVessels` two pressure compartments linked by a `VesselSegment`, and `AxialTissue` a mechanical mass connected through a `TissueElement`. These use global degrees of freedom. Set the initial voltage, amount, pressures or mechanical state before a transient solve. Choose a small explicit step for a first linear comparison and use a supported implicit method for stiff relaxation. The Bioheat provider is a separate thermal result and provider route. For a coupled example, `test_biophysics_input_flexibility.py` exchanges global typed outputs and inputs and compares with a matrix exponential.

```
# Use run_lumped and scalar_series from the cooling chapter.
from onesim.physics.biophysics import OrganClearance

domain = LumpedDomain()
organ = OrganClearance(domain=domain.system, volume=2.0, clearance=1.0)
_, samples, time = run_lumped(
    domain, organ, ((organ.amount, 1.0),),
    (organ.amount, organ.concentration), end=1.0, dt=0.01, interval=0.1,
)
```

```
np.testing.assert_allclose(
    scalar_series(samples, organ.amount), np.exp(-0.5*time), atol=1e-9,
)
```

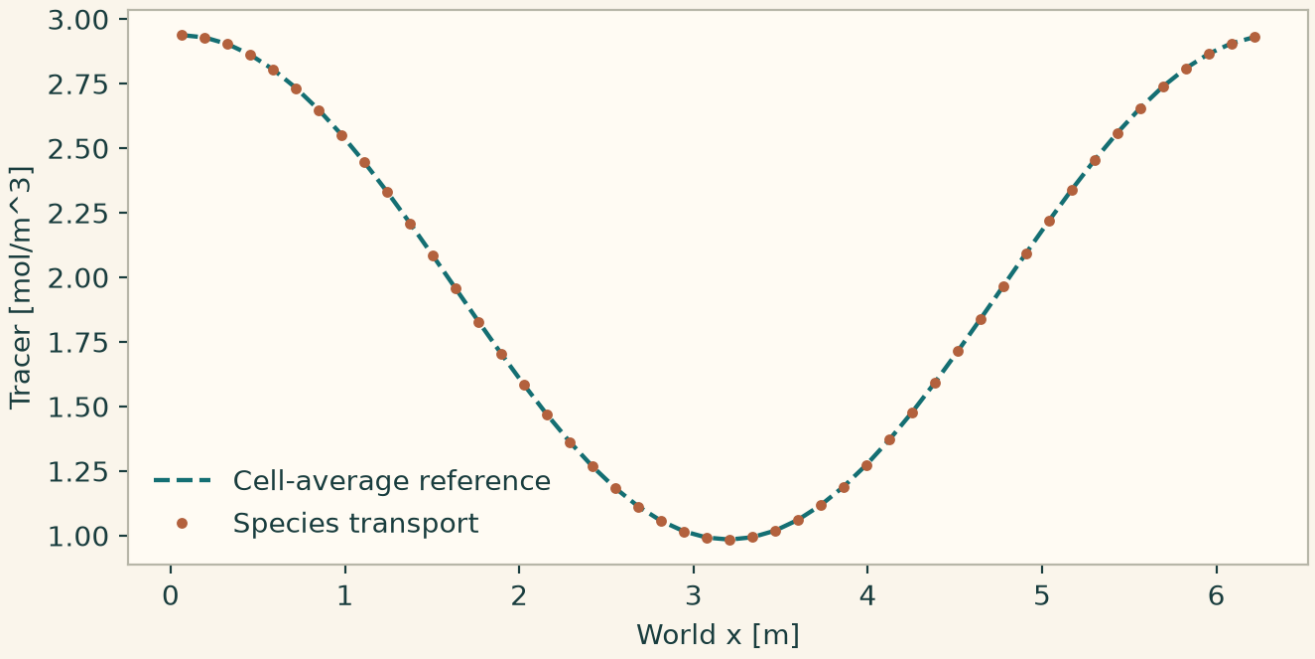
5.4 Chemistry

`onesim.physics.chemistry`

Choose `SpeciesTransport` for line advection, diffusion and decay. Use a `TransportSpecies` entry per stable species ID and obtain each field with `model.concentration(species_id)`. Initialize all transported concentrations on the same spatial support. Set diffusivity in m^2/s and first-order decay in s^{-1}. A periodic Fourier mode translates at the prescribed velocity and attenuates at a rate given by diffusion plus decay. For an open domain include boundary transport in the budget. For FD, FV, H1 and DG variants see `test_species_transport_flexibility.py`.

```
# Use run_periodic_line from this chapter.
from onesim.numerics.fvm import FiniteVolume, MUSCL, Rusanov, TwoPointDiffusion
from onesim.physics.chemistry import SpeciesTransport, TransportSpecies

line = Line(start=(0, 0, 0), end=(2*np.pi, 0, 0))
v, diffusion, decay = 0.6, 0.03, 0.2
transport = SpeciesTransport(
    domain=line.line, velocity=v,
    species=(TransportSpecies("tracer", diffusivity=diffusion, decay_rate=decay),),
)
tracer = transport.concentration("tracer")
n, t = 48, 0.1
result = run_periodic_line(
    line, transport, ((tracer, lambda x: 2 + np.cos(x)),),
    Numerics(
        spatial=FiniteVolume(convection=Rusanov(), reconstruction=MUSCL(),
                             diffusion=TwoPointDiffusion()),
        time_integrator=ClassicalRungeKutta4(),
    ),
    cells=n, end=t, dt=0.001,
)
field = result.field(tracer)
x = field.space.dof_coordinates[:, 0]
exact = np.exp(-decay*t) * (
    2 + np.sinc(1/n) * np.exp(-diffusion*t) * np.cos(x - v*t)
)
np.testing.assert_allclose(field.values, exact, atol=0.002, rtol=0)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8,3.4),layout="constrained")
ax.plot(x, exact, "--", label="Cell-average reference")
ax.plot(x, field.values, ".", label="Species transport")
```

```
ax.set(xlabel="World x [m]",ylabel=f"Tracer [{tracer.unit}]")
ax.legend(frameon=False)
plt.show()
```



For homogeneous kinetics, build `ReactionNetworkIR` from species and reaction declarations, use `ReactionKinetics`, and initialize each transported number density. `Chemistry` and `ChemistryKinetics` describe the evaluation contract for the network. Check stoichiometric invariants and the rate-law units before choosing a time integrator. Strongly separated reaction times suggest comparing against a small-step reference and using an implicit stage with a compatible rate derivative. `LumpedElectrode` composes Butler–Volmer reactions and a double layer. `LumpedBattery` uses `BatteryChemistry` for open-circuit and resistance closures. Plot voltage, state of charge and temperature against time using a clearly stated current sign. See `test_kinetics_input_flexibility.py` for a network-driven global model.

5.5 Electromagnetics

`onesim.physics.electromagnetics`

For an AC field, choose `FrequencyDomainMaxwell`. Supply the frequency in Hz, an electromagnetic material and the phasor and amplitude conventions. The stationary solve returns complex phasors at that frequency. Add explicit PEC, impedance, waveguide port or periodic conditions suited to the experiment. `WaveguideModeBasis` describes incident modes and relative amplitudes. Choose a compatible H(curl) or mimetic carrier and a complex linear solver. Plot magnitude and phase separately and check absorbed power with the declared amplitude convention.

```
# geometry is a volume geometry. Add the conditions for your experiment.
from onesim.materials.electromagnetic import IsotropicElectromagneticMaterial
from onesim.numerics.fem import FiniteElement, Nedelec
from onesim.numerics.linear import ComplexSparseDirect
from onesim.physics.electromagnetics import FrequencyDomainMaxwell
```

```
em = FrequencyDomainMaxwell(
    domain=geometry.volume, frequency=1e7,
    material=IsotropicElectromagneticMaterial(
        relative_permittivity=1.0, relative_permeability=1.0, conductivity=0.001,
    ),
    phasor_convention="exp(-i*omega*t)", amplitude_convention="peak",
)
numerics = Numerics(
    spatial=FiniteElement(element=Nedelec(order=1)),
    linear_solver=ComplexSparseDirect(),
)
```

The fully sourced periodic case in `test_maxwell_method_flexibility.py` supplies a known field for both phasor signs and integrates the absorbed power density. `test_finite_integration_electromagnetics.py` compares supported metric and topological choices. Use `MagneticDiffusion` for a transient induction balance on a line. Use `SeriesRLC`, `AveragedBuckConverter`, or `DCMotor` for circuit and electromechanical states on `LumpedDomain`. For an RLC circuit initialize charge and current, then compare the voltage and stored energy to the circuit balance. The averaged buck model uses duty cycle as a physical input.

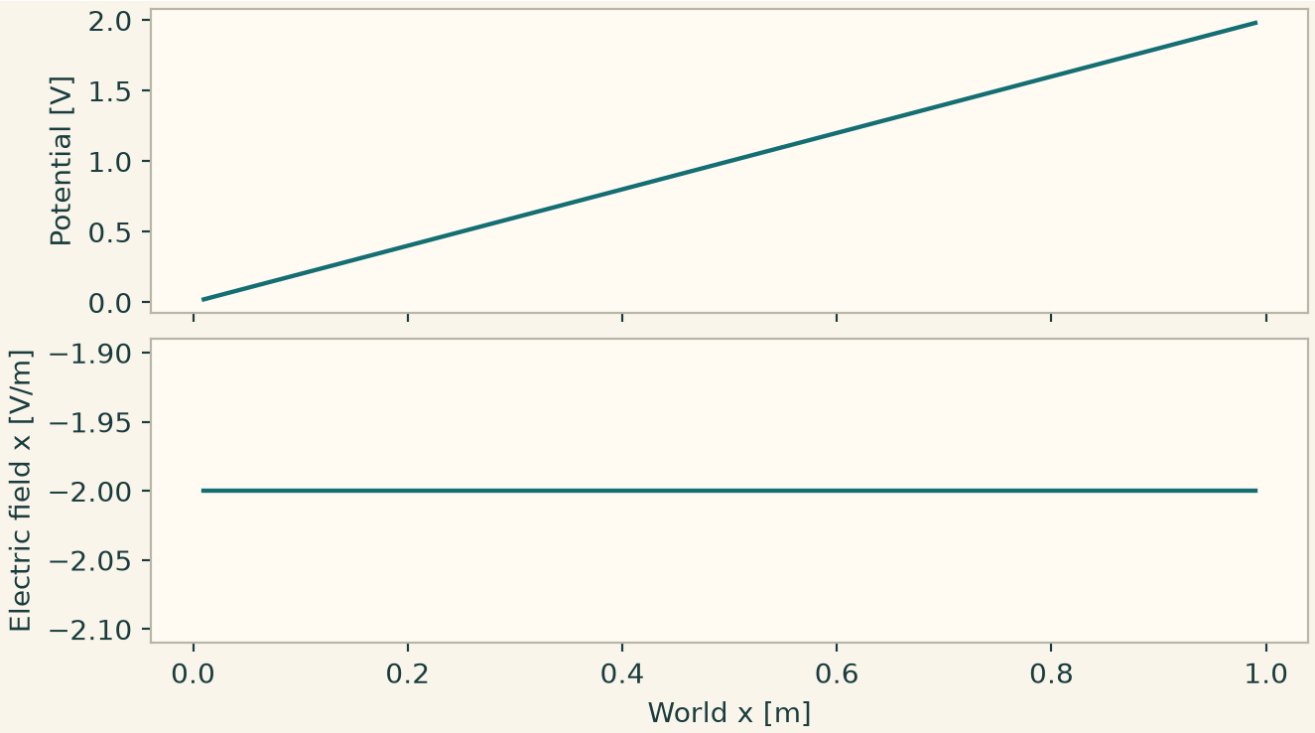
5.6 Electrostatics

`onesim.physics.electrostatics`

`PoissonEquation` solves $-\nabla \cdot (\varepsilon \nabla \phi) = \rho_q$ with $\mathbf{E} = -\nabla \phi$. Specify permittivity in F/m and volume charge in C/m^3. `FixedPotential` sets volts. `NormalElectricDisplacement` prescribes outward $\mathbf{D} \cdot \mathbf{n}$ in C/m^2 and its zero counterpart represents insulation. For an all-flux problem check the integrated charge and flux compatibility and set a supported reference for the additive potential constant. Sample the potential and electric field on the same line, with separate axes or panels.

```
import numpy as np
from onesim.cad.d3 import Box
from onesim.materials import VacuumPermittivity
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.linear import SparseDirect
from onesim.physics.electrostatics import FixedPotential, PoissonEquation
from onesim.simulation import Problem, Simulation

box = Box(length=1.0, width=1.0, height=1.0)
electro = PoissonEquation(domain=box.volume, permittivity=VacuumPermittivity())
problem = Problem(
    name="Uniform electric field", geometry=box, physics=(electro,),
    boundary_conditions=(
        FixedPotential(boundary=box.boundaries.all, physics=electro,
                       value=lambda x: 2*x),
    ),
)
```

```
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
sim = Simulation(
    problem=problem, mesh=mesh,
    numerics=Numerics(spatial=FiniteElement(element=Lagrange(order=1)),
                      linear_solver=SparseDirect()),
)
sim.validate().require_valid()
result = sim.solve_stationary()
np.testing.assert_allclose(
    result.field(electro.potential).evaluate([[0.25, 0.4, 0.6]]), [0.5],
    atol=1e-10,
)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

x = np.linspace(.01,.99,101)
points = np.column_stack((x,np.full_like(x,.4),np.full_like(x,.6)))
potential = result.field(electro.potential).evaluate(points)
electric_x = result.field(electro.electric_field).evaluate(points)[:,0]
np.testing.assert_allclose(electric_x,-2,atol=1e-9)
theme.use_theme()
fig,(a,b)=plt.subplots(2,1,sharex=True,figsize=(6.8,3.8),layout="constrained")
a.plot(x,potential);a.set_ylabel("Potential [V]")
b.plot(x,electric_x);b.set(xlabel="World x [m]",ylabel="Electric field x [V/m]")
plt.show()
```



For supplied charge from another mesh, see `test_cell_projection_poisson.py`. That test projects a native cell field to FE quadrature through an explicit `FieldCoupling`.

5.7 Geotechnics

`onesim.physics.geotechnics`

`GroundwaterFlow` advances hydraulic head with conductivity, specific storage and recharge. A spatially periodic mode with constant coefficients and zero recharge has mean head conserved and amplitude decaying with diffusivity K/S_s. The shifted line in `test_public_groundwater.py` checks that the actual coordinate enters the source and initial field. Use FD central difference and central Laplacian on the uniform periodic example. Plot head and Darcy flux against physical distance.

`TerzaghiConsolidation` describes one-dimensional consolidation with prescribed total stress. `BiotConsolidation` adds elastic displacement and fluid-content coupling. Distinguish drained pressure conditions, prescribed outward discharge and impermeability. Read pore pressure and displacement together and check a storage or consolidation balance. `test_consolidation_method_flexibility.py` and `test_consolidation_physical_ports.py` show concrete declarations. For a saturation front use `BuckleyLeverettFlow` with a fractional-flow law such as `QuadraticCoreyFractionalFlow`, a saturation inlet and an outflow. Compare front position and water balance across meshes and inspect the bounds $0 \le S \le 1$.

5.8 Heat transfer

`onesim.physics.heat_transfer`

Use `HeatConduction` for a spatial Fourier balance with material properties and optional volumetric heating in $\mathrm{W/m^3}$. Supply `FixedTemperature` in K, `HeatFlux` as positive outward flux in $\mathrm{W/m^2}$, or `ConvectiveHeatFlux` with an ambient temperature and transfer coefficient. `RadiativeHeatFlux` applies a nonlinear grey diffuse boundary law to a spatial model. `ConvectiveHeatTransfer` adds a prescribed velocity field to the volume problem, as in Chapter 3. `HeatAdvectionDiffusion` is the line model with specified thermal diffusivity and advection speed. `RadiationCooling` and `PhaseChangeHeating` use lumped thermal states. Phase change uses `ThermalPhaseChangeLawSet` and a finite mushy interval. Plot temperature and enthalpy or liquid fraction across the transition to make the latent-heat effect visible.

For a first conduction notebook, prescribe a linear temperature trace on all faces of a box, set zero volumetric source and choose P1 FEM. A linear solution and constant conductive flux give independent verification. See `test_public_stationary.py`. The source exposes separate thermal service configurations, validation reports and feature entry points for integrations that use the facade instead of `Simulation`.

5.9 Hydrodynamics

`onesim.physics.hydrodynamics`

For incompressible flow, construct `IncompressibleNavierStokes` with density and dynamic viscosity, usually via `NewtonianFluid`. Select physical wall conditions such as `NoSlipWall`, `FixedVelocity` or `RotatingWall`. A `PressureTraction` imposes $-p_{\mathrm{ext}}\mathbf{n}$ and a `FlowTraction` specifies the full outward Cauchy traction. When the conditions leave pressure determined up to a constant, apply a supported pressure reference such as `MeanValue`. When a physical pressure traction already fixes it, use that pressure datum. Choose a compatible mixed FEM pair or an FV pressure–velocity strategy. Compare a laminar velocity profile, divergence and net boundary flow before plotting a streamline panel. `test_mixed_flow_natural_boundaries.py` gives a

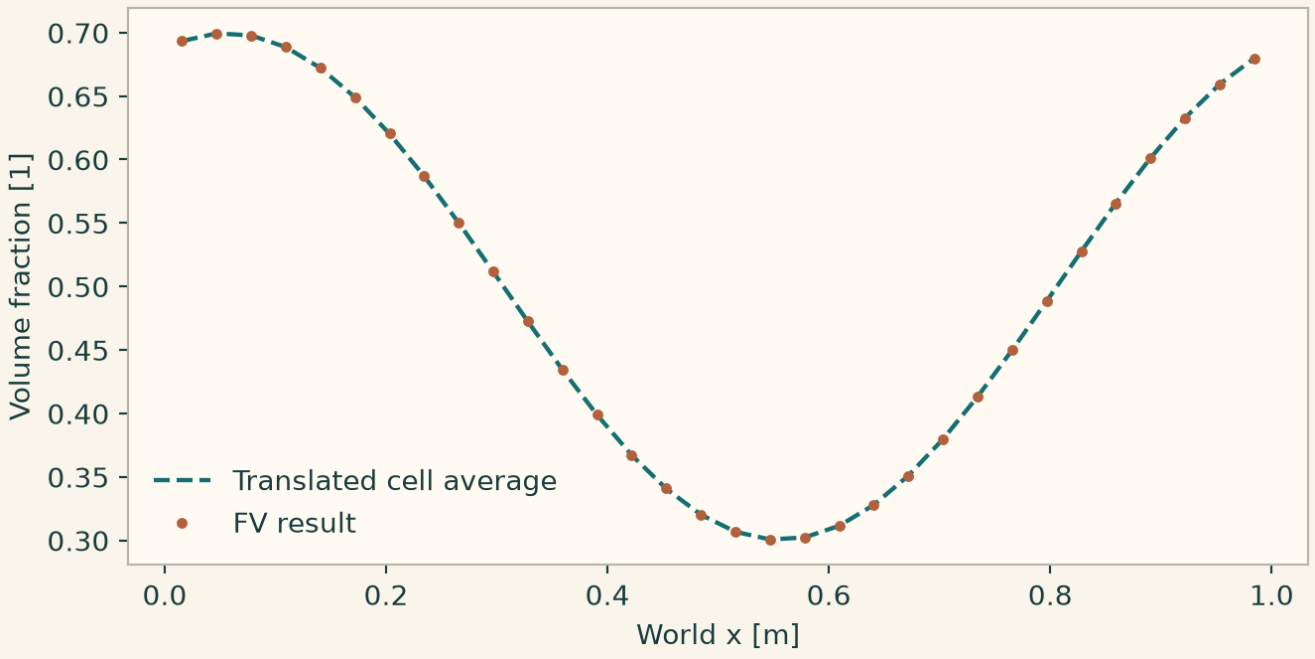
Taylor–Hood example with a known pressure and velocity.

For compressible flow, initialize density, momentum and total energy with a consistent `IdealGasEquationOfState`. Choose a conservative method and an `EulerInvariantDomain` where supported. `ShallowWater` uses depth and discharge. `TopographicShallowWater` adds bed elevation and a wet/dry regime. Verify lake-at-rest preservation before a dam-break study with dry cells. `PhaseAdvection` transports a phase fraction and has a corresponding invariant-domain declaration. `CahnHilliard` evolves composition and chemical potential from a free-energy and mobility law. `ReactiveMixtureTransport` adds species and thermal sources, and `OldroydBShear` evolves homogeneous viscoelastic stress. For porous flow, `DarcyPressureFlow` uses pressure, permeability, viscosity and pressure storage. Keep its physical pressure in Pa distinct from groundwater head in metres.

5.9.1 Verify conservative transport of a phase fraction

The periodic test holds the transport speed fixed and starts with a smooth fraction in $[0, 1]$. The integral is independent of advection on the closed periodic domain. Use the cell-supported fraction for the balance and plot it against the translated cell average.

```
# Use run_periodic_line from this chapter.
from onesim.numerics.fvm import FiniteVolume, MUSCL, Rusanov
from onesim.physics.hydrodynamics import PhaseAdvection

line = Line(start=(0, 0, 0), end=(1, 0, 0))
phase = PhaseAdvection(
    domain=line.line, velocity=0.7, first_density=1000.0, second_density=1.0,
)
result = run_periodic_line(
    line, phase, ((phase.volume_fraction, lambda x: 0.5 + 0.2*np.cos(2*np.pi*x)),),
    Numerics(spatial=FiniteVolume(convection=Rusanov(), reconstruction=MUSCL()),
             time_integrator=ClassicalRungeKutta4()),
    cells=32, end=0.08, dt=0.001,
)
fraction = result.field(phase.volume_fraction)
assert abs(fraction.integrate().value - 0.5) < 2e-12
assert np.all((fraction.values >= 0) & (fraction.values <= 1))

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

x = fraction.space.dof_coordinates[:,0]
reference = .5+.2*np.sinc(1/32)*np.cos(2*np.pi*(x-.7*.08))
theme.use_theme()
fig,ax=plt.subplots(figsize=(6.8,3.4),layout="constrained")
ax.plot(x,reference,"--",label="Translated cell average")
ax.plot(x,fraction.values,"o",markersize=3,label="FV result")
ax.set(xlabel="World x [m]",ylabel="Volume fraction [1]")
ax.legend(frameon=False)
plt.show()
```



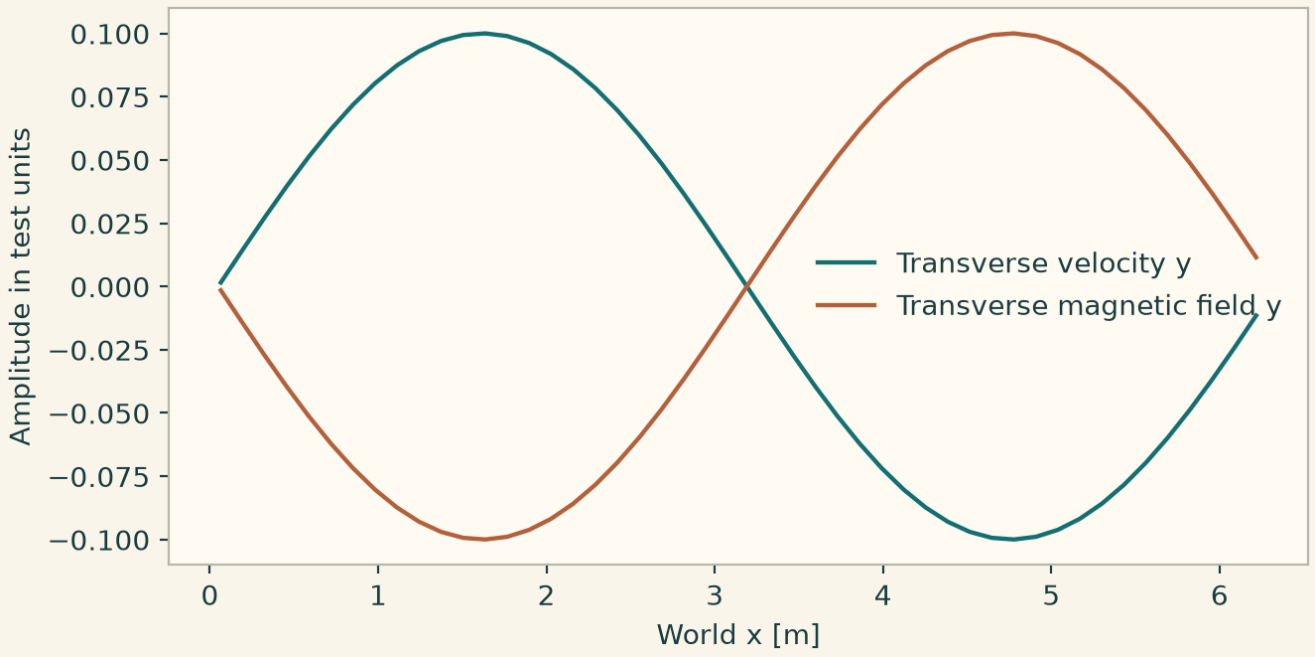
5.10 Magnetohydrodynamics

`onesim.physics.magnetohydrodynamics`

`TransverseMHDWaves` evolves two velocity and two magnetic perturbations along a background field. Start with periodic endpoints and a single polarization, then add the second. The nominal ideal Alfvén speed is $|B_0|/\sqrt{\rho\mu}$. Compare both propagation direction and component amplitudes. The FD, FV and DG examples in `test_transverse_mhd_method_flexibility.py` include a rotated line, conservation checks and comparison on the actual carrier. `InductionlessMomentum` uses a fixed field in the inductionless regime and a dedicated no-slip condition. `TwoFluidExchange` and `RadiationMatterExchange` are homogeneous exchange models. Initialize both sides of the exchange and test momentum or energy transfer with externally imposed sources accounted for. The result adapters cover additional provider result families. Their names indicate result types and by themselves give no model construction recipe.

```
# Use run_periodic_line from this chapter.
from onesim.numerics.fdm import FiniteDifference, PolynomialDifference
from onesim.physics.magnetohydrodynamics import TransverseMHDWaves

line = Line(start=(0, 0, 0), end=(2*np.pi, 0, 0))
wave = TransverseMHDWaves(
    domain=line.line, density=2.0, permeability=0.5, longitudinal_field=1.0,
)
result = run_periodic_line(
    line, wave,
    ((wave.velocity_y, lambda x: 0.1*np.sin(x)), (wave.velocity_z, 0.0),
     (wave.magnetic_y, lambda x: -0.1*np.sin(x)), (wave.magnetic_z, 0.0)),
    Numerics(spatial=FiniteDifference(gradient=PolynomialDifference()),
             time_integrator=ClassicalRungeKutta4()),
    cells=48, end=0.05, dt=0.001,
)
u = result.field(wave.velocity_y)
x = u.space.dof_coordinates[:, 0]
```

```
np.testing.assert_allclose(u.values, 0.1*np.sin(x - 0.05), atol=5e-4, rtol=0)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

b = result.field(wave.magnetic_y)
theme.use_theme()
fig,ax=plt.subplots(figsize=(6.8,3.4),layout="constrained")
ax.plot(x,u.values,label="Transverse velocity y")
ax.plot(x,b.values,label="Transverse magnetic field y")
ax.set(xlabel="World x [m]",ylabel="Amplitude in test units")
ax.legend(frameon=False)
plt.show()
```



5.11 Magnetostatics

`onesim.physics.magnetostatics`

Assemble the model from permeability, a formulation and current sources. `MagneticVectorPotential(gauge=CoulombGauge())` gives a vector-potential model with a divergence constraint. `SolenoidCurrent` supplies azimuthal winding current and a meshing support. `MagneticInsulation` sets tangential potential to zero. The explicit potential, magnetic-field and reluctance boundary declarations represent other tangential boundary models. Use the coil example in Chapter 3 as a starting point. Record the gauge, outer boundary distance and winding mesh size. For a linear material, plot B/I for several currents on the same axes to verify scaling.

5.12 Multibody dynamics

`onesim.physics.multibody`

`SimplePendulum` advances angle and angular velocity. `EulerTop` evolves rotation about principal inertia axes. `TorsionalDrivetrain` uses shaft stiffness, damping and gear ratios. `DrivenFourBar`

also has geometry and an assembly branch. `LinearJointSystem` adds algebraic constraints to inertial dynamics. For a linkage, check constraint residuals and initial configuration consistency as well as the trajectory. `test_multibody_input_flexibility.py` gives driven examples and tests physical inputs.

```
# Use run_lumped and scalar_series from the cooling chapter.
from onesim.physics.multibody import SimplePendulum

domain = LumpedDomain()
pendulum = SimplePendulum(domain=domain.system, length=0.5)
_, samples, time = run_lumped(
    domain, pendulum,
    ((pendulum.angle, 0.15), (pendulum.angular_velocity, 0.0)),
    (pendulum.angle, pendulum.energy), end=2.0, dt=0.005, interval=0.025,
)
theta = scalar_series(samples, pendulum.angle)
energy = scalar_series(samples, pendulum.energy)
np.testing.assert_allclose(energy, np.full_like(energy, energy[0]), atol=1e-8)
```

Plot angle and phase against time or convert the angle to a bob position for an animation. The small-angle period $2\pi\sqrt{\ell/g}$ provides a reference for small initial angles. Use energy or torque balance for larger amplitudes.

5.13 Multiphysics

`onesim.physics.multiphysics`

Use `HeatedResistor`, `FluidColumnPiston`, `VoiceCoil`, `ThermalExchange` or `AdiabaticReaction` for a packaged lumped balance. These models expose typed states and inputs. For example, a voice coil couples electrical current to force and mechanical velocity to back electromotive force. Initialize current, displacement and velocity, and plot all three against time. For thermal exchange with zero external power, check constancy of $C_1T_1 + C_2T_2$ and the exponential decay of $T_1 - T_2$.

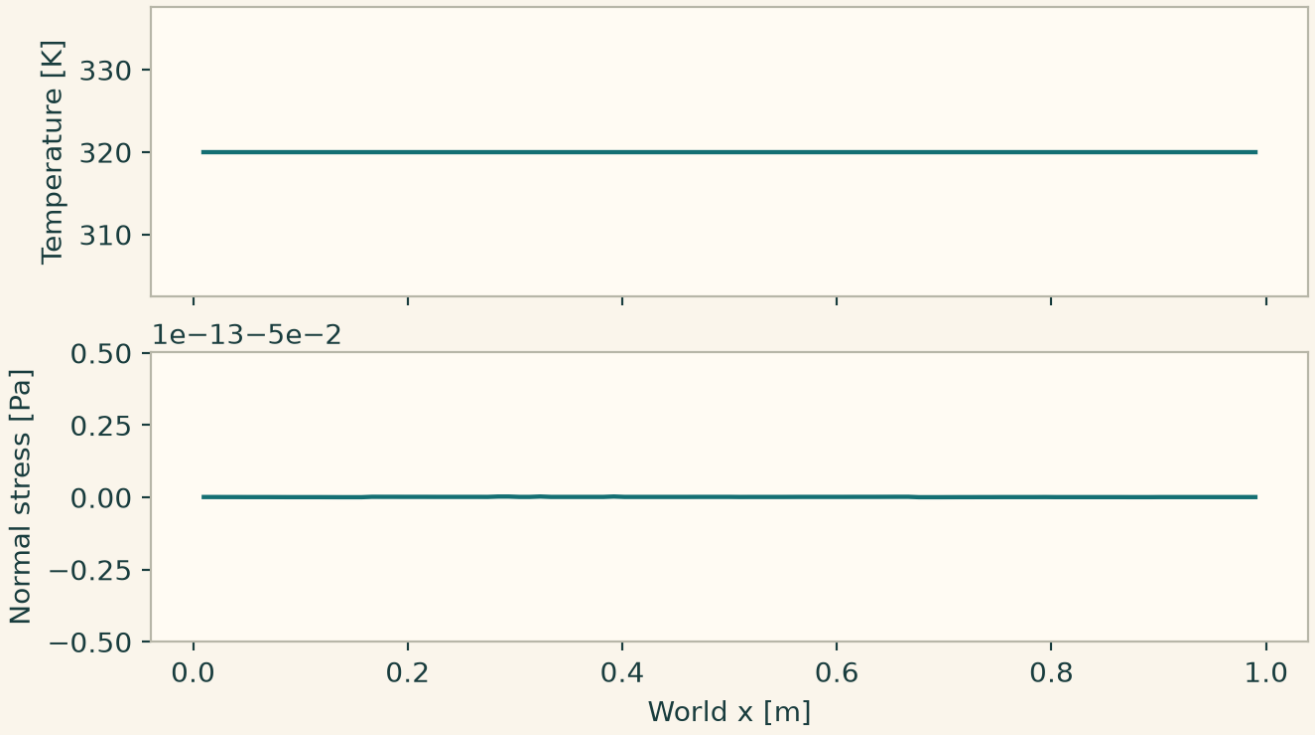
For independently declared physics, compose a `Problem` with both models, attach conditions to their owners and connect typed fields via `FieldCoupling`. Select each mesh and `Numerics` by model identity. The physical link specifies what is transmitted. The numerical transfer specifies how to represent it on the target support. A coupling strategy specifies when to solve each block. For thermoelasticity, use thermal `temperature` as the source and the elastic model's temperature input as the target, with `FiniteElementEvaluation()` to evaluate at mechanical quadrature points. Enable `IsotropicThermalExpansion(reference_temperature=...)` in the elastic model. `test_fe_thermoelastic_coupling.py` checks free expansion and restrained thermal stress. The electrothermal, thermomechanical, fluid–structure and general coupled service exports carry separate configuration and provider contracts.

5.13.1 Transfer a temperature to mechanical quadrature

This stationary test holds the faces at a uniform temperature above the stress-free reference and clamps all displacement components. The expected normal stress is $-E\alpha\Delta T/(1-2\nu)$. The separate FE spaces share a mesh here. The coupling still evaluates the thermal field on the receiver's quadrature rather than copying coefficients.

```
import numpy as np
from onesim.cad.d3 import Box
from onesim.materials import IsotropicSolid
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.field_transfer import FiniteElementEvaluation
from onesim.numerics.linear import SparseDirect
from onesim.physics.heat_transfer import FixedTemperature, HeatConduction
from onesim.physics.solid_mechanics import (
    FixedDisplacement, IsotropicThermalExpansion, LinearElasticity,
)
from onesim.simulation import Problem, Simulation
from onesim.simulation.field_connections import FieldCoupling

box = Box(length=1, width=1, height=1)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
E, nu, alpha = 100.0, 0.3, 1e-5  # Pa, 1, 1/K
T_ref, T_wall = 300.0, 320.0
material = IsotropicSolid(
    density=1, specific_heat_capacity=1, thermal_conductivity=1,
    youngs_modulus=E, poisson_ratio=nu, thermal_expansion_coefficient=alpha,
)
heat = HeatConduction(domain=box.volume, material=material)
solid = LinearElasticity(
    domain=box.volume, material=material,
    thermal_strain=IsotropicThermalExpansion(reference_temperature=T_ref),
)
problem = Problem(
    name="Constrained heating", geometry=box, physics=(heat, solid),
    boundary_conditions=(
        FixedTemperature(boundary=box.boundaries.all, physics=heat, value=T_wall),
        FixedDisplacement(boundary=box.boundaries.all, physics=solid, value=(0, 0, 0)),
    ),
    couplings=(FieldCoupling(
        source=heat.temperature, target=solid.temperature,
        transfer=FiniteElementEvaluation(),
    ),),
)
sim = Simulation(
    problem=problem, mesh={heat: mesh, solid: mesh},
    numerics={
        m: Numerics(spatial=FiniteElement(element=Lagrange(order=p)),
                    linear_solver=SparseDirect())
        for m, p in ((heat, 1), (solid, 2))
    },
)
sim.validate().require_valid()
result = sim.solve_stationary()
point = [[0.2, 0.3, 0.4]]
expected = -E * alpha * (T_wall - T_ref) / (1 - 2*nu)
```

```
np.testing.assert_allclose(result.field(heat.temperature).evaluate(point), [T_wall])
np.testing.assert_allclose(result.field(solid.displacement).evaluate(point), [[0, 0,
    0]],
                           atol=1e-12)
sigma = result.field(solid.stress).evaluate(point)[0]
np.testing.assert_allclose(sigma, [expected]*3 + [0]*3, atol=1e-10)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

x = np.linspace(.01,.99,101)
points = np.column_stack((x,np.full_like(x,.3),np.full_like(x,.4)))
temperature = result.field(heat.temperature).evaluate(points)
normal_stress = result.field(solid.stress).evaluate(points)[:,0]
theme.use_theme()
fig,(a,b)=plt.subplots(2,1,sharex=True,figsize=(6.8,3.8),layout="constrained")
a.plot(x,temperature);a.set_ylabel("Temperature [K]")
b.plot(x,normal_stress);b.set(xlabel="World x [m]",ylabel="Normal stress [Pa]")
plt.show()
```



The low modulus makes this a simple manufactured balance in SI units. To build a free-expansion variant, change the supports to remove rigid-body motion while allowing thermal expansion. Then check displacement and stress against the corresponding physical reference.

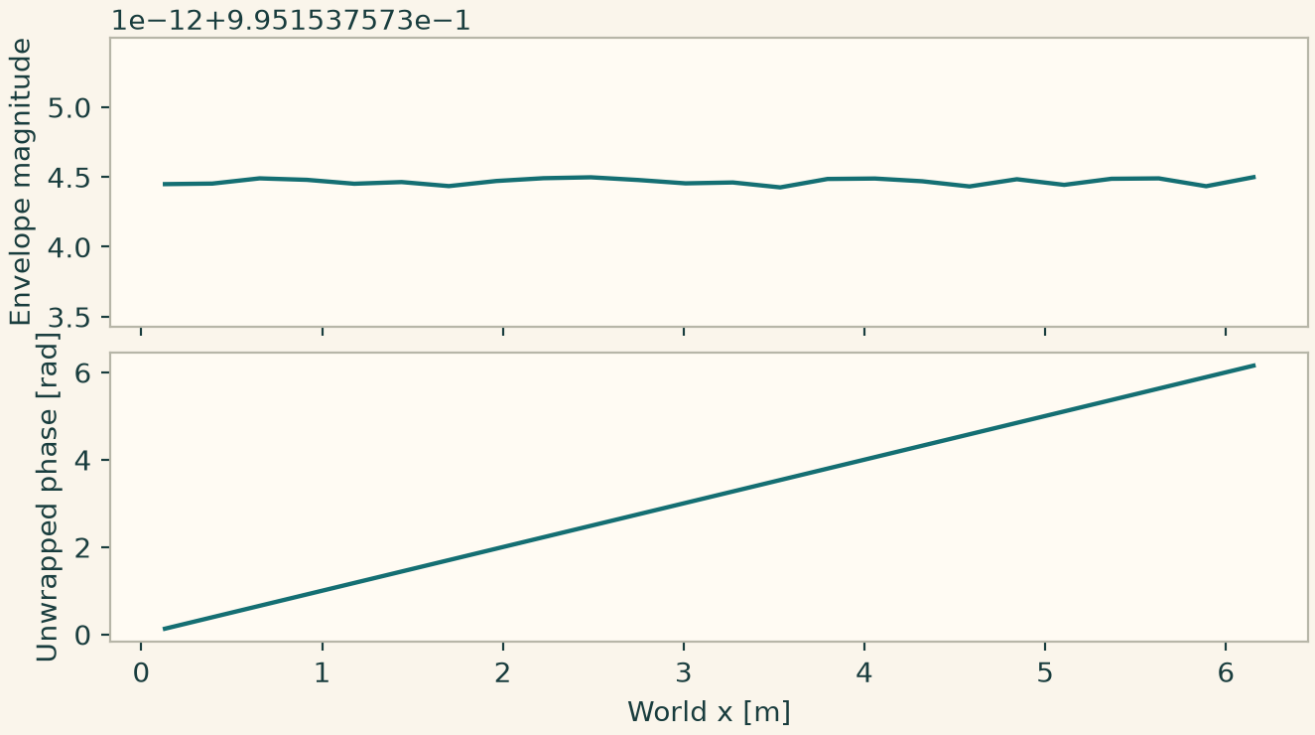
5.14 Optics

`onesim.physics.optics`

Use `ComplexParaxialEnvelope` for a complex field in a frame moving at `group_speed`, `ParaxialEnvelope` for separate real and imaginary fields, and `PhotonEnvelope` for the related

photon-energy observables. Specify wavenumber in m^{-1}, group speed in m/s, permittivity and material or source perturbations. Set a complex initial amplitude for `model.envelope` or two real initial fields for `ParaxialEnvelope`. A periodic Fourier mode isolates diffraction, phase and absorption. Sample complex amplitude on a line and plot $|E|$ and $\arg E$ in separate panels. The propagation coordinate in that model is $z = v_g t$. `test_optics_method_flexibility.py` uses selected FD, FV and DG diffusion, an explicit stability rule or an implicit step. A complex dtype and a stability bound appropriate to a dispersive operator matter when switching methods.

```
# Use run_periodic_line from this chapter.
from onesim.numerics.fvm import FiniteVolume, TwoPointDiffusion
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.optics import ComplexParaxialEnvelope

line = Line(start=(0, 0, 0), end=(2*np.pi, 0, 0))
optical = ComplexParaxialEnvelope(
    domain=line.line, wavenumber=2.0, group_speed=1.0, permittivity=2.0,
    absorption=0.1, refractive_index_perturbation=0.05,
)
result = run_periodic_line(
    line, optical, ((optical.envelope, lambda x: np.exp(1j*x)),),
    Numerics(
        spatial=FiniteVolume(diffusion=TwoPointDiffusion()),
        time_integrator=BackwardEuler(),
        nonlinear_solver=Newton(
            linear_solver=SparseDirect(), jacobian="colored_finite_difference",
            relative_tolerance=1e-11, absolute_tolerance=1e-12,
        ),
    ),
    cells=24, end=0.02, dt=0.005,
)
E = result.field(optical.envelope)
x = E.space.dof_coordinates[:, 0]
reference = np.sinc(1/24) * np.exp(1j*x + (-0.1 - 0.15j)*0.02)
np.testing.assert_allclose(E.values, reference, atol=2e-4, rtol=0)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

theme.use_theme()
fig,(a,b)=plt.subplots(2,1,sharex=True,figsize=(6.8,3.8),layout="constrained")
a.plot(x,np.abs(E.values));a.set_ylabel("Envelope magnitude")
b.plot(x,np.unwrap(np.angle(E.values)))
b.set(xlabel="World x [m]",ylabel="Unwrapped phase [rad]")
plt.show()
```



Plot `np.abs(E.values)` and `np.unwrap(np.angle(E.values))` against the physical coordinates. The first panel shows attenuation and the second shows phase. A complex-valued error comparison checks both quantities at once.

5.15 Plasma physics

`onesim.physics.plasma`

Choose the model at the scale of the question. `ChargedParticle` uses prescribed electric and magnetic inputs and integrates position and velocity. A uniform magnetic field with zero electric field gives a speed and gyrofrequency check. `UniformGuidingCenter` describes a reduced orbit, `ColdPlasmaOscillation` a homogeneous velocity and electric field, and the waterbag declarations evolve velocity contours on a line. For a waterbag initialize ordered lower and upper contours, inspect density and enforce an ordered invariant domain where supported. `test_plasma_uniform_input_flexibility.py` shows dimensioned drive inputs and feedback for the lumped models.

`DriftDiffusionPlasma` composes gas, chemistry, electron energy, electron collisions, magnetized electron transport and ion mobility. Use explicit initial number densities and energy, with background and transported species distinguished. Supply or couple the electric, magnetic and heating inputs with their required units. Choose the wall physics explicitly, for example `AbsorbingPlasmaWall` or `NoParticleFlux`. A line or volume finite-volume assembly can use a compatible Scharfetter–Gummel flux and local source treatment. Use `test_public_plasma_balance.py` and `test_native_multimesh_plasma.py` to check the applicable mesh and source contracts. Verify particle, charge and energy balances before interpreting densities or mean energy.

For ECR coupling, `NativeEcrPlasma` composes the species, energy, Poisson and Maxwell blocks. `NativeEcrDiscretization`, `NativeEcrStateTransfer`, `EcrExcitation` and the potential gauge express separate choices. Keep excitation, phasor, cold-electron response and absorbed-power transfer consistent. Start with the supplied native ECR tests for your intended geometry and execution

route. The public constructors accept explicit data and collaborators. Check which collision and transport data your distribution actually contains.

5.15.1 Check a homogeneous plasma oscillation

Set collision frequency and external current to zero for an independent reference. With initial electron velocity zero and electric field E_0, the field evolves as $E_0\cos(\omega_p t)$, where $\omega_p^2 = n_e e^2/(m_e \varepsilon)$. The coefficients here are scaled test values supplied to the model. For an application, use its actual physical parameters and resolve the resulting time scale.

```
# Use run_lumped and scalar_series from the cooling chapter.
from onesim.physics.plasma import ColdPlasmaOscillation

domain = LumpedDomain()
plasma = ColdPlasmaOscillation(
    domain=domain.system, density=2.0, electron_mass=3.0,
    elementary_charge=1.0, permittivity=4.0, collision_frequency=0.0,
)
_, samples, time = run_lumped(
    domain, plasma,
    ((plasma.velocity, 0.0), (plasma.electric_field, 0.3), (plasma.thermal_energy,
        0.0)),
    (plasma.electric_field, plasma.velocity), end=1.0, dt=0.01, interval=0.05,
)
omega = np.sqrt(2/(3*4))
np.testing.assert_allclose(scalar_series(samples, plasma.electric_field),
                           0.3*np.cos(omega*time), atol=1e-9, rtol=0)
```

5.16 Solid mechanics

`onesim.physics.solid_mechanics`

`LinearElasticity` uses small-strain elasticity and an elastic material. `FixedDisplacement`, `Traction`, `TractionFree` and `BodyForce` describe supports and loads. Remove rigid-body modes through physical constraints and evaluate stress inside the domain. Use the cantilever for a force, deflection and mesh-refinement workflow. For thermal strain, couple a temperature field and set a stress-free reference. Inspect stress component ordering before selecting the first normal component or constructing an invariant.

For global dynamics, `StructuralOscillator` takes mass, stiffness, damping and force. `MaxwellRelaxation` evolves stress under a strain-rate input, `FatigueCrackGrowth` uses `ParisLaw`, `HertzImpact` uses `HertzContactLaw`, and `CohesiveOpening` uses `BilinearCohesiveLaw`. Initialize each law's state, follow its units and check the relevant quantity such as dissipated energy, contact force, crack length or traction versus opening. Use a resolved time step around rapid contact or softening. The vibration example in Chapter 2 gives a complete extraction and plotting pattern.

5.17 Systems and networks

`onesim.physics.systems`

Start with a compact lumped balance such as `ResistorCapacitor`, `ThermalCapacitance`, `HydraulicAccumulator`, `FirstOrderControl`, `WellMixedTank`, or `SealedGasChamber`. State the initial voltage, temperature, pressure or other stored state. Set the forcing directly or connect the

typed input. For a network service, build the topology using the matching node and branch contracts. Examples include `ElectricalNode` and `ElectricalBranch`, `ThermalNode` and `ThermalLink`, or `ProcessUnit` and `ProcessStream`. `SignalBlock` is the corresponding control declaration. Pass them to the configuration for that family, validate node IDs, port orientation and reference conditions, then execute the matching facade. Plot node histories and signed branch rates. Check that internal transfers cancel in the whole-network balance.

```
# Use run_lumped and scalar_series from the cooling chapter.
from onesim.physics.systems import ResistorCapacitor

domain = LumpedDomain()
rc = ResistorCapacitor(
    domain=domain.system, resistance=2.0, capacitance=0.5, supply_voltage=3.0,
)
_, samples, time = run_lumped(
    domain, rc, ((rc.voltage, 0.0),), (rc.voltage, rc.current),
    end=2.0, dt=0.01, interval=0.1,
)
np.testing.assert_allclose(
    scalar_series(samples, rc.voltage), 3*(1 - np.exp(-time)), atol=1e-8,
)
```

CHAPTER 6

Worked physics and service recipes

Each recipe constructs its own model and supplies its own conditions, method and verification. Run the shared cells at the start of this chapter before the recipes that use them. The three helpers serve different contracts. Global histories contain one sample per scalar field and output time. Line profiles retain the location of native coefficients. Service configurations return their own result objects and carriers. The examples in this chapter and the preceding chapters together cover all 73 public physics model declarations in the inspected checkout.

6.1 Shared cell for lumped histories

Use this cell for a single lumped model with scalar field references. Output snapshots contain accepted model state and derived values. Each plotted ordinate inherits its unit from the field definition. Supply the named initial states before choosing observables.

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

from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.linear import SparseDirect
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.results.storage import MemoryOutputSink, Output
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue


def history_case(
    domain, model, initial, observe, *, end, dt, spatial=None, linear_solver=None
):
    refs = tuple(model.field(name) for name in observe)
    problem = Problem(
        name=type(model).__name__,
        geometry=domain,
        physics=(model,),
        initial_conditions=tuple(
            InitialValue(model.field(name), value) for name, value in initial.items()
        ),
    )
    sim = Simulation(
        problem=problem,
        mesh=LumpedMesh(domain),
        numerics=Numerics(
            spatial=LumpedParameters() if spatial is None else spatial,
            time_integrator=ClassicalRungeKutta4(),
```

```
            linear_solver=linear_solver,
        ),
    )
    sim.validate().require_valid()
    sink = MemoryOutputSink()
    sim.run(
        start_time=0,
        end_time=end,
        time_step=dt,
        output=Output(fields=refs, interval=4 * dt, sink=sink),
    )
    t = np.array([float(s.time) for s in sink.snapshots])
    assert np.isclose(t[-1], end)
    data = {
        ref.name: np.array(
            [np.asarray(s.fields[ref.field_id].values).item() for s in sink.snapshots]
        )
        for ref in refs
    }
    assert all(np.isfinite(y).all() for y in data.values())
    return t, data


def history_plot(title, model, t, data):
    theme.use_theme()
    fig, axes = plt.subplots(
        len(data),
        1,
        sharex=True,
        squeeze=False,
        figsize=(7.2, 1.22 * len(data) + 0.85),
        layout="constrained",
    )
    for ax, (name, y) in zip(axes[:, 0], data.items()):
        ref = model.field(name)
        ax.plot(t, y)
        ax.set_ylabel(ref.unit)
        ax.set_title(name.replace("_", " ").capitalize(), loc="left", fontsize=9)
        ax.grid(alpha=0.25)
    axes[-1, 0].set_xlabel("Time [s]")
    fig.suptitle(title, color=theme.INK, fontweight="bold")
    return fig
```

6.2 Shared cell for one-dimensional profiles

Use this cell for the periodic and explicitly bounded line experiments. For nonperiodic examples pass the physical boundary tuple. The display helper expects one real scalar per native coordinate. Complex and vector recipes select a physical component or magnitude explicitly.

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

from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics import Numerics
from onesim.numerics.fdm import FiniteDifference, CentralDifference, CentralLaplacian
from onesim.numerics.fvm import FiniteVolume, Rusanov, MUSCL, TwoPointDiffusion
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import ClassicalRungeKutta4, BackwardEuler
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary


def line_experiment(
    line,
    model,
    initial,
    spatial,
    *,
    cells,
    end,
    dt,
    conditions=None,
    implicit=False,
    linear_solver=None
):
    if conditions is None:
        pair = PeriodicPair(line.boundaries.start, line.boundaries.end)
        conditions = (PeriodicBoundary(physics=model, pair=pair),)
    problem = Problem(
        name=type(model).__name__,
        geometry=line,
        physics=(model,),
        initial_conditions=tuple(
            InitialValue(model.field(name), value) for name, value in initial.items()
        ),
        boundary_conditions=conditions,
    )
    numerics = Numerics(
        spatial=spatial,
        time_integrator=BackwardEuler() if implicit else ClassicalRungeKutta4(),
        linear_solver=linear_solver,
        nonlinear_solver=(
            Newton(
                linear_solver=SparseDirect(),
                jacobian="colored_finite_difference",
                absolute_tolerance=1e-12,
```

```
                relative_tolerance=1e-10,
            )
            if implicit
            else None
        ),
    )
    sim = Simulation(
        problem=problem,
        mesh=StructuredLineMesher(cells=cells).generate(line),
        numerics=numerics,
    )
    sim.validate().require_valid()
    return sim.run(start_time=0, end_time=end, time_step=dt)


def profile_figure(title, result, model, names, *, reference=None):
    theme.use_theme()
    fig, axes = plt.subplots(
        len(names),
        1,
        sharex=True,
        squeeze=False,
        figsize=(7.2, 1.25 * len(names) + 0.85),
        layout="constrained",
    )
    for ax, name in zip(axes[:, 0], names):
        ref = model.field(name)
        field = result.field(ref)
        x = field.space.dof_coordinates[:, 0]
        y = np.asarray(field.values).reshape(-1)
        assert x.shape == y.shape and np.isfinite(y).all()
        order = np.argsort(x)
        ax.plot(x[order], y[order], ".-", markersize=2, label="Computed")
        if reference is not None and name in reference:
            ax.plot(x[order], reference[name](x[order]), "--", label="Reference")
            ax.legend(frameon=False)
        ax.set_title(name.replace("_", " ").capitalize(), loc="left", fontsize=9)
        ax.set_ylabel(ref.unit)
        ax.grid(alpha=0.25)
    axes[-1, 0].set_xlabel("World x [m]")
    fig.suptitle(title, fontweight="bold")
    return fig
```

6.3 Shared cell for facade configurations

Use this cell for provider-selected configurations. The biophysics facades also accept a method-neutral problem with an explicit state. Validate first, then use the typed result fields shown in each recipe. The configurations in these examples assume the provider registry shipped with the inspected source.

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


def run_service(service, configuration, problem=None):
    args = (configuration,) if problem is None else (problem, configuration)
    issues = service.validate(*args)
    if issues:
        raise ValueError("\n".join(issues))
    return service.run(*args)


def service_figure(title, x, x_label, *series):
    theme.use_theme()
    fig, axes = plt.subplots(
        len(series),
        1,
        sharex=True,
        squeeze=False,
        figsize=(7.2, 1.3 * len(series) + 0.85),
        layout="constrained",
    )
    for ax, (label, values) in zip(axes[:, 0], series):
        y = np.asarray(values)
        assert y.ndim == 1 and y.shape == np.shape(x) and np.isfinite(y).all()
        ax.plot(x, y, ".-" if len(y) < 12 else "-", markersize=4)
        ax.set_ylabel(label)
        ax.grid(alpha=0.25)
    axes[-1, 0].set_xlabel(x_label)
    fig.suptitle(title, fontweight="bold")
    return fig
```

6.4 Acoustics recipes

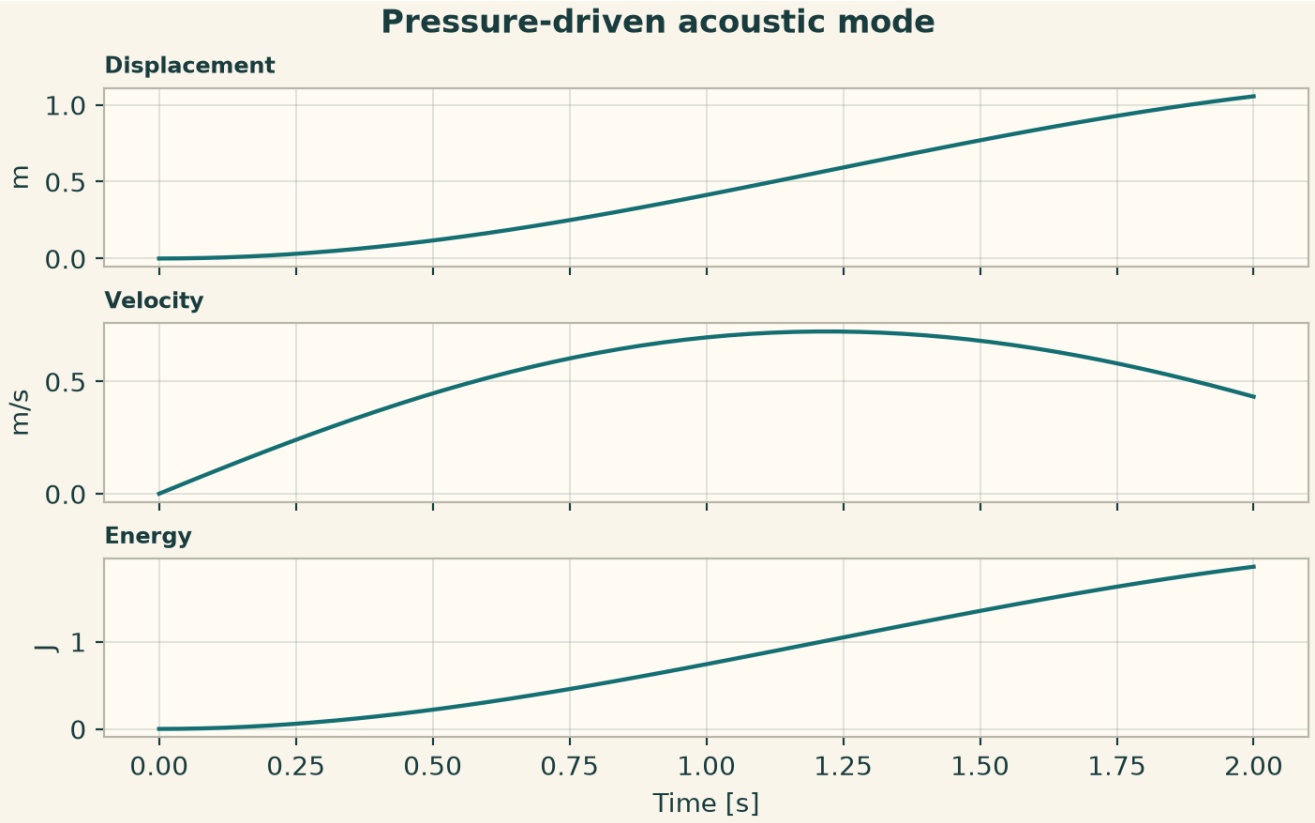
This section includes the field-based declarations and the independently callable acoustics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.4.1 Pressure-driven acoustic mode

Apply a constant pressure to a single effective area. Displacement and modal energy show the build-up from rest.

```
from onesim.physics.acoustics import PressureDrivenMode

domain = LumpedDomain()
model = PressureDrivenMode(
    domain=domain.system,
    mass=2,
    stiffness=3,
```

```
    area=0.5,
    damping=0.4,
    pressure_loading=4,
)
t, data = history_case(
    domain,
    model,
    dict(displacement=0.0, velocity=0.0),
    ("displacement", "velocity", "energy"),
    end=2,
    dt=0.01,
)
assert data["displacement"][-1] > 0
fig = history_plot("Pressure-driven acoustic mode", model, t, data)
plt.show()
```

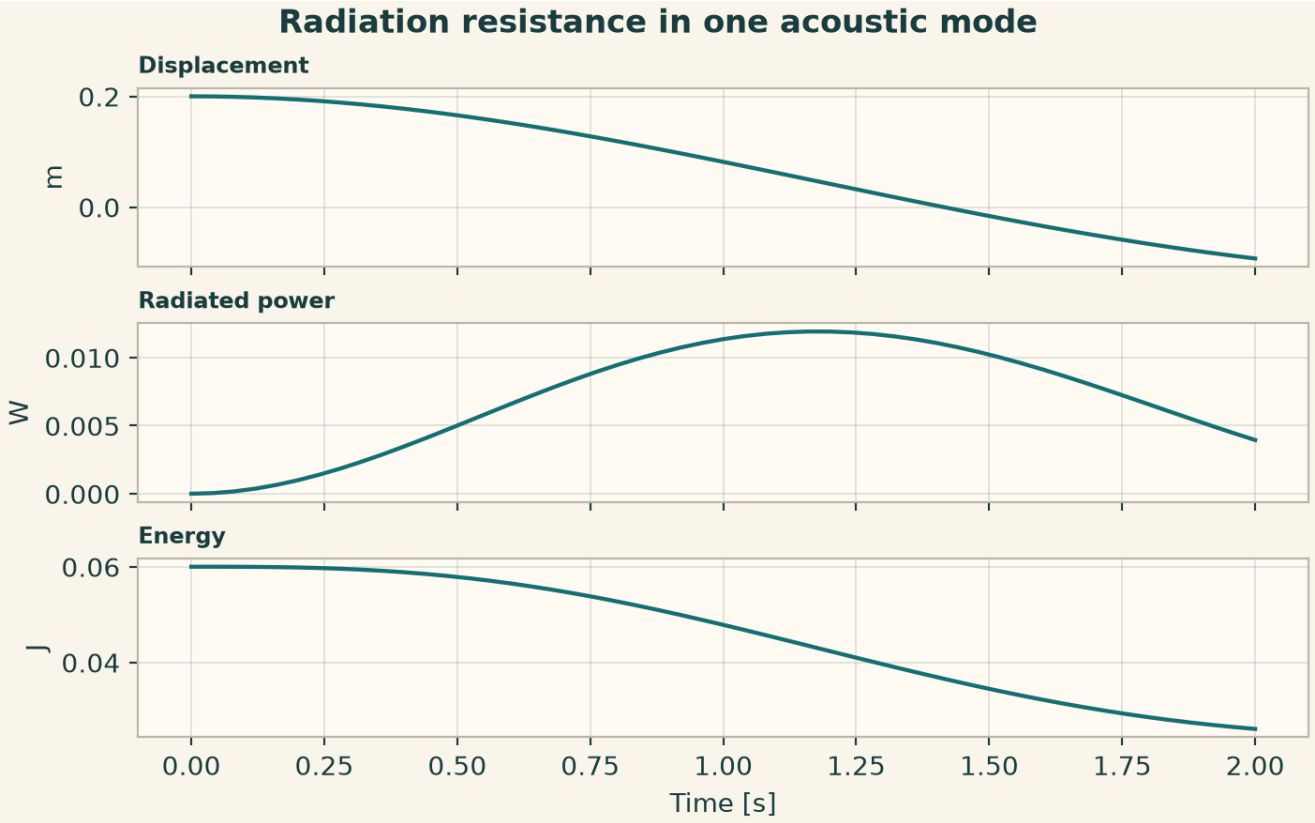


6.4.2 Radiation resistance in one acoustic mode

Release an initially displaced mode. The constant radiation resistance converts modal motion into nonnegative radiated power.

```
from onesim.physics.acoustics import RadiatingMode

domain = LumpedDomain()
model = RadiatingMode(
    domain=domain.system,
    mass=2,
    stiffness=3,
```

```
    area=0.5,
    damping=0.4,
    radiation_resistance=0.3,
)
t, data = history_case(
    domain,
    model,
    dict(displacement=0.2, velocity=0.0),
    ("displacement", "radiated_power", "energy"),
    end=2,
    dt=0.01,
)
assert data["radiated_power"].min() >= 0 and data["energy"][-1] < data["energy"][0]
fig = history_plot("Radiation resistance in one acoustic mode", model, t, data)
plt.show()
```

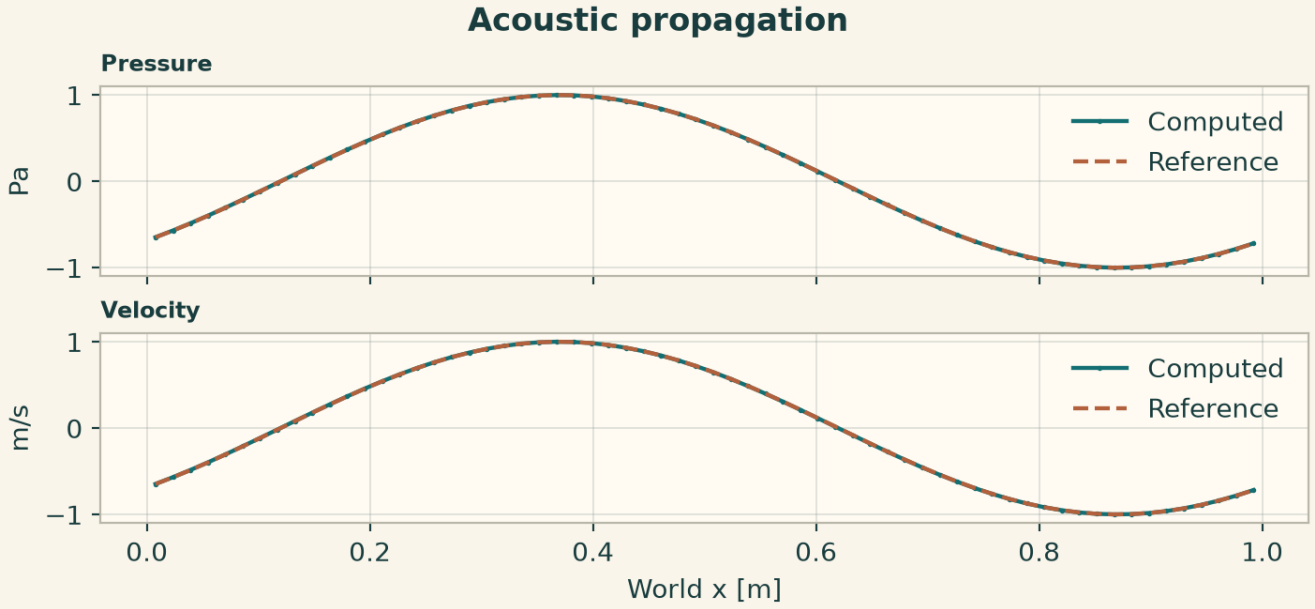


6.4.3 Propagation of a plane acoustic wave

Initialize pressure and velocity with the same right-moving characteristic. Their relative amplitude is set by acoustic impedance. A periodic domain isolates the phase error.

```
from onesim.physics.acoustics import LinearAcoustics

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = LinearAcoustics(domain=line.line, density=1, sound_speed=1)
wave = lambda x: np.sin(2 * np.pi * x)
end = 0.12
result = line_experiment(
```

```
    line,
    model,
    dict(pressure=wave, velocity=wave),
    FiniteDifference(gradient=CentralDifference()),
    cells=64,
    end=end,
    dt=0.001,
)
exact = lambda x: wave(x - end)
p = result.field(model.pressure)
np.testing.assert_allclose(p.values, exact(p.space.dof_coordinates[:, 0]), atol=0.002)
fig = profile_figure(
    "Acoustic propagation",
    result,
    model,
    ("pressure", "velocity"),
    reference={"pressure": exact, "velocity": exact},
)
plt.show()
```

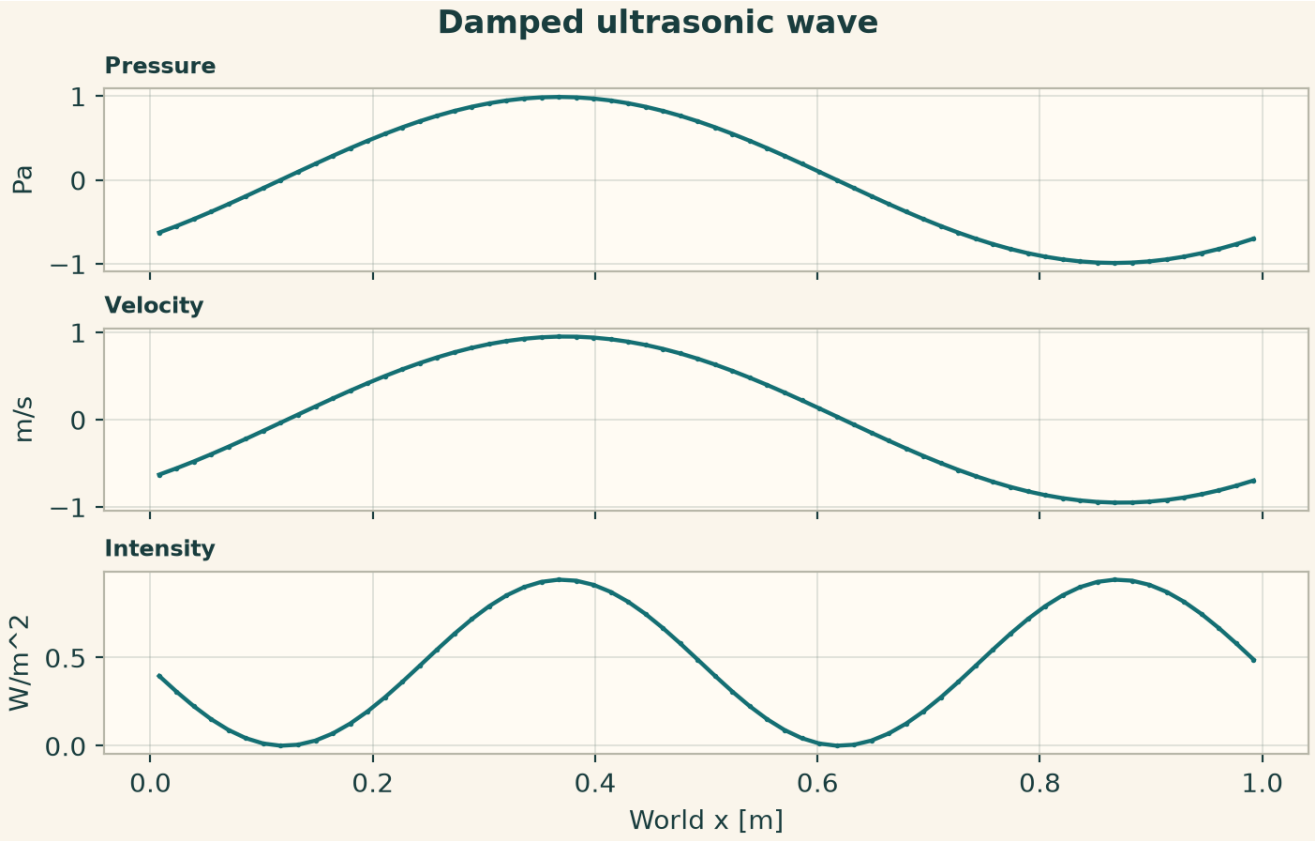


6.4.4 Damped ultrasonic wave and intensity

This declaration shares the pressure and velocity balance and exposes an intensity diagnostic. The signed intensity is pressure times particle velocity.

```
from onesim.physics.acoustics import UltrasonicAcoustics

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = UltrasonicAcoustics(
    domain=line.line, density=1, sound_speed=1, velocity_damping=0.5
)
wave = lambda x: np.sin(2 * np.pi * x)
result = line_experiment(
    line,
```

```
    model,
    dict(pressure=wave, velocity=wave),
    FiniteVolume(convection=Rusanov(), reconstruction=MUSCL()),
    cells=64,
    end=0.12,
    dt=0.001,
)
p = result.field(model.pressure).values
u = result.field(model.velocity).values
np.testing.assert_allclose(result.field(model.intensity).values, p * u)
fig = profile_figure(
    "Damped ultrasonic wave", result, model, ("pressure", "velocity", "intensity")
)
plt.show()
```

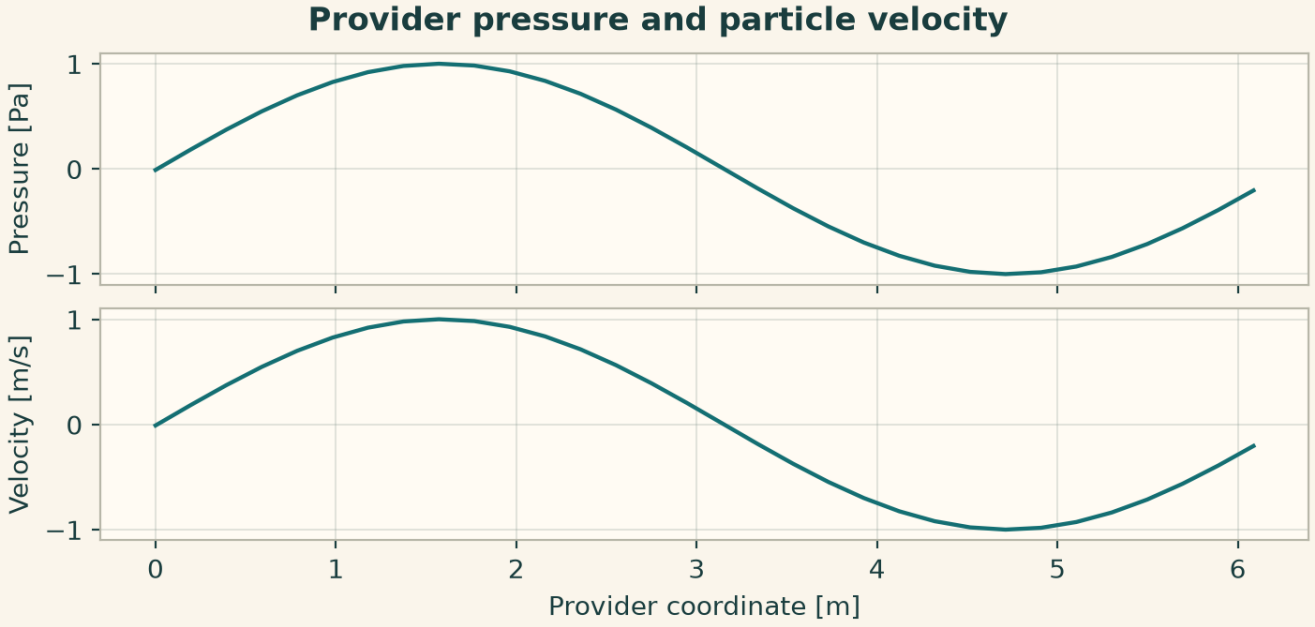


6.4.5 Provider pressure and particle velocity

Specify both initial arrays on the provider coordinate grid. The wave moves without coupling to a mechanical mode.

```
from onesim.physics.acoustics import AcousticsConfiguration, AcousticsPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = AcousticsConfiguration(
    parameters=dict(
```

```
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_pressure=wave,
        initial_velocity=wave,
        sound_speed=1,
    )
)
result = run_service(AcousticsPublicApi(), configuration)
fig = service_figure(
    "Provider pressure and particle velocity",
    result.coordinates,
    "Provider coordinate [m]",
    ("Pressure [Pa]", result.pressure),
    ("Velocity [m/s]", result.velocity),
)
plt.show()
```

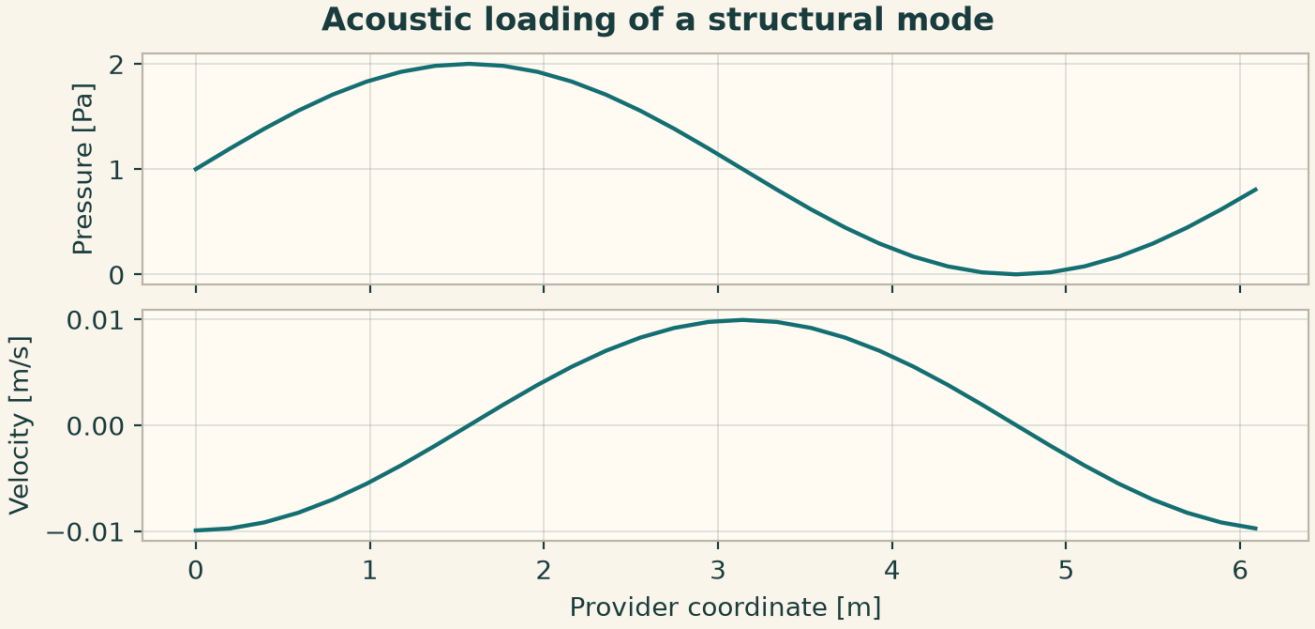


6.4.6 Acoustic loading of a structural mode

The structural service delegates the line wave calculation and advances a driven mode from its spatial pressure. Read the acoustic carrier through the nested result.

```
from onesim.physics.acoustics import (
    StructuralAcousticsConfiguration,
    StructuralAcousticsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = StructuralAcousticsConfiguration(
    parameters=dict(
```

```
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_pressure=1 + wave,
        initial_velocity=0.15,
        modal_mass=2,
        modal_stiffness=3,
        area=0.5,
    )
)
result = run_service(StructuralAcousticsPublicApi(), configuration)
fig = service_figure(
    "Acoustic loading of a structural mode",
    result.acoustics_result.coordinates,
    "Provider coordinate [m]",
    ("Pressure [Pa]", result.acoustics_result.pressure),
    ("Velocity [m/s]", result.acoustics_result.velocity),
)
plt.show()
```

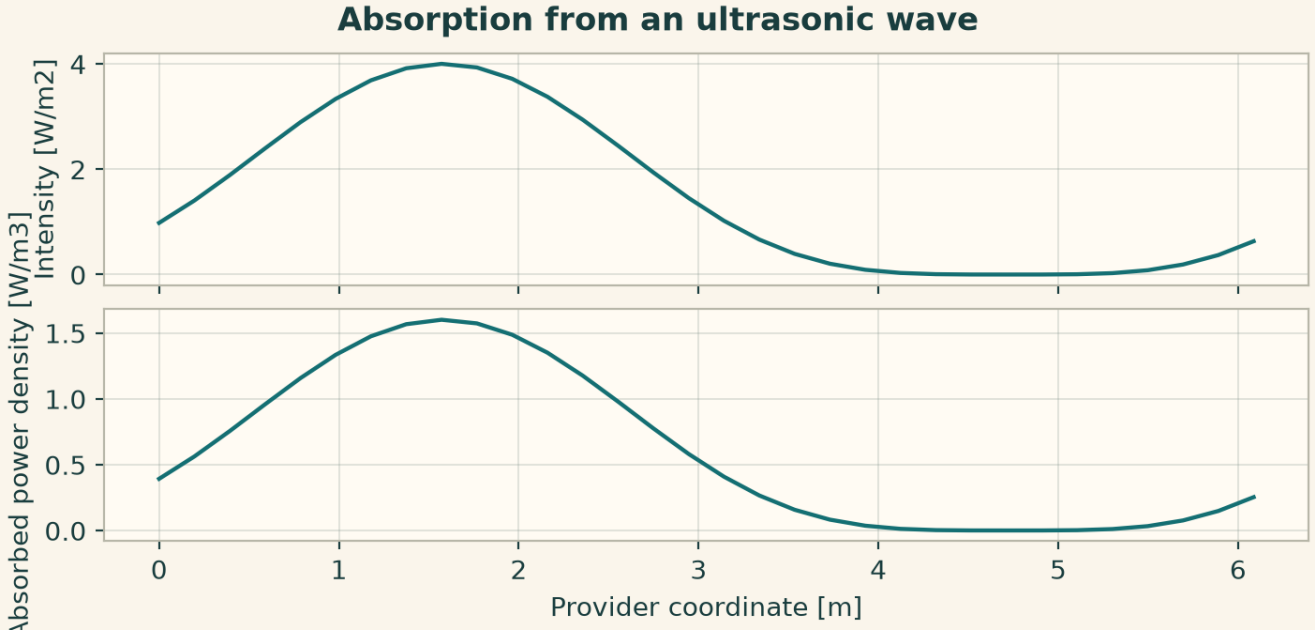


6.4.7 Absorption from an ultrasonic wave

Specify the absorption coefficient separately from the wave parameters. The facade returns absorbed power density on its acoustic coordinates.

```
from onesim.physics.acoustics import UltrasoundConfiguration, UltrasoundPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = UltrasoundConfiguration(
    parameters=dict(
        cell_count=n,
```

```
        final_time=0.01,
        time_step=0.0001,
        initial_pressure=1 + wave,
        initial_velocity=1 + wave,
        absorption_coefficient=0.4,
    )
)
result = run_service(UltrasoundPublicApi(), configuration)
fig = service_figure(
    "Absorption from an ultrasonic wave",
    result.coordinates,
    "Provider coordinate [m]",
    ("Intensity [W/m2]", result.intensity),
    ("Absorbed power density [W/m3]", result.absorbed_power_density),
)
plt.show()
```

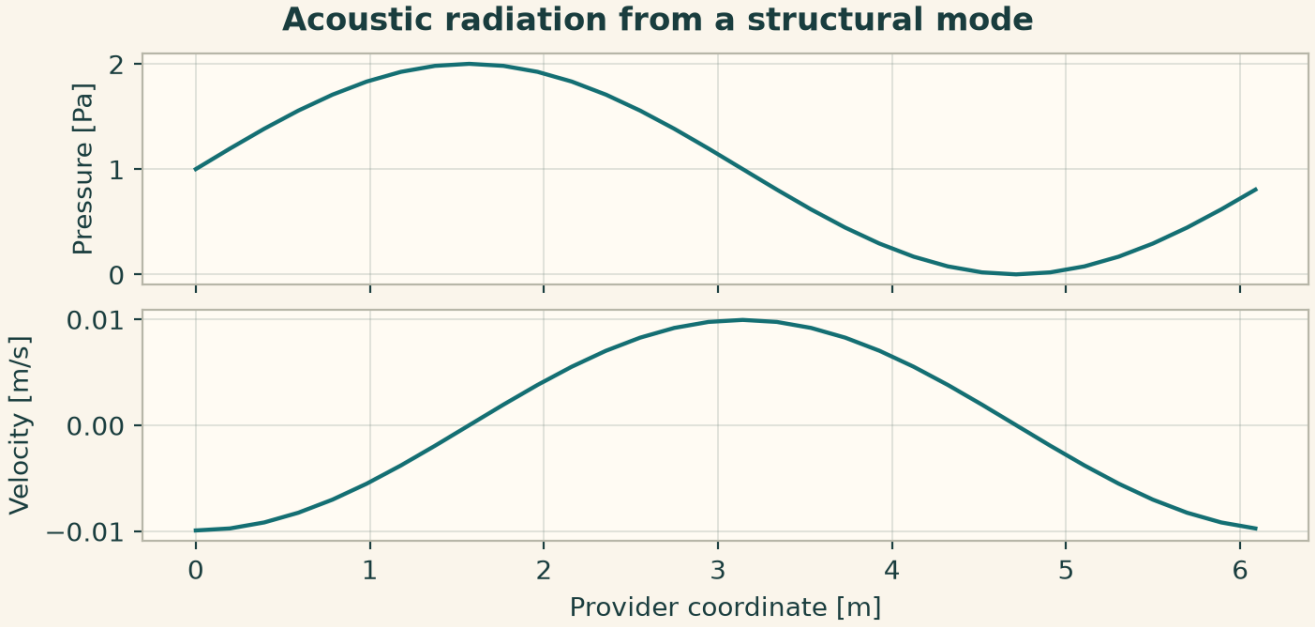


6.4.8 Acoustic radiation from a structural mode

Supply initial modal velocity and radiation resistance. The final radiated power is a mode diagnostic, and the nested acoustic result retains the carrier.

```
from onesim.physics.acoustics import (
    VibroacousticsConfiguration,
    VibroacousticsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = VibroacousticsConfiguration(
    parameters=dict(
        cell_count=n,
```

```
        final_time=0.01,
        time_step=0.0001,
        initial_pressure=1 + wave,
        initial_velocity=0.15,
        radiation_resistance=0.4,
    )
)
result = run_service(VibroacousticsPublicApi(), configuration)
fig = service_figure(
    "Acoustic radiation from a structural mode",
    result.structural_acoustics.acoustics_result.coordinates,
    "Provider coordinate [m]",
    ("Pressure [Pa]", result.structural_acoustics.acoustics_result.pressure),
    ("Velocity [m/s]", result.structural_acoustics.acoustics_result.velocity),
)
plt.show()
```

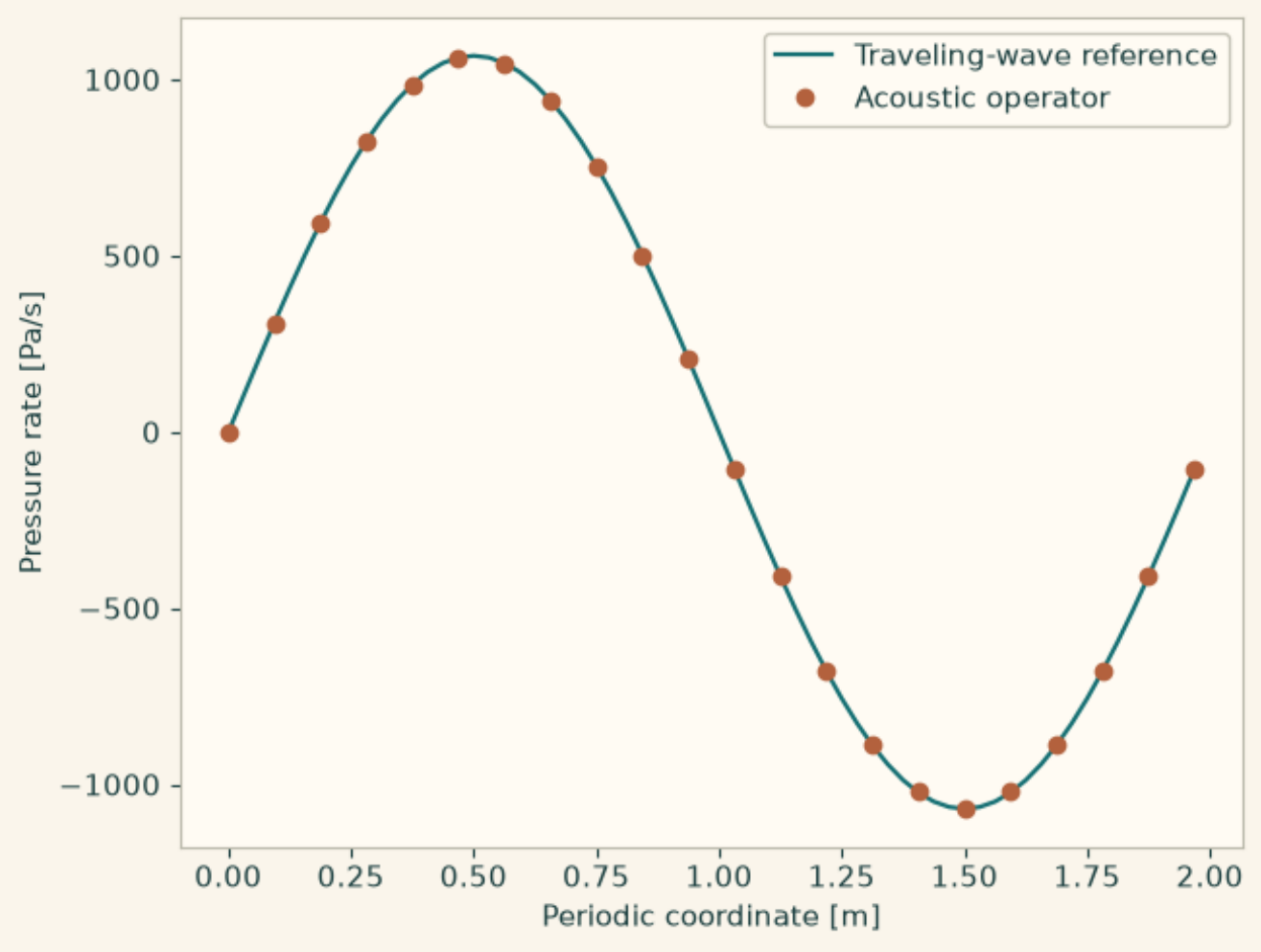


6.4.9 Evaluate the low-level acoustic rate

Bind an acoustic provider directly to its spatial operator. Compare the pressure and velocity rates for a periodic traveling wave with the analytical and centered-difference references.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
theme.use_theme()

from onesim.numerics.discretization.equations.contracts.operators.operator_contracts
    import SpatialExecutionContext
from onesim.physics.acoustics import AcousticsProvider, AcousticsSpatialOperator

density = 1.2  # kg/m³
speed = 340.0  # m/s
```

```
count = 64
length = 2.0  # m
provider = AcousticsProvider(cell_count=count, density=density, sound_speed=speed,
                             bounds=(0.0, length), periodic=True)
operator = AcousticsSpatialOperator(provider, kernel_choice="numpy")
x, = operator.coordinate_axes
wave_number = 2*np.pi/length
pressure = np.cos(wave_number*x)
velocity = pressure/(density*speed)
state = np.stack((pressure, velocity))
context = SpatialExecutionContext("periodic-line", "pressure-velocity", 0.0)
rate = np.asarray(operator.residual(state, context).values)
exact_pressure_rate = speed * wave_number * np.sin(wave_number*x)
discrete_factor = np.sinc(wave_number * operator.spacings[0] / np.pi)
np.testing.assert_allclose(rate[0], discrete_factor * exact_pressure_rate, atol=1e-10)
assert np.max(np.abs(rate[0] - exact_pressure_rate)) < 0.002 *
    np.max(np.abs(exact_pressure_rate))
np.testing.assert_allclose(rate[1], rate[0]/(density*speed), atol=1e-12)
fields = operator.reconstruction(state, context)
np.testing.assert_allclose(fields["pressure"], pressure)
np.testing.assert_allclose(fields["velocity"], velocity)
fig, ax = plt.subplots(layout="constrained")
ax.plot(x, exact_pressure_rate, label="Traveling-wave reference")
ax.plot(x[::3], rate[0, ::3], "o", label="Acoustic operator")
ax.set(xlabel="Periodic coordinate [m]", ylabel="Pressure rate [Pa/s]")
ax.legend()
plt.show()
```



6.5 Biophysics recipes

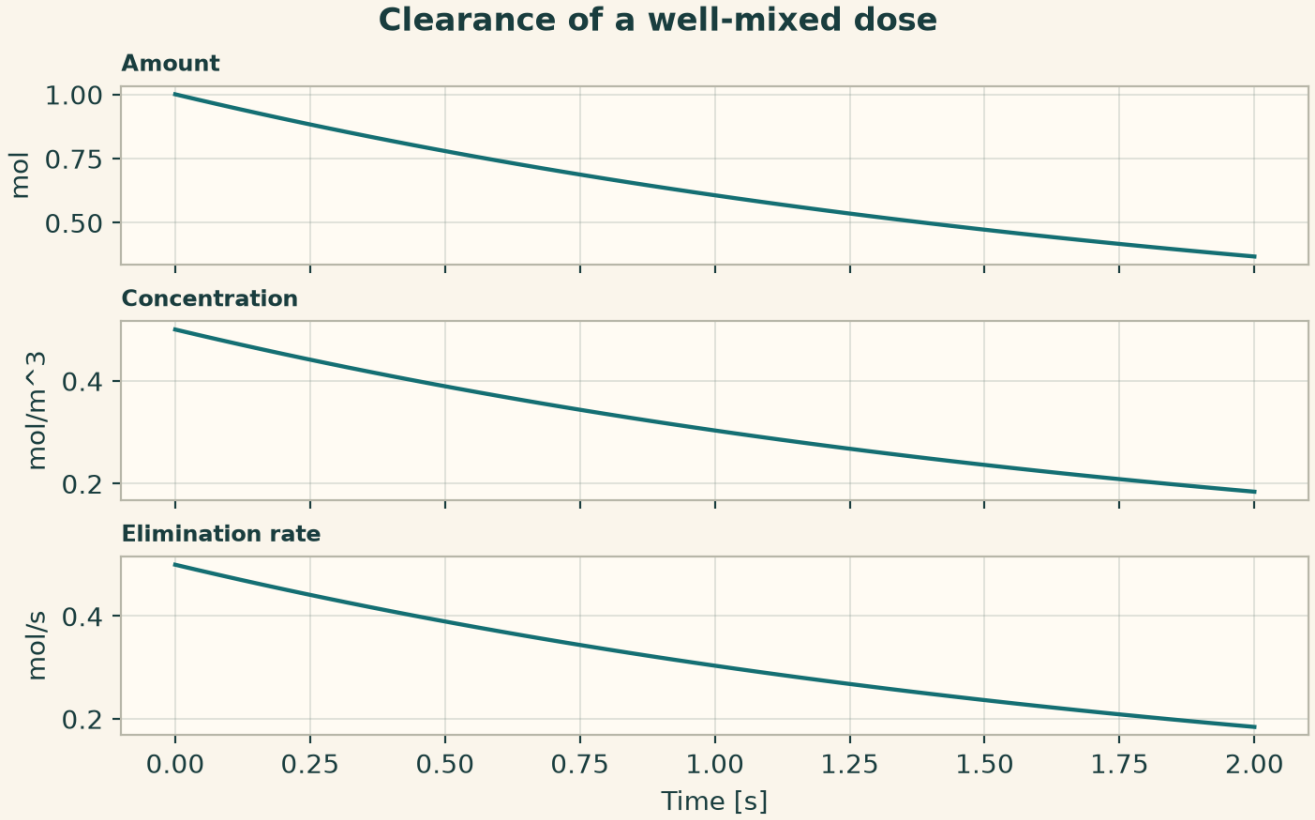
This section includes the field-based declarations and the independently callable biophysics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.5.1 Clearance of a well-mixed dose

The initial amount is divided by the compartment volume to obtain concentration. With zero infusion, the amount decays at clearance divided by volume.

```
from onesim.physics.biophysics import OrganClearance

domain = LumpedDomain()
model = OrganClearance(
    domain=domain.system,
    volume=2,
    clearance=1,
)
t, data = history_case(
    domain,
    model,
    dict(amount=1.0),
    ("amount", "concentration", "elimination_rate"),
    end=2,
```

```
    dt=0.01,
)
np.testing.assert_allclose(data["amount"], np.exp(-t / 2), atol=1e-9)
fig = history_plot("Clearance of a well-mixed dose", model, t, data)
plt.show()
```

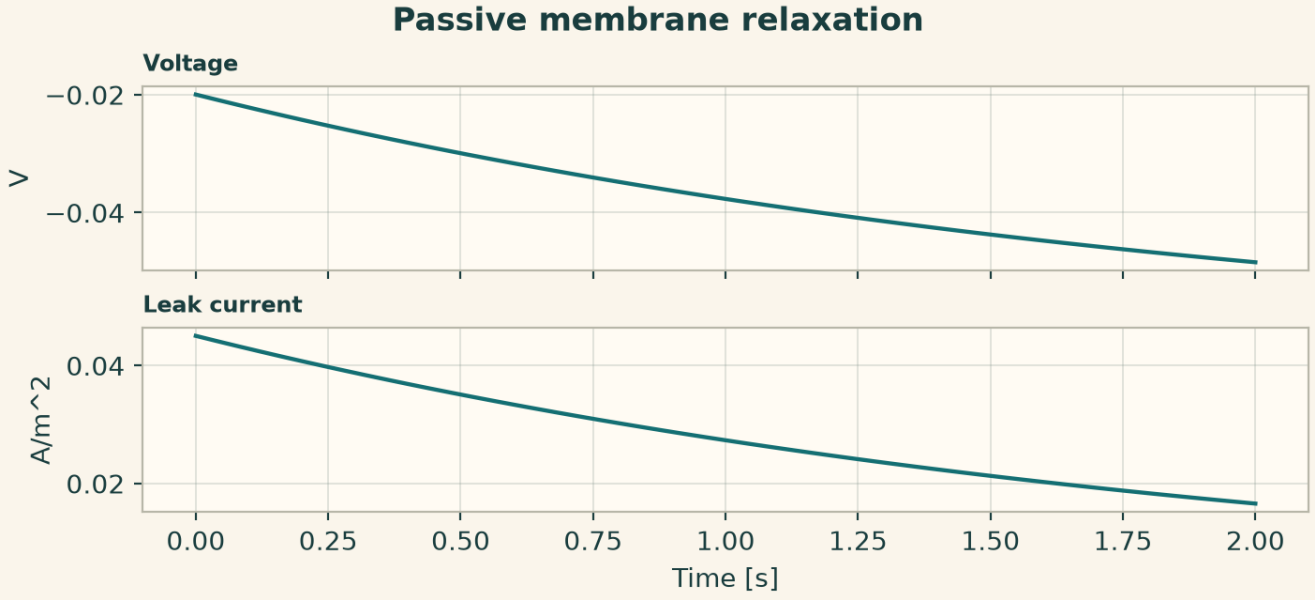


6.5.2 Passive membrane relaxation

A charged membrane relaxes toward its reversal potential. The voltage and signed leak current expose the capacitive and conductive parts of the balance.

```
from onesim.physics.biophysics import PassiveMembrane

domain = LumpedDomain()
model = PassiveMembrane(
    domain=domain.system,
    capacitance=2,
    conductance=1,
    reversal_potential=-0.065,
)
t, data = history_case(
    domain,
    model,
    dict(voltage=-0.02),
    ("voltage", "leak_current"),
    end=2,
    dt=0.01,
)
```

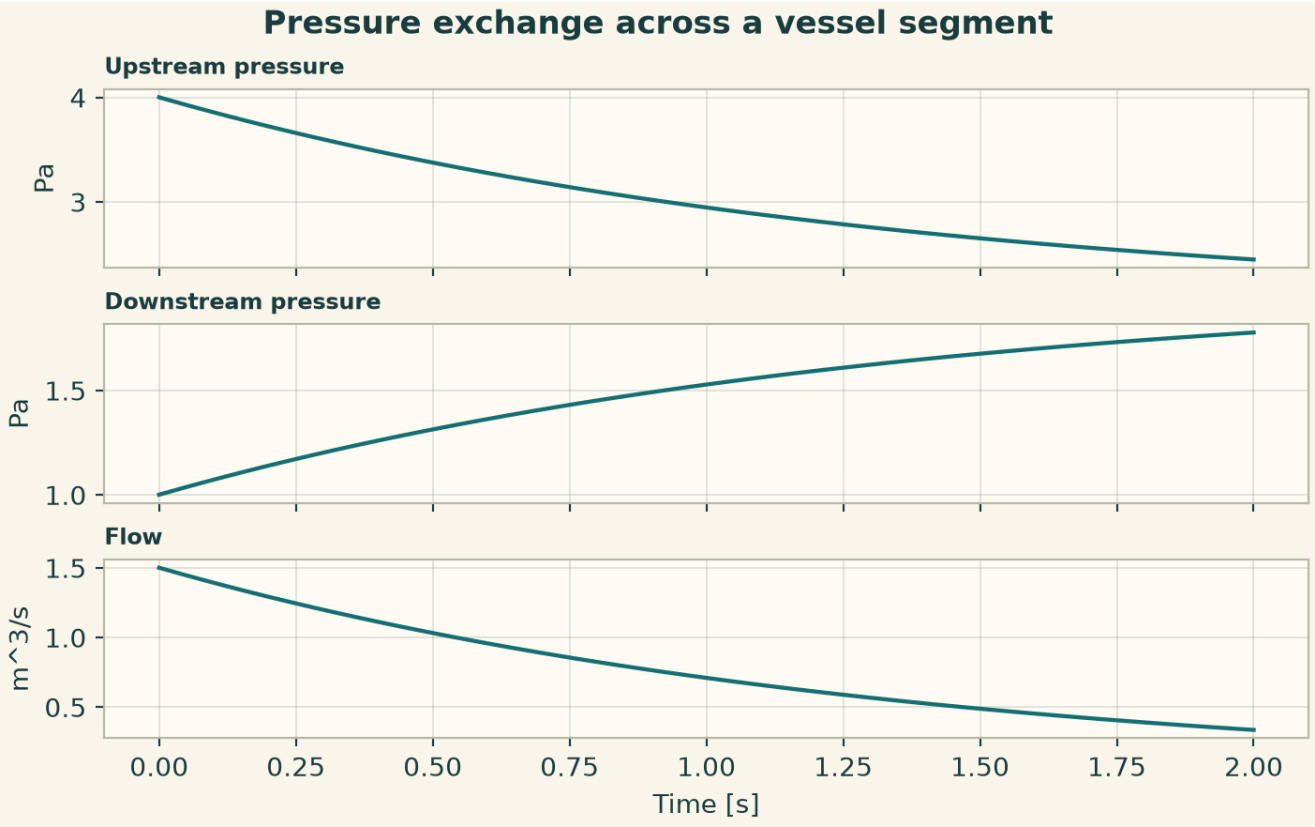
```
np.testing.assert_allclose(
    data["voltage"], -0.065 + 0.045 * np.exp(-t / 2), atol=1e-9
)
fig = history_plot("Passive membrane relaxation", model, t, data)
plt.show()
```



6.5.3 Pressure exchange across a vessel segment

Two compliances exchange fluid through a resistive segment. Their compliance-weighted pressure sum is constant without external flow.

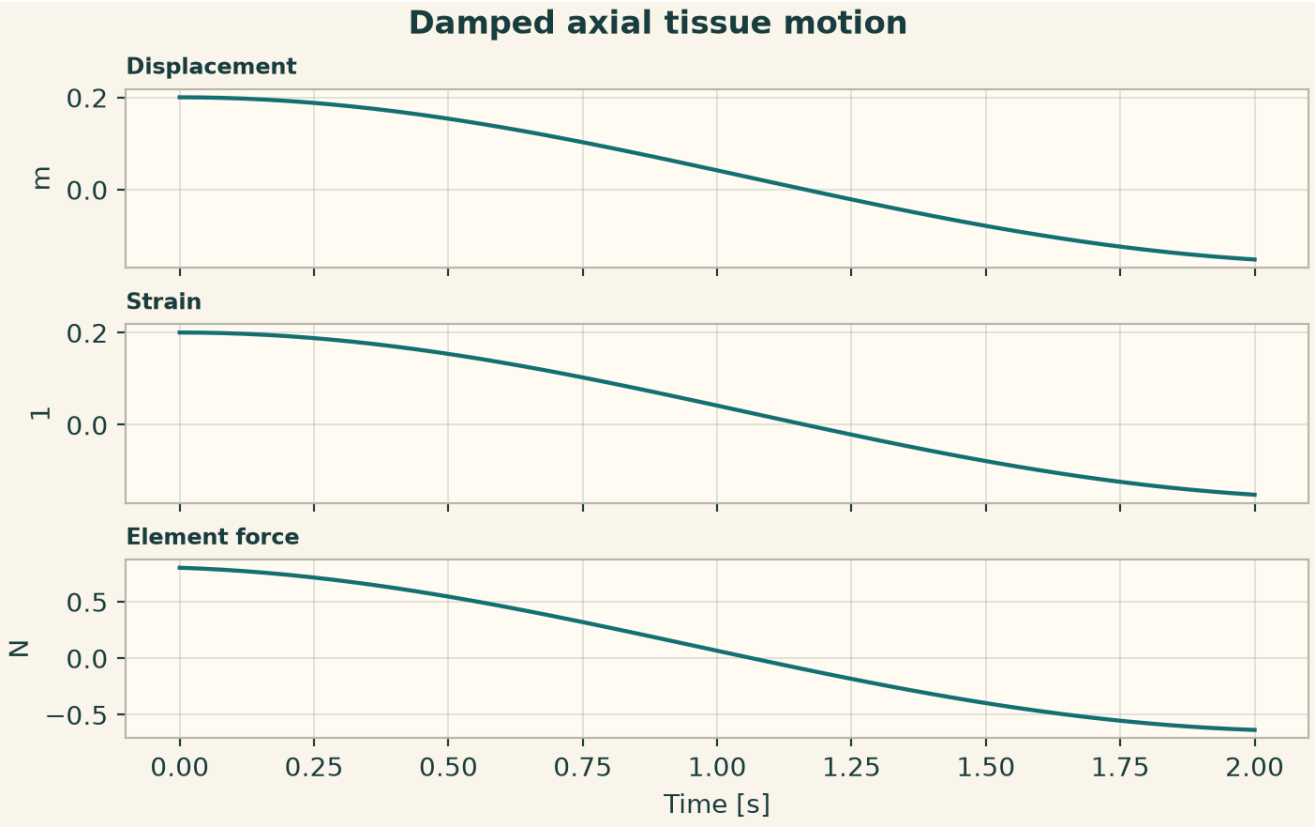
```
from onesim.physics.biophysics import CompliantVessels, VesselSegment

domain = LumpedDomain()
model = CompliantVessels(
    domain=domain.system,
    upstream_compliance=1,
    downstream_compliance=2,
    vessel=VesselSegment("bridge", "up", "down", 2),
)
t, data = history_case(
    domain,
    model,
    dict(upstream_pressure=4.0, downstream_pressure=1.0),
    ("upstream_pressure", "downstream_pressure", "flow"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(
    data["upstream_pressure"] + 2 * data["downstream_pressure"], 6.0
)
fig = history_plot("Pressure exchange across a vessel segment", model, t, data)
plt.show()
```



6.5.4 Damped axial tissue motion

An axial element stores and dissipates mechanical work. The mass displacement is the input to the tissue element strain and force calculation.

```
from onesim.physics.biophysics import AxialTissue, TissueElement

domain = LumpedDomain()
model = AxialTissue(
    domain=domain.system,
    mass=2,
    element=TissueElement("fiber", 0, 1, stiffness=4, rest_length=1, damping=0.4),
)
t, data = history_case(
    domain,
    model,
    dict(displacement=0.2, velocity=0.0),
    ("displacement", "strain", "element_force"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["strain"], data["displacement"], atol=1e-12)
fig = history_plot("Damped axial tissue motion", model, t, data)
plt.show()
```

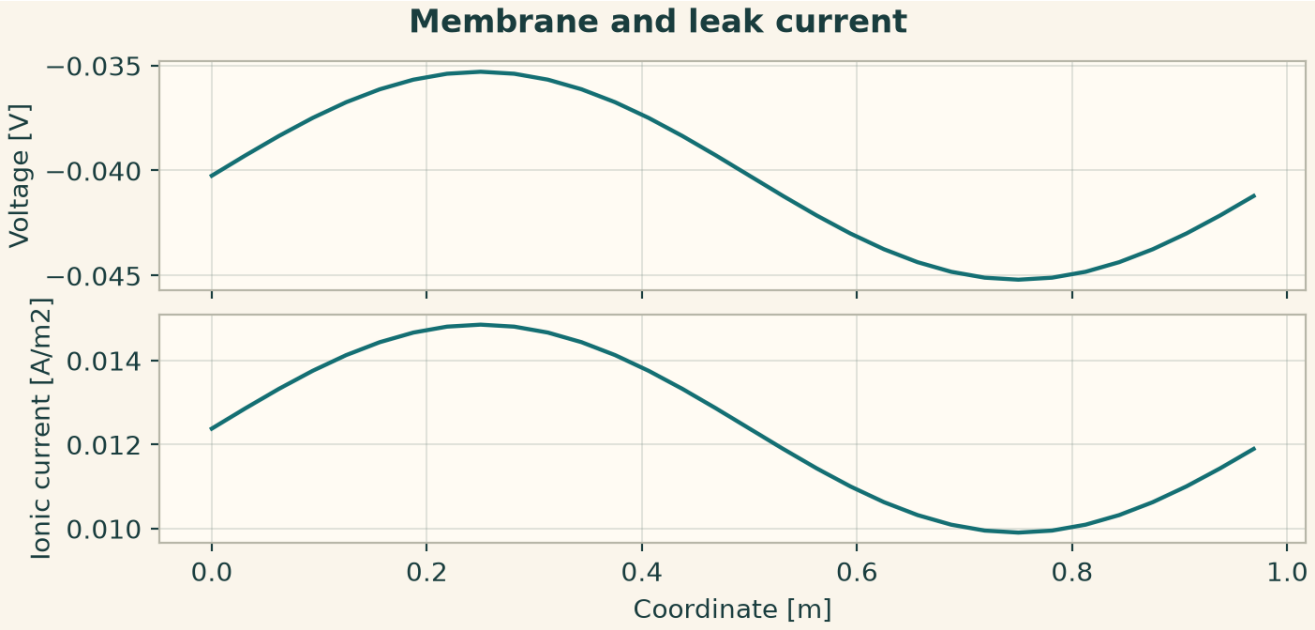


6.5.5 Bioelectrical provider with an explicit membrane state

The facade consumes a method-neutral initial-value problem and a separate numerical configuration. The state and ion list belong to the biophysical provider contract.

```
from onesim.physics.biophysics import (
    BioelectricalConfiguration,
    BioelectricalPublicApi,
)
from onesim.physics.biophysics.bioelectrical.models.bioelectrical_systems.state import
    (
    BioelectricalState,
    IonSpecies,
)
from onesim.simulation.problem_definition.declarations.physics.models.problem import (
    InitialValueProblem,
)

x = np.linspace(0, 1, 32, endpoint=False)
problem = InitialValueProblem(
    "Membrane with a leak",
    "bioelectrical",
    1,
    BioelectricalState(voltage=-0.04 + 0.005 * np.sin(2 * np.pi * x)),
    boundary_conditions=("periodic",),
)
configuration = BioelectricalConfiguration(
    parameters=dict(
        cell_count=len(x),
```

```
        species=(IonSpecies("leak", 0.5, -0.065),),
        diffusion=0.01,
        final_time=0.02,
        time_step=0.0005,
    )
)
result = run_service(BioelectricalPublicApi(), configuration, problem)
fig = service_figure(
    "Membrane and leak current",
    x,
    "Coordinate [m]",
    ("Voltage [V]", result.state.voltage),
    ("Ionic current [A/m2]", result.ionic_currents["leak"]),
)
plt.show()
```

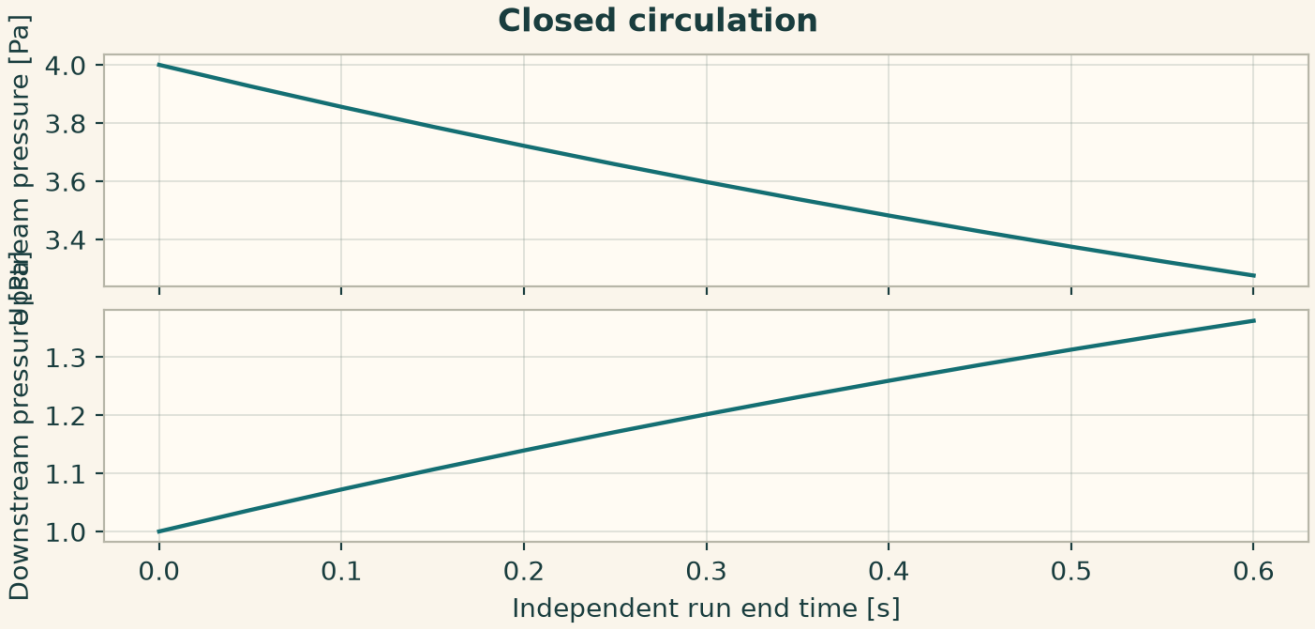


6.5.6 Circulatory service with compartment pressures

Declare the network geometry as named compartment IDs and segment endpoints. The problem supplies the initial pressures and the configuration supplies compliances and resistance.

```
from onesim.physics.biophysics import (
    CirculatoryConfiguration,
    CirculatoryPublicApi,
    VesselSegment,
)
from onesim.physics.biophysics.circulation.models.circulatory_systems.state import (
    CirculatoryState,
)
from onesim.simulation.problem_definition.declarations.physics.models.problem import (
    InitialValueProblem,
)
```

```
problem = InitialValueProblem(
    "Closed circulation",
    "circulation",
    1,
    CirculatoryState({"up": 4.0, "down": 1.0}),
    boundary_conditions=("closed",),
)
t = np.linspace(0, 0.6, 13)
results = [
    run_service(
        CirculatoryPublicApi(),
        CirculatoryConfiguration(
            parameters=dict(
                compliances={"up": 1.0, "down": 2.0},
                segments=(VesselSegment("bridge", "up", "down", 2.0),),
                final_time=float(end),
                time_step=0.005,
            )
        ),
        problem,
    )
    for end in t
]
up = np.array([r.state.pressures["up"] for r in results])
down = np.array([r.state.pressures["down"] for r in results])
np.testing.assert_allclose(up + 2 * down, 6.0)
fig = service_figure(
    "Closed circulation",
    t,
    "Independent run end time [s]",
    ("Upstream pressure [Pa]", up),
    ("Downstream pressure [Pa]", down),
)
plt.show()
```

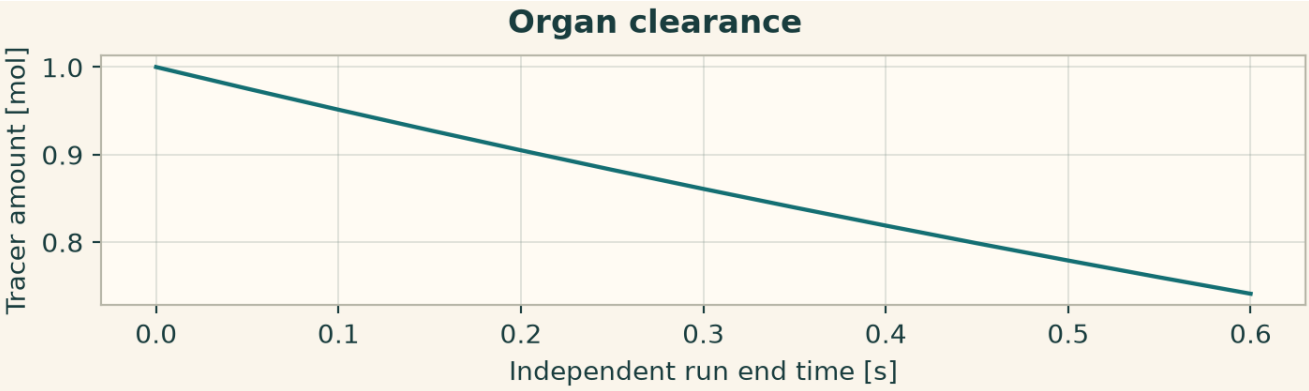


6.5.7 Organ service with a species-specific clearance

The state indexes amounts by organ and species. Clearances are keyed by the same identifiers, and the result keeps both local concentrations and global amounts.

```
from onesim.physics.biophysics import OrganConfiguration, OrganPublicApi
from onesim.physics.biophysics.organs.models.organ_models.state import OrganState
from onesim.simulation.problem_definition.declarations.physics.models.problem import (
    InitialValueProblem,
)

problem = InitialValueProblem(
    "Clear a dose",
    "organ",
    1,
    OrganState(("body",), ("tracer",), np.array([[1.0]])),
    boundary_conditions=("closed",),
)
t = np.linspace(0, 0.6, 13)
results = [
    run_service(
        OrganPublicApi(),
        OrganConfiguration(
            parameters=dict(
                volumes={"body": 2.0},
                clearances={("body", "tracer"): 1.0},
                final_time=float(end),
                time_step=0.005,
            )
        ),
        problem,
    )
    for end in t
]
```

```
amount = np.array([r.total_amounts["tracer"] for r in results])
np.testing.assert_allclose(amount, np.exp(-t / 2), atol=1e-8)
fig = service_figure(
    "Organ clearance",
    t,
    "Independent run end time [s]",
    ("Tracer amount [mol]", amount),
)
plt.show()
```

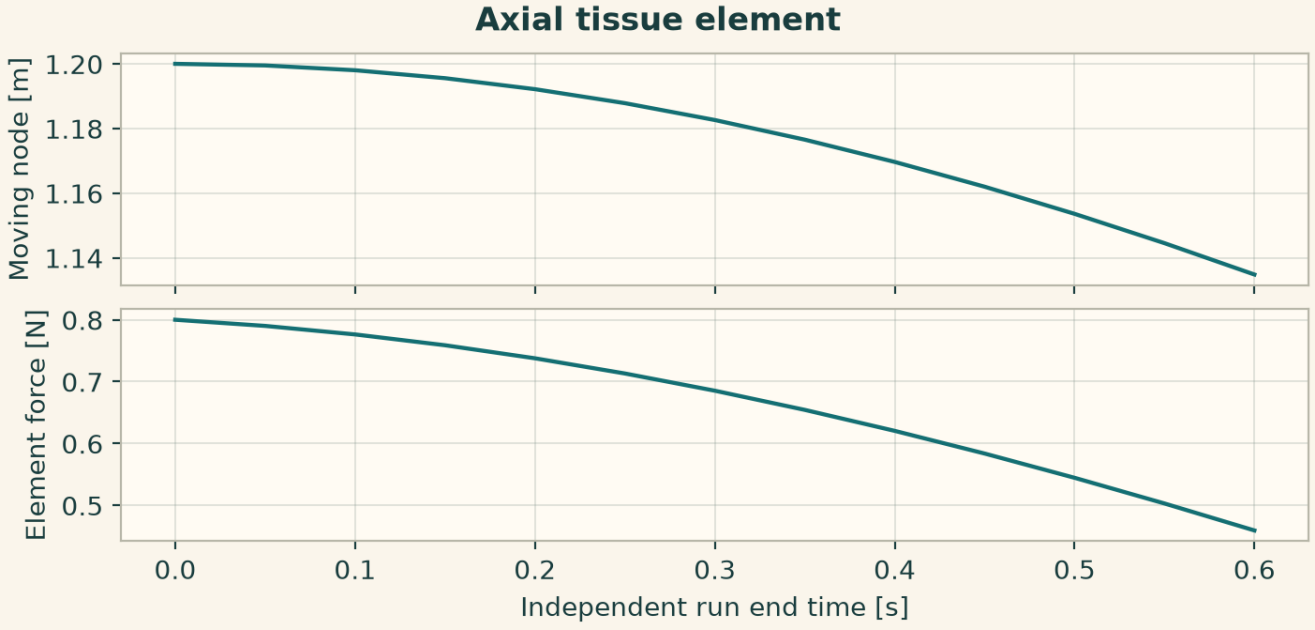


6.5.8 Tissue service with a fixed node and an axial element

Keep one node fixed and release a stretched element. Positions and velocities are node state. Force is an element diagnostic.

```
from onesim.physics.biophysics import (
    TissueConfiguration,
    TissuePublicApi,
    TissueElement,
)
from onesim.physics.biophysics.tissue.models.tissue_mechanics.state import TissueState
from onesim.simulation.problem_definition.declarations.physics.models.problem import (
    InitialValueProblem,
)

problem = InitialValueProblem(
    "Axial tissue",
    "tissue",
    1,
    TissueState(np.array([0.0, 1.2]), np.zeros(2)),
    boundary_conditions=("fixed",),
)
t = np.linspace(0, 0.6, 13)
results = [
    run_service(
        TissuePublicApi(),
        TissueConfiguration(
            parameters=dict(
                masses=(1.0, 2.0),
```

```
                elements=(
                    TissueElement(
                        "fiber", 0, 1, stiffness=4, rest_length=1, damping=0.4
                    ),
                ),
                fixed_nodes=(0,),
                final_time=float(end),
                time_step=0.005,
            )
        ),
        problem,
    )
    for end in t
]
fig = service_figure(
    "Axial tissue element",
    t,
    "Independent run end time [s]",
    ("Moving node [m]", [r.state.positions[1] for r in results]),
    ("Element force [N]", [r.element_forces["fiber"] for r in results]),
)
plt.show()
```

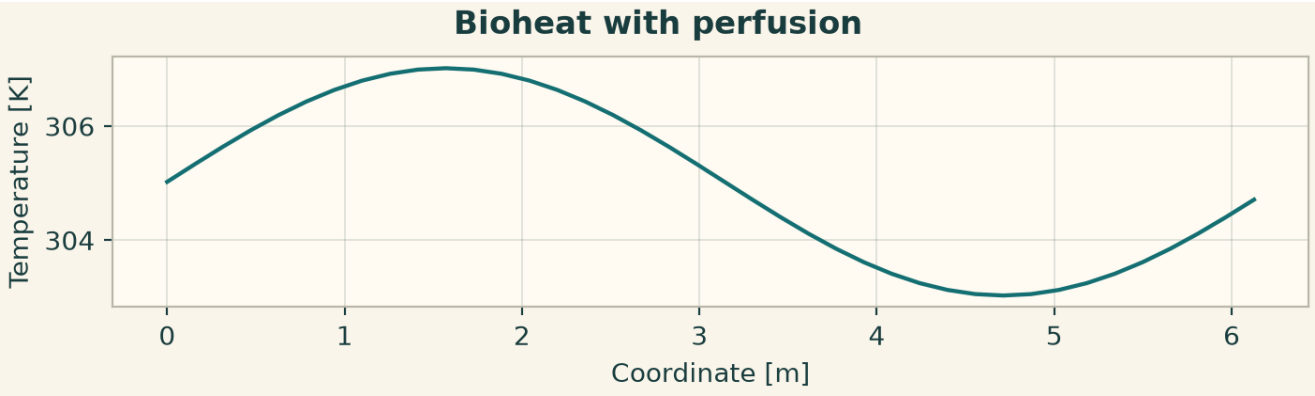


6.5.9 Bioheat provider with perfusion and an external source

Set the conductive and perfusion coefficients explicitly. The returned thermal profile is evaluated on the provider's line carrier.

```
from onesim.physics.biophysics import BioheatProvider

n = 40
x = np.linspace(0, 1, n, endpoint=False)
provider = BioheatProvider(
```

```
    cell_count=n,
    perfusion_capacity_rate=3e4,
    blood_temperature=310.0,
    external_heat=2e4,
    initial_temperature=305.0 + 2 * np.sin(2 * np.pi * x),
    final_time=0.4,
    time_step=0.005,
)
result = provider.execute()
fig = service_figure(
    "Bioheat with perfusion",
    result.coordinates,
    "Coordinate [m]",
    ("Temperature [K]", result.temperature),
)
plt.show()
```



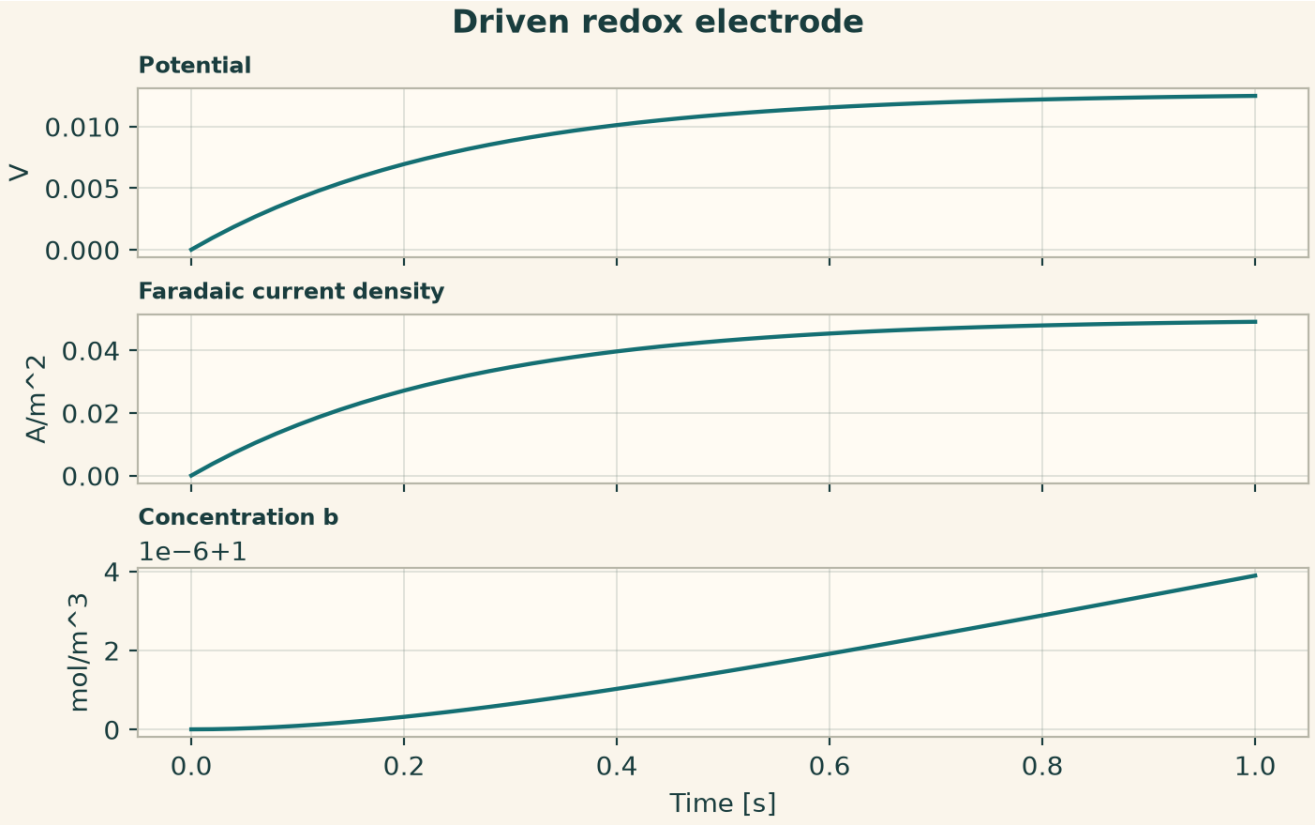
6.6 Chemistry recipes

6.6.1 Driven redox electrode

Use explicit stoichiometry, an equilibrium-potential closure, and a current density. Track the overpotential through the electrode potential and the transfer of material between species.

```
from onesim.physics.chemistry import LumpedElectrode, ElectrochemicalReaction

domain = LumpedDomain()
model = LumpedElectrode(
    domain=domain.system,
    reactions=(
        ElectrochemicalReaction("redox", {"A": -1, "B": 1}, 0.1, lambda c, T: 0.0),
    ),
    area=0.01,
    volume=0.001,
    double_layer_capacitance=1,
    applied_current_density=0.05,
)
t, data = history_case(
    domain,
```

```
    model,
    dict(potential=0.0, concentration_A=1.0, concentration_B=1.0),
    ("potential", "faradaic_current_density", "concentration_B"),
    end=1,
    dt=0.005,
)
assert data["concentration_B"][-1] > data["concentration_B"][0]
fig = history_plot("Driven redox electrode", model, t, data)
plt.show()
```

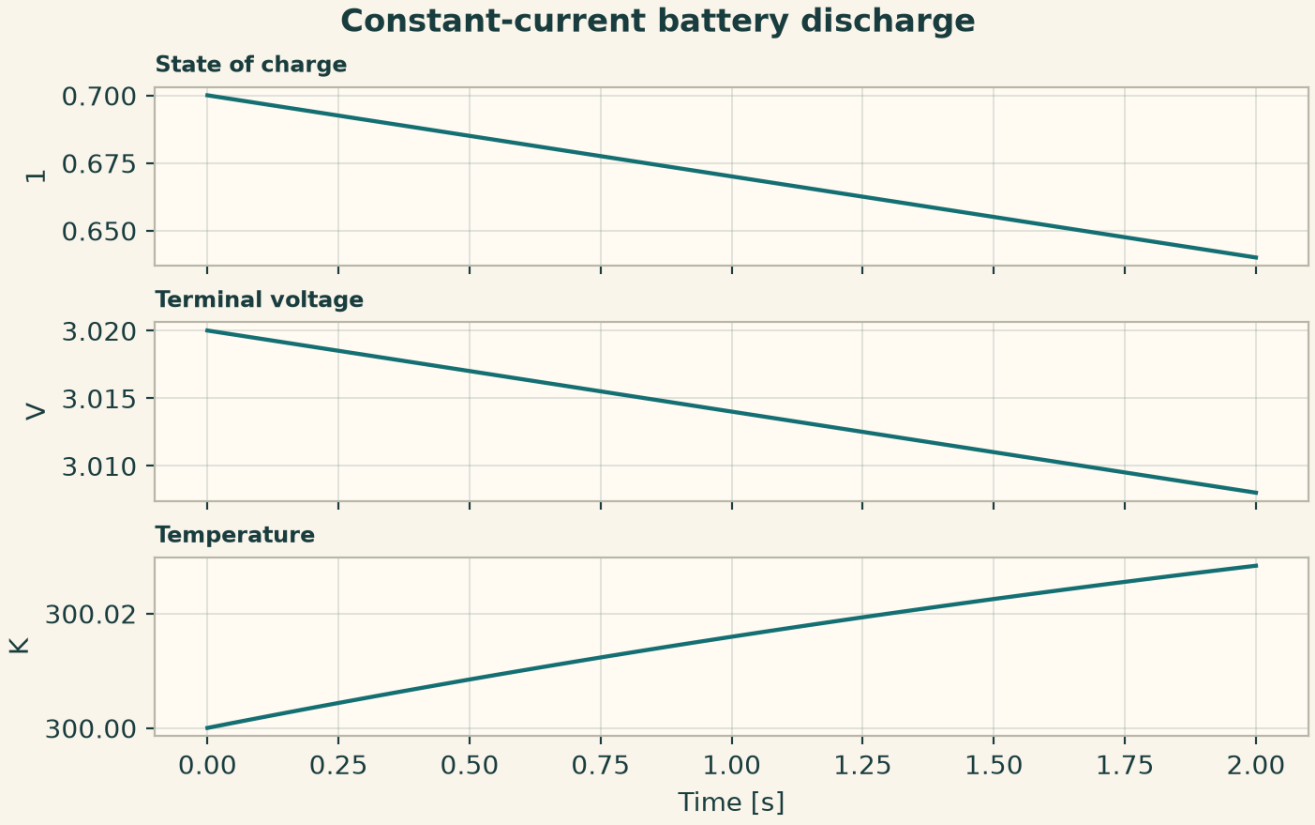


6.6.2 Constant-current battery discharge

Use a state-dependent open-circuit voltage and constant internal resistance. A positive current removes charge and generates ohmic heat.

```
from onesim.physics.chemistry import LumpedBattery, BatteryChemistry

domain = LumpedDomain()
model = LumpedBattery(
    domain=domain.system,
    chemistry=BatteryChemistry("cell", lambda s, T: 3 + 0.2 * s, lambda s, T: 0.4),
    charge_capacity=10,
    current=0.3,
    thermal_capacity=2,
    heat_transfer=0.5,
    ambient_temperature=300,
)
t, data = history_case(
```

```
    domain,
    model,
    dict(state_of_charge=0.7, temperature=300.0),
    ("state_of_charge", "terminal_voltage", "temperature"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["state_of_charge"], 0.7 - 0.03 * t, atol=1e-9)
fig = history_plot("Constant-current battery discharge", model, t, data)
plt.show()
```

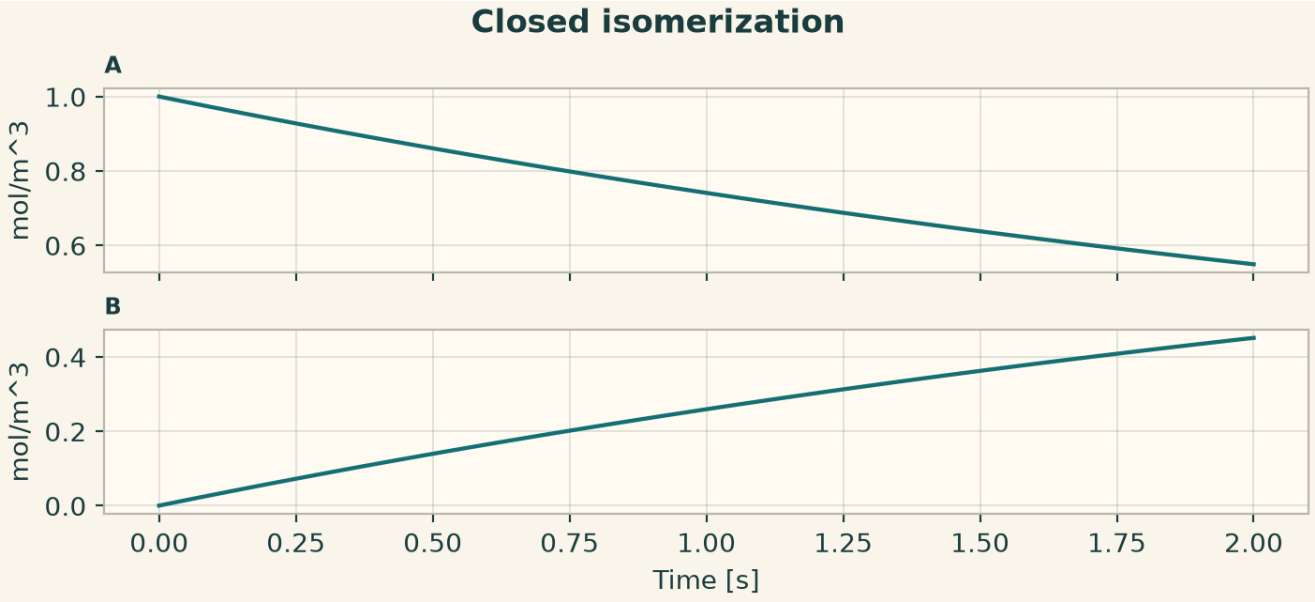


6.6.3 Closed isomerization reaction

Declare atom-balanced species and one reaction. The constant rate coefficient multiplies the reactant number density. Number density moves between the two species while their sum stays fixed.

```
from onesim.physics.chemistry import (
    ReactionIR,
    ReactionKinetics,
    ReactionNetworkIR,
    SpeciesIR,
)

domain = LumpedDomain()
species = tuple(
    SpeciesIR(n, 0, 1.66054e-27, composition=(("H", 1),)) for n in ("A", "B")
)
```

```
reaction = ReactionIR(
    "conversion", (("A", 1),), (("B", 1),), {"kind": "constant", "value": 0.3}
)
network = ReactionNetworkIR("isomers", "1", species, (reaction,), strict_balance=True)
model = ReactionKinetics(domain=domain.system, network=network)
a, b = model.concentration("A").name, model.concentration("B").name
t, data = history_case(domain, model, {a: 1.0, b: 0.0}, (a, b), end=2, dt=0.01)
np.testing.assert_allclose(data[a], np.exp(-0.3 * t), atol=1e-9)
np.testing.assert_allclose(data[a] + data[b], 1, atol=1e-12)
fig = history_plot("Closed isomerization", model, t, data)
plt.show()
```



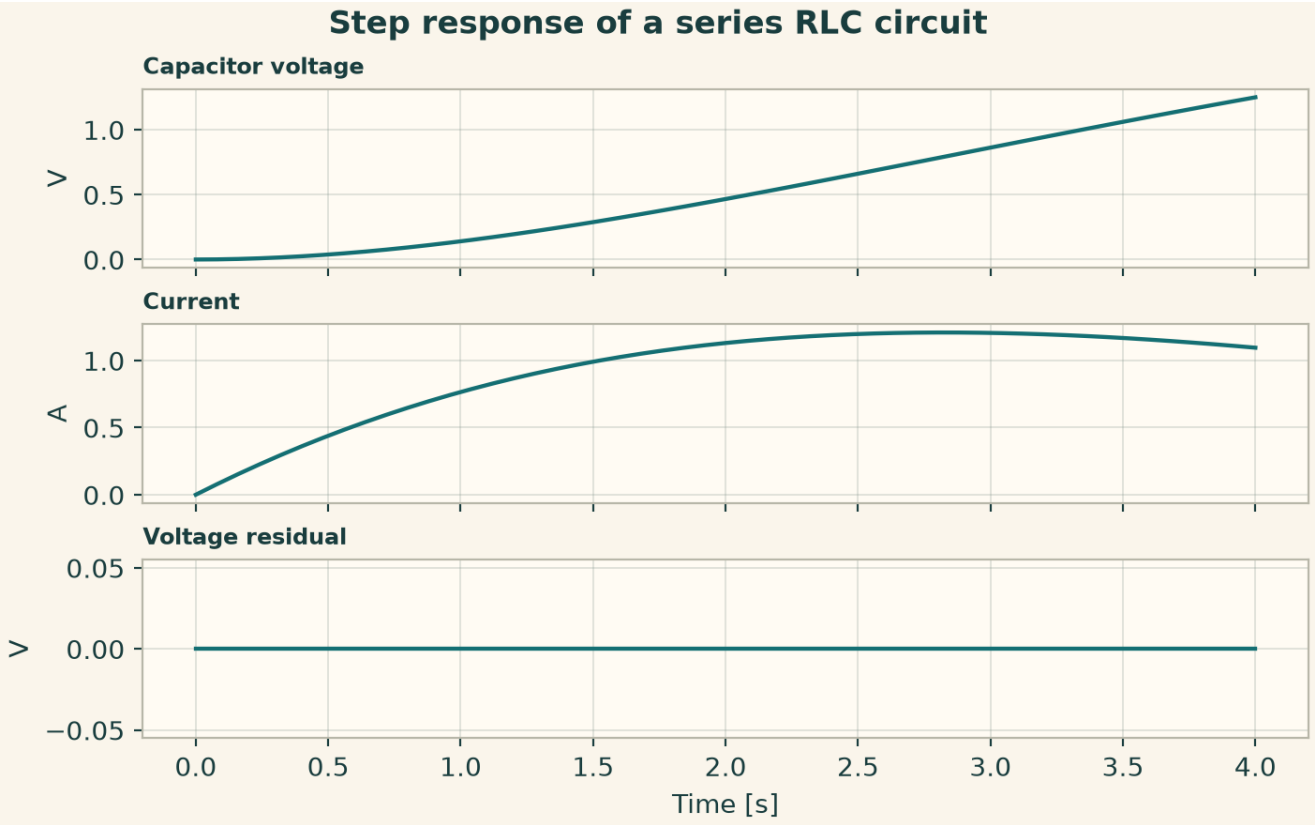
6.7 Electromagnetics recipes

6.7.1 Step response of a series RLC circuit

Apply a voltage step to a series circuit initially at rest. Read charge through capacitor voltage and verify Kirchhoff closure through the derived voltage residual.

```
from onesim.physics.electromagnetics import SeriesRLC

domain = LumpedDomain()
model = SeriesRLC(
    domain=domain.system,
    resistance=1,
    inductance=2,
    capacitance=3,
    source_voltage=2,
)
t, data = history_case(
    domain,
    model,
    dict(charge=0.0, current=0.0),
    ("capacitor_voltage", "current", "voltage_residual"),
```

```
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(data["voltage_residual"], 0, atol=1e-12)
fig = history_plot("Step response of a series RLC circuit", model, t, data)
plt.show()
```

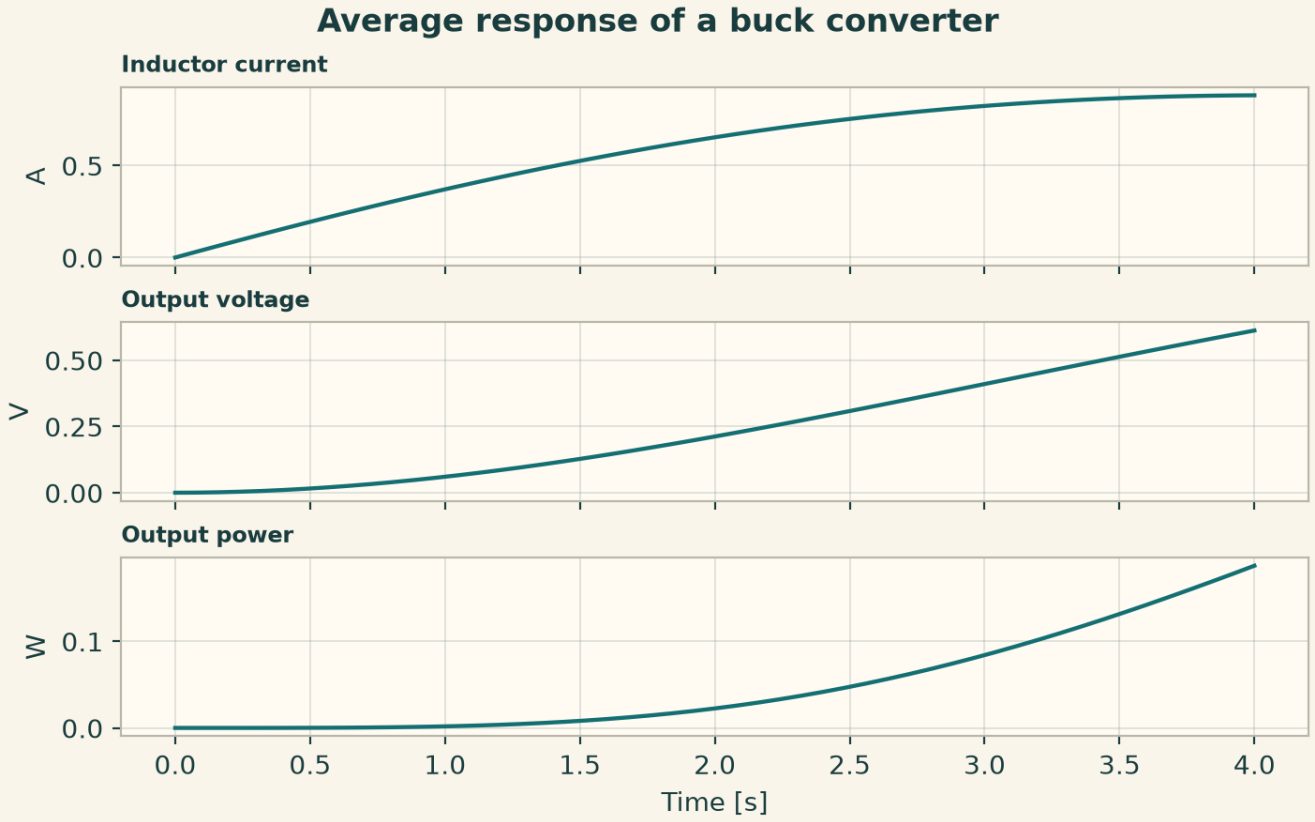


6.7.2 Average response of a buck converter

Keep duty cycle fixed and advance the average inductor current and output voltage. The model represents averaged switching and load dynamics.

```
from onesim.physics.electromagnetics import AveragedBuckConverter

domain = LumpedDomain()
model = AveragedBuckConverter(
    domain=domain.system,
    input_voltage=2,
    duty_cycle=0.4,
    inductance=2,
    capacitance=3,
    load_resistance=2,
    series_resistance=0.2,
)
t, data = history_case(
    domain,
    model,
    dict(inductor_current=0.0, output_voltage=0.0),
```

```
    ("inductor_current", "output_voltage", "output_power"),
    end=4,
    dt=0.02,
)
assert data["output_power"].min() >= 0 and data["output_voltage"][-1] > 0
fig = history_plot("Average response of a buck converter", model, t, data)
plt.show()
```

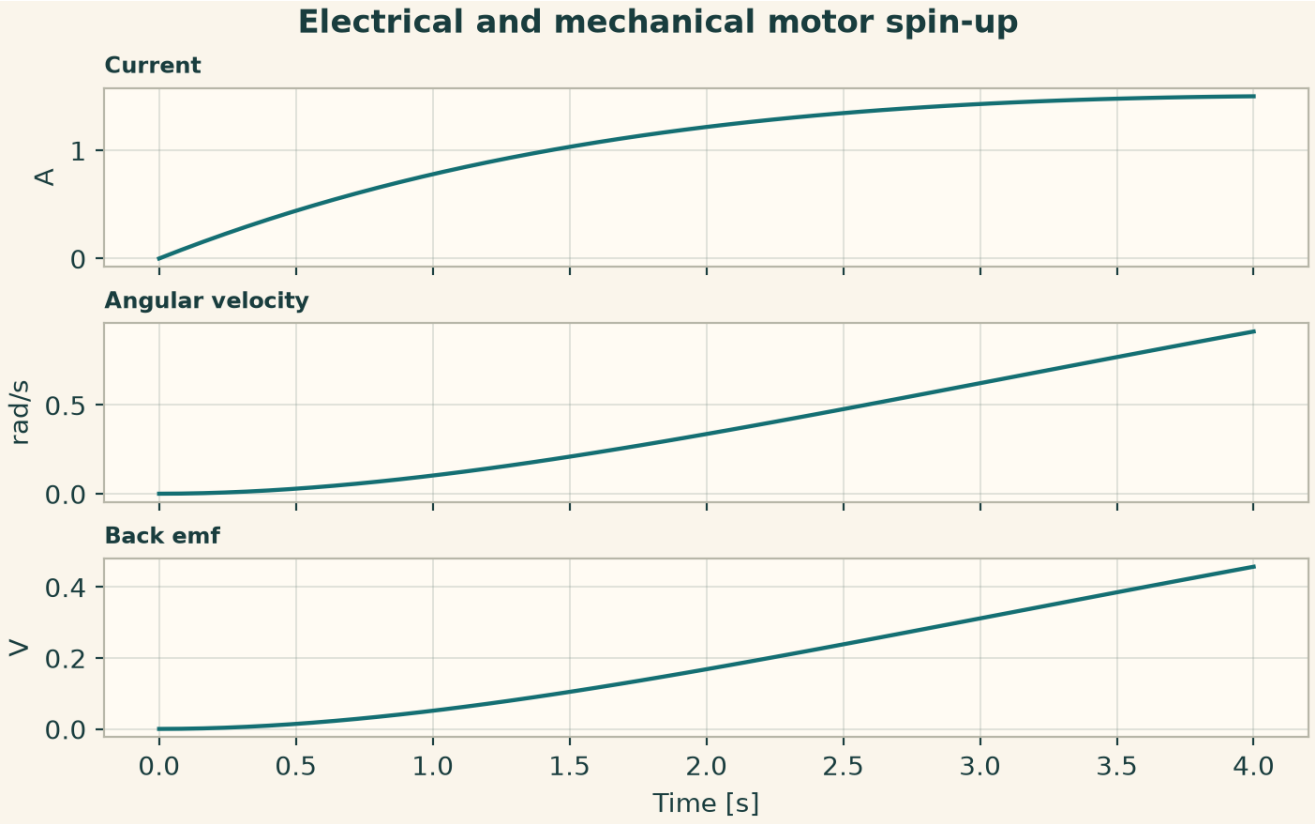


6.7.3 Electrical and mechanical motor spin-up

A fixed supply and no load torque drive current, angular acceleration and back EMF. Verify the proportionality between speed and back EMF.

```
from onesim.physics.electromagnetics import DCMotor

domain = LumpedDomain()
model = DCMotor(
    domain=domain.system,
    resistance=1,
    inductance=2,
    motor_constant=0.5,
    inertia=2,
    viscous_friction=0.2,
    applied_voltage=2,
)
t, data = history_case(
    domain,
    model,
```

```
    dict(current=0.0, angular_velocity=0.0, angle=0.0),
    ("current", "angular_velocity", "back_emf"),
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(data["back_emf"], 0.5 * data["angular_velocity"])
fig = history_plot("Electrical and mechanical motor spin-up", model, t, data)
plt.show()
```

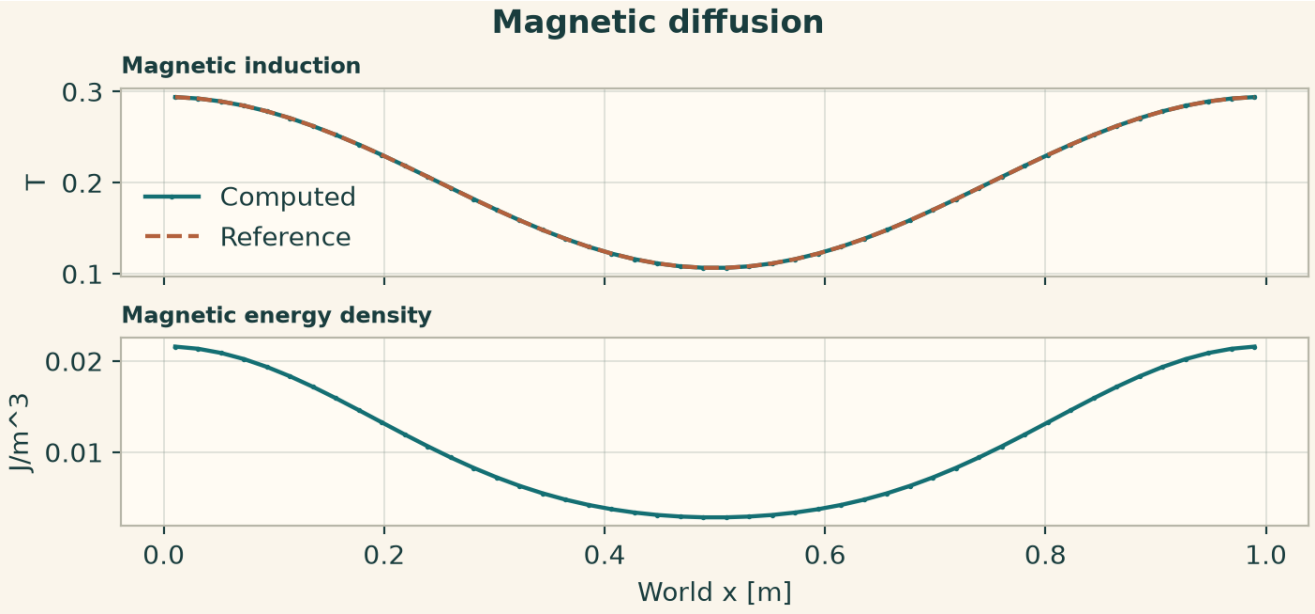


6.7.4 Decay of a periodic magnetic Fourier mode

The diffusivity is the reciprocal of permeability times conductivity. Compare the native sample values against the decaying magnetic mode.

```
from onesim.physics.electromagnetics import MagneticDiffusion

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = MagneticDiffusion(domain=line.line, permeability=2, conductivity=25)
end = 0.08
wave = lambda x: 0.2 + 0.1 * np.cos(2 * np.pi * x)
result = line_experiment(
    line,
    model,
    dict(magnetic_induction=wave),
    FiniteDifference(diffusion=CentralLaplacian()),
    cells=48,
    end=end,
    dt=0.0005,
```

```
)
exact = lambda x: 0.2 + 0.1 * np.cos(2 * np.pi * x) * np.exp(
    -0.02 * (2 * np.pi) ** 2 * end
)
b = result.field(model.field("magnetic_induction"))
np.testing.assert_allclose(b.values, exact(b.space.dof_coordinates[:, 0]), atol=3e-4)
fig = profile_figure(
    "Magnetic diffusion",
    result,
    model,
    ("magnetic_induction", "magnetic_energy_density"),
    reference={"magnetic_induction": exact},
)
plt.show()
```



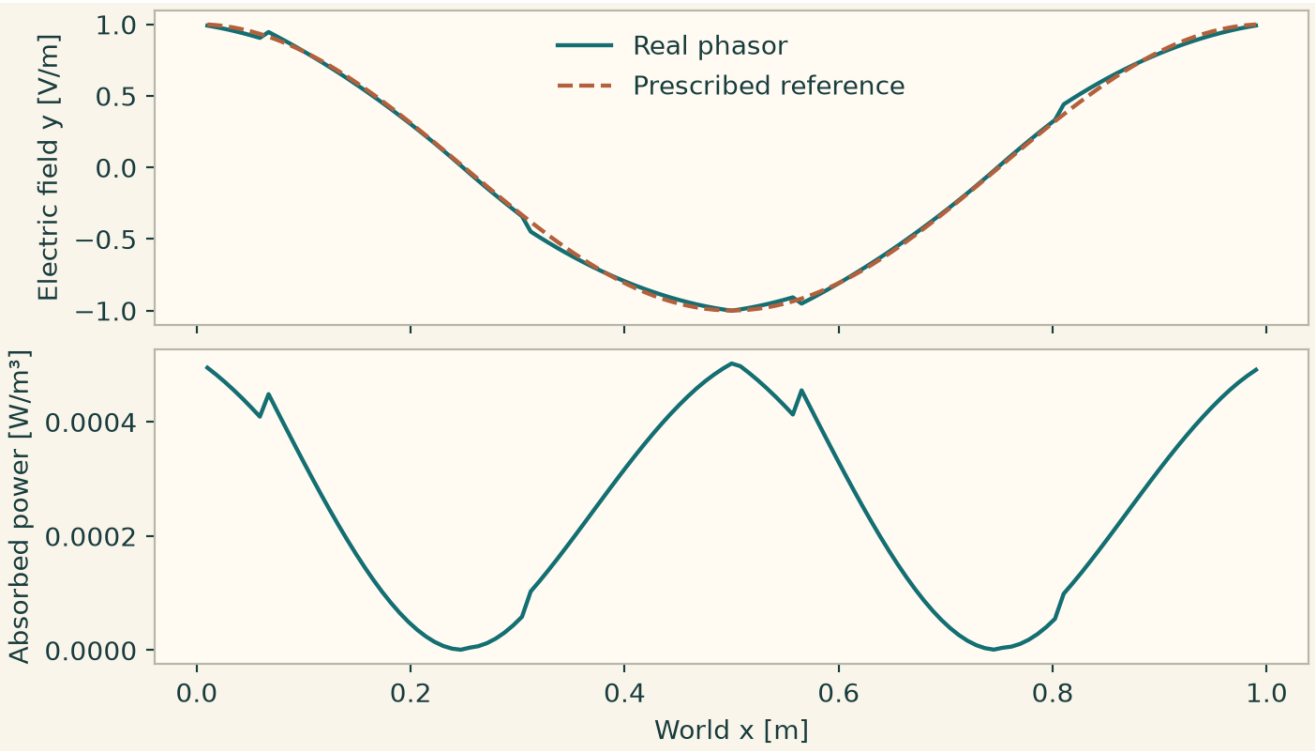
6.7.5 Driven periodic Maxwell phasor

Construct an impressed current to produce an electric cosine mode. Compare the real phasor component at physical points and plot absorbed power separately. The current uses the chosen time sign and peak amplitude convention.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials.electromagnetic import IsotropicElectromagneticMaterial
from onesim.materials.properties.units.constants import VACUUM_PERMEABILITY as MU0
from onesim.materials.properties.units.constants import VACUUM_PERMITTIVITY as EPS0
from onesim.meshing import PeriodicPair, StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Nedelec
from onesim.numerics.linear import ComplexSparseDirect
from onesim.physics.electromagnetics import FrequencyDomainMaxwell
```

```
from onesim.simulation import Problem, Simulation
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

box = Box(length=1, width=1, height=1)
frequency, conductivity = 1e7, 0.001
omega = 2 * np.pi * frequency
epsilon = EPS0 + 1j * conductivity / omega
factor = (4 * np.pi**2 - omega**2 * MU0 * epsilon) / (1j * omega * MU0)


def current(x, y, z):
    return np.cos(2 * np.pi * x)[:, None] * np.array([0, factor, 0])


model = FrequencyDomainMaxwell(
    domain=box.volume,
    frequency=frequency,
    impressed_current=current,
    material=IsotropicElectromagneticMaterial(
        relative_permittivity=1,
        relative_permeability=1,
        conductivity=conductivity,
    ),
    phasor_convention="exp(-i*omega*t)",
    amplitude_convention="peak",
)
conditions = tuple(
    PeriodicBoundary(
        physics=model,
        pair=PeriodicPair(
            getattr(box.boundaries, axis + "_min"),
            getattr(box.boundaries, axis + "_max"),
        ),
    )
    for axis in "xyz"
)
mesh = StructuredTetrahedralMesher(cells=(4, 4, 4)).generate(box)
sim = Simulation(
    problem=Problem(
        name="Driven Maxwell wave",
        geometry=box,
        physics=(model,),
        boundary_conditions=conditions,
    ),
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteElement(element=Nedelec(order=2)),
        linear_solver=ComplexSparseDirect(),
    ),
)
sim.validate().require_valid()
```

```
result = sim.solve_stationary()
x = np.linspace(0.01, 0.99, 121)
points = np.column_stack((x, np.full_like(x, 0.31), np.full_like(x, 0.47)))
ey = result.field(model.electric_field).evaluate(points)[:, 1]
q = np.asarray(result.field(model.absorbed_power_density).evaluate(points)).ravel()
assert np.max(np.abs(ey - np.cos(2 * np.pi * x))) < 0.25
assert q.min() >= 0
theme.use_theme()
fig, (ax, power) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
ax.plot(x, ey.real, label="Real phasor")
ax.plot(x, np.cos(2 * np.pi * x), "--", label="Prescribed reference")
ax.set_ylabel("Electric field y [V/m]")
ax.legend(frameon=False)
power.plot(x, q)
power.set(xlabel="World x [m]", ylabel="Absorbed power [W/m³]")
plt.show()
```



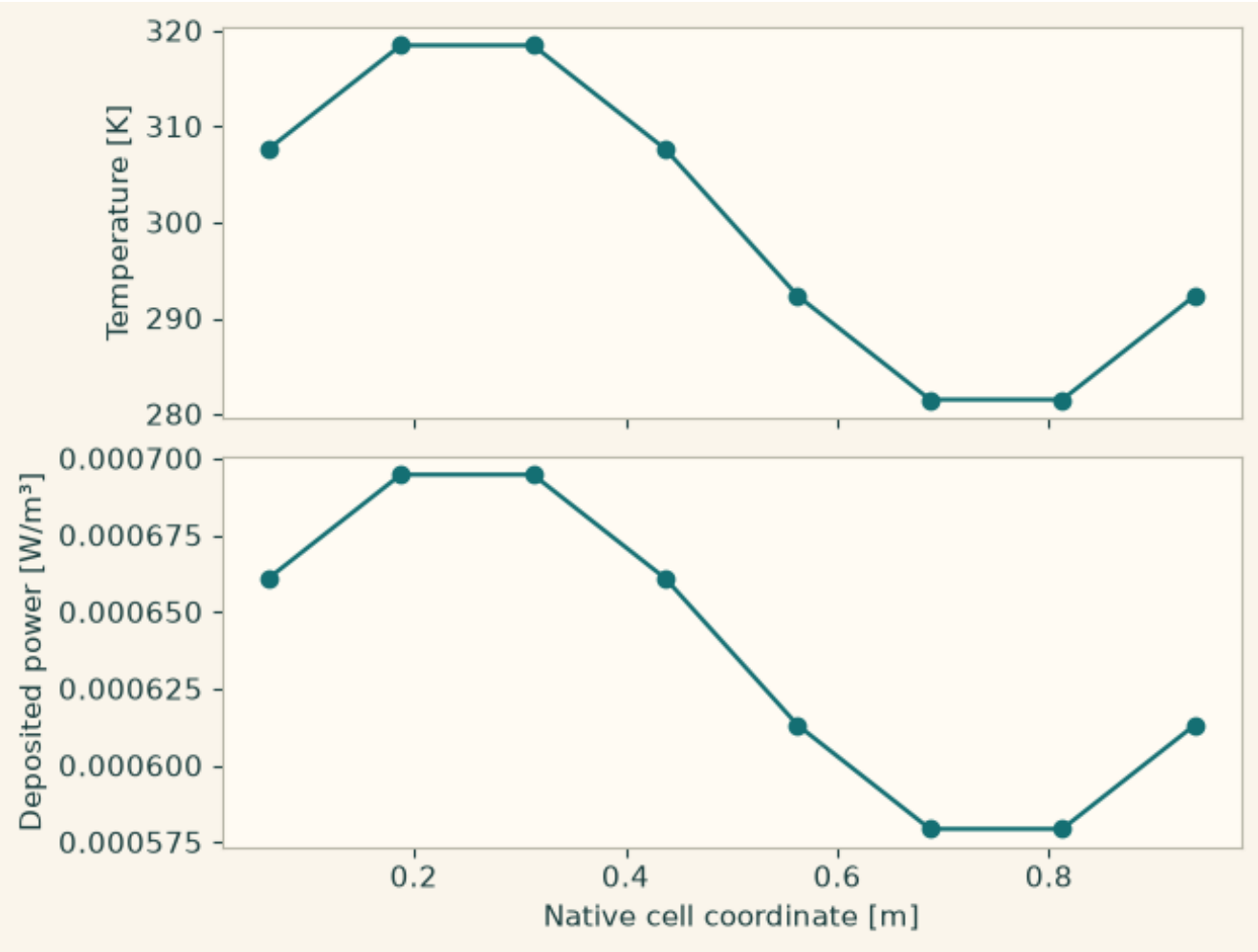
6.7.6 Couple cell temperature and native Maxwell edges

Use the two directed stationary coupling blocks with a temperature-dependent dielectric tensor. The Maxwell block uses real coordinates followed by imaginary coordinates. Verify the temperature derivatives in both coupling directions and plot the absorbed power in the heat cells.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
theme.use_theme()
```

```
from onesim.materials.properties.property_ir import MaterialIR, MaterialPropertyIR
from onesim.materials.properties.numerics.evaluation.property_evaluation import
    MaterialPropertyService
from onesim.physics.electromagnetics import operators as em_ops

grid_shape = (8, 4, 4)
x_cells = (np.arange(grid_shape[0]) + 0.5) / grid_shape[0]
reference_temperature = np.full(grid_shape, 300.0)
temperature = reference_temperature + 20 * np.broadcast_to(
    np.sin(2 * np.pi * x_cells)[:, None, None], grid_shape,
)
diagonal = "1.5 + 0.002*(temperature - 300) + 1j*(0.2 + 0.001*(temperature - 300))"
permittivity = MaterialPropertyIR(
    "epsilon_r", ((diagonal, 0, 0), (0, diagonal, 0), (0, 0, diagonal)),
    "1", value_kind="tensor", value_type="complex", symmetry="general",
    dependencies=("temperature",), dependency_units={"temperature": "K"},
)
material = MaterialIR("thermal-dielectric", "1", (permittivity,))
evaluator = MaterialPropertyService().evaluator(material, "epsilon_r")
incident_source = np.broadcast_to([1+0.2j, 0, 0], (*grid_shape, 3)).copy()
reference = em_ops.periodic_maxwell_fvm_operator(
    shape=grid_shape, extents=(1.0, 1.0, 1.0), wave_number_squared=4.0,
    relative_permittivity=evaluator, source=incident_source,
    material_state={"temperature": reference_temperature},
)
material_to_em = em_ops.periodic_maxwell_material_coupling(
    "temperature-to-maxwell", target="maxwell", source="heat",
    operator=reference, dependency="temperature",
)
power_to_heat = em_ops.periodic_maxwell_power_coupling(
    "maxwell-to-heat", target="heat", source="maxwell", operator=reference,
    dependency="temperature", residual_measure="density",
)
electric = np.broadcast_to([1.0+0.2j, 0.4j, 0.0], (*grid_shape, 3)).ravel()
edge_coordinates = np.r_[electric.real, electric.imag]
cell_temperature = temperature.ravel()
em_update = material_to_em.operator.nonlinear_contribution(edge_coordinates,
    cell_temperature, None)
heat_update = power_to_heat.operator.nonlinear_contribution(cell_temperature,
    edge_coordinates, None)
np.testing.assert_allclose(
    material_to_em.operator.nonlinear_contribution(edge_coordinates,
        reference_temperature.ravel(), None).residual,
    0.0, atol=1e-14,
)
# Verify the two temperature derivatives at the same realified electric state.
direction = np.cos(np.arange(cell_temperature.size))
h = 1e-3
perturbations = (cell_temperature - h * direction, cell_temperature + h * direction)
em_minus, em_plus = [
```

```
    material_to_em.operator.nonlinear_contribution(edge_coordinates, values,
        None).residual
    for values in perturbations
]
heat_minus, heat_plus = [
    power_to_heat.operator.nonlinear_contribution(values, edge_coordinates,
        None).residual
    for values in perturbations
]
np.testing.assert_allclose(
    em_update.source_jacobian @ direction, (em_plus - em_minus)/(2*h),
    atol=1e-12, rtol=1e-7,
)
np.testing.assert_allclose(
    heat_update.target_jacobian @ direction, (heat_plus - heat_minus)/(2*h),
    atol=1e-12, rtol=1e-7,
)
# The source appears with a minus sign on the left of the heat balance.
power_density = -heat_update.residual.reshape(grid_shape)
assert np.all(power_density > 0)
fig, (a, b) = plt.subplots(2, 1, sharex=True, layout="constrained")
a.plot(x_cells, temperature[:, 0, 0], "o-")
b.plot(x_cells, power_density[:, 0, 0], "o-")
a.set_ylabel("Temperature [K]")
b.set(xlabel="Native cell coordinate [m]", ylabel="Deposited power [W/m³]")
plt.show()
```



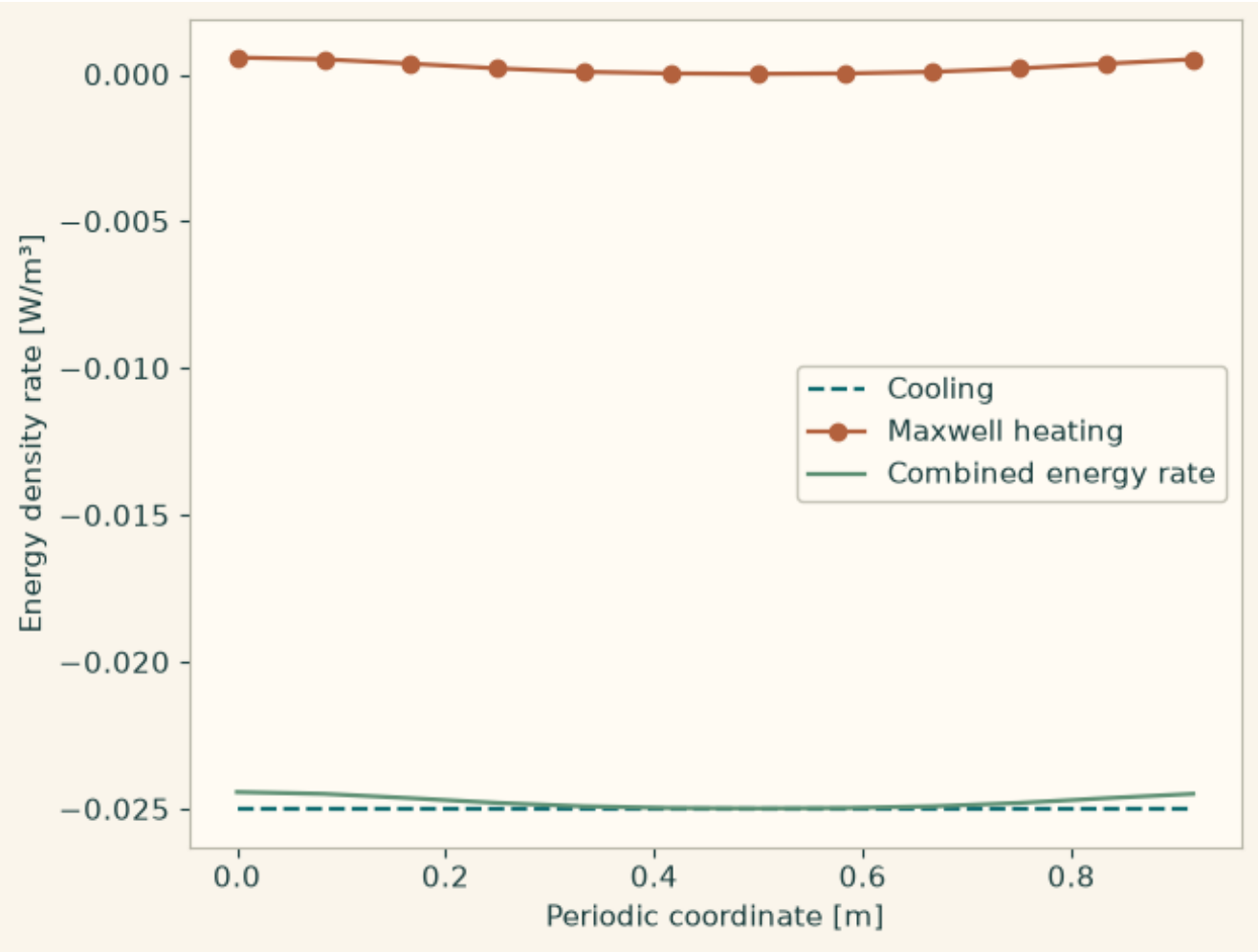
6.7.7 Compose Maxwell heating into an energy equation and DAE

Add a power source to a declared energy balance and evaluate its contribution with a manufactured electric field. The base provider supplies a linear cooling rate. Compare the full DAE adapters and the standalone energy-rate adapter on the same snapshot. The periodic compatibility names are evaluated alongside the canonical constructors.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
theme.use_theme()

from scipy import sparse
from onesim.numerics.discretization.equations.contracts.operators.operator_contracts
    import (
    DaeResidual, SpatialExecutionContext,
)
from onesim.physics.electromagnetics import operators as em_ops
from onesim.results.assessment.numerics.spatial_result_adapter import
    SpatialResultFieldSpec
from onesim.simulation.problem_definition.declarations.specification import
    EquationSpec, FieldSpec
from onesim.simulation.state.history.state_snapshot import StateSnapshot
from onesim.simulation.state.layout.state_vector import Residual
```

```
energy_id = "energy_density"
energy_equation = EquationSpec.from_residual(
    "energy-balance", "dt(energy_density) + 0.05 * energy_density = 0",
    (energy_id,), domain="periodic-box",
    time_derivative_order=1, residual_unit="W/m^3",
    field_definitions=(FieldSpec(energy_id, "J/m^3", carrier_kind="cell"),),
)
heated_equation = em_ops.maxwell_heated_equation(energy_equation,
    energy_field=energy_id)
legacy_equation = em_ops.periodic_maxwell_heated_equation(energy_equation,
    energy_field=energy_id)
assert heated_equation == legacy_equation
assert "absorbed_power_density" in heated_equation.dependent_fields
assert heated_equation.term_definitions[-1].kind == "source"
print(heated_equation.formulation.residual_form)

grid = (12, 3, 3)
x = np.arange(grid[0]) / grid[0]
phase_amplitude = 0.6 + 0.4 * np.cos(2*np.pi*x)
electric = np.zeros((*grid, 3), dtype=complex)
electric[..., 0] = (1 + 0.3j) * phase_amplitude[:, None, None]
# E points along x and depends only on x, so curl(E) is zero.
k2 = 4.0
epsilon = 1.5 + 0.2j
maxwell = em_ops.periodic_maxwell_fvm_operator(
    shape=grid, extents=(1.0, 1.0, 1.0), wave_number_squared=k2,
    relative_permittivity=epsilon, source=-k2 * epsilon * electric,
)
constraint = em_ops.periodic_maxwell_dae_constraint(maxwell, material_fields={})
energy = np.full(np.prod(grid), 0.5)
snapshot = StateSnapshot("maxwell-energy", 0.0, {
    energy_id: energy,
    constraint.field_ids[0]: electric.real,
    constraint.field_ids[1]: electric.imag,
})
context = SpatialExecutionContext(constraint.to_carrier().carrier_id,
    snapshot.state_type, 0.0)
np.testing.assert_allclose(constraint(snapshot, 0.0), 0.0, atol=1e-12)

class CellCoolingRate:
    state_field_ids = (energy_id,)
    jacobian_scalar_representation = "physical_real"
    result_fields = (SpatialResultFieldSpec(energy_id, "Energy density", "J/m^3",
        "cell"),)

    def residual(self, values, context):
        return Residual(-0.05 * np.asarray(values), context.state_type)

    def real_jacobian(self, values, context):
        return -0.05 * sparse.eye(np.size(values), format="csr")
```

```
class CellCoolingDae:
    # The field order of the physical rate and its codec derivative is explicit.
    result_fields = CellCoolingRate.result_fields

    def residual(self, snapshot, context):
        rate = CellCoolingRate()
        return Residual(DaeResidual(
            differential_rate=lambda state, t: rate.residual(state.field(energy_id),
            context).values,
            algebraic_residual=constraint,
            differential_field_ids=(energy_id,),
            differential_codec_jacobian=self.rate_jacobian,
            algebraic_field_ids=constraint.field_ids,
        ), context.state_type)

    @staticmethod
    def rate_jacobian(snapshot, time, codec):
        columns = codec.field_dof_indices(energy_id)
        rows = np.arange(len(columns))
        return sparse.csr_matrix(
            (np.full(len(columns), -0.05), (rows, columns)),
            shape=(len(columns), len(codec.dof_map.descriptors)),
        )

rate_operator = em_ops.periodic_maxwell_energy_rate(CellCoolingRate(), constraint,
    energy_field=energy_id)
dae_operator = em_ops.maxwell_dae_heating(CellCoolingDae(), constraint,
    energy_field=energy_id)
dae_alias = em_ops.periodic_maxwell_dae_heating(CellCoolingDae(), constraint,
    energy_field=energy_id)
total = rate_operator.residual(snapshot, context).values
for adapter in (dae_operator, dae_alias):
    residual = adapter.residual(snapshot, context).values
    np.testing.assert_allclose(residual.differential_rate(snapshot, 0.0), total)
    np.testing.assert_allclose(residual.algebraic_residual(snapshot, 0.0), 0.0,
        atol=1e-12)
cooling = CellCoolingRate().residual(energy, context).values
power = maxwell.absorbed_power_density(electric).ravel()
np.testing.assert_allclose(total - cooling, power, atol=1e-16)
fig, ax = plt.subplots(layout="constrained")
ax.plot(x, cooling.reshape(grid)[:, 0, 0], "--", label="Cooling")
ax.plot(x, power.reshape(grid)[:, 0, 0], "o-", label="Maxwell heating")
ax.plot(x, total.reshape(grid)[:, 0, 0], label="Combined energy rate")
ax.set(xlabel="Periodic coordinate [m]", ylabel="Energy density rate [W/m³]")
ax.legend()
plt.show()
```



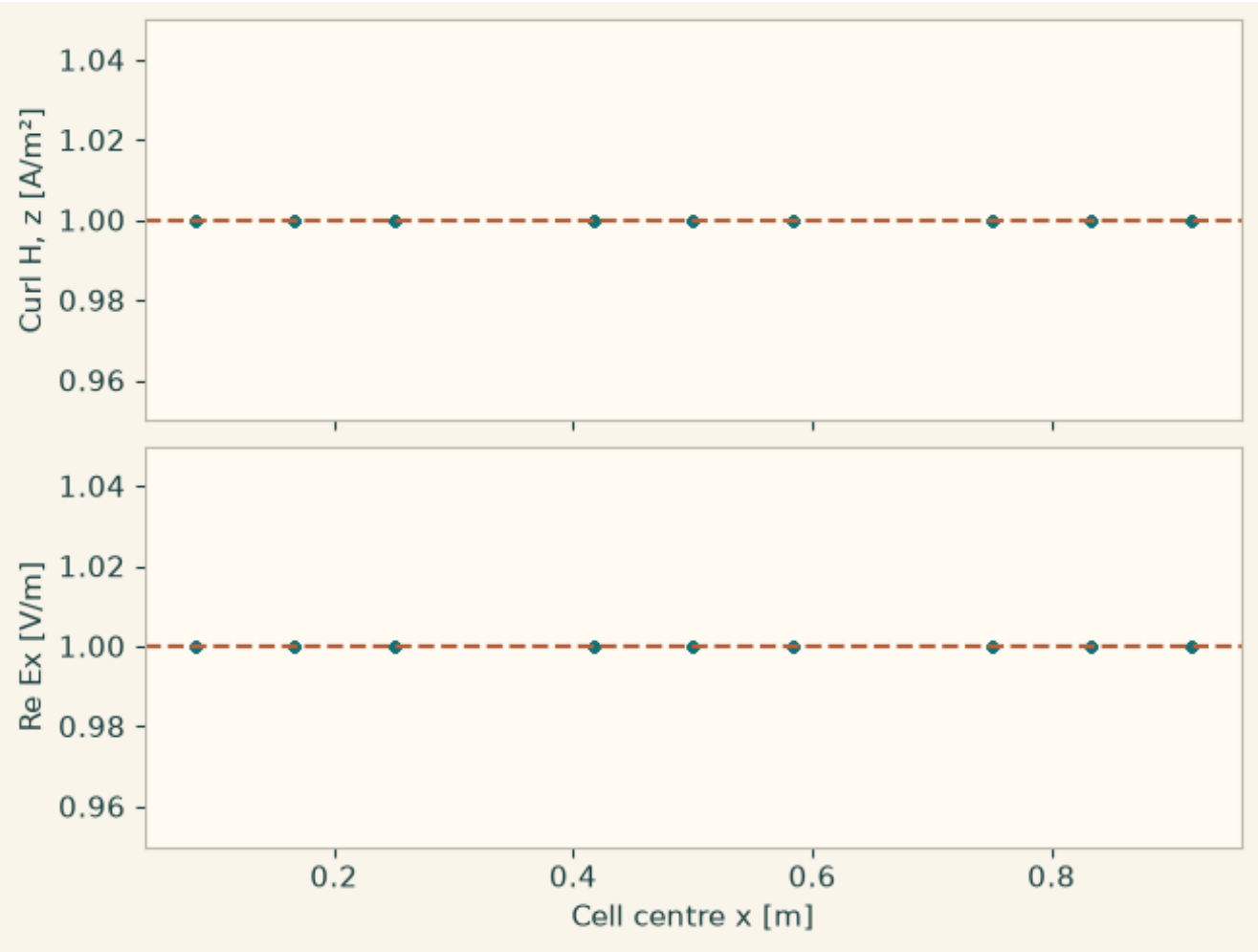
6.7.8 Assemble nodal and Nédélec curl–curl systems

Assemble, solve and reconstruct a real nodal curl–curl system with an affine trace and a complex Nédélec system with a manufactured uniform field. The latter has an explicit empty essential-edge set. Feed the solved edge coefficients into the corresponding DAE constraint and check its residual.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
theme.use_theme()

from onesim.cad.d3 import Box
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics.fem.contracts.vector_fem import (
    NedelecCurlCurlConfiguration, NedelecCurlCurlRequest,
    NodalCurlCurlConfiguration, NodalCurlCurlRequest,
)
from onesim.physics.electromagnetics import operators as em_ops
from onesim.simulation.state.history.state_snapshot import StateSnapshot
from scipy.sparse import issparse
from scipy.sparse.linalg import spsolve

geometry = Box(length=1.0, width=1.0, height=1.0)
```

```
mesh = StructuredTetrahedralMesher(cells=(3, 3, 3)).generate(geometry)
points = np.asarray(mesh.coordinates)
tetrahedra = np.asarray(mesh.cells)
boundary = tuple(np.flatnonzero(np.any(np.isclose(points, 0.0) | np.isclose(points,
    1.0), axis=1)).tolist())
rotation = np.column_stack((-0.5*points[:, 1], 0.5*points[:, 0],
    np.zeros(len(points))))
# Prescribe the affine field only on the Dirichlet nodes.
boundary_values = np.zeros_like(points)
boundary_values[list(boundary)] = rotation[list(boundary)]
nodal = em_ops.nodal_curl_curl_operator(
    NodalCurlCurlRequest(
        points, tetrahedra, source=(0.0, 0.0, 0.0), curl_coefficient=1.0,
        boundary_nodes=boundary, boundary_values=boundary_values,
    ),
    NodalCurlCurlConfiguration(),
)
nodal_system = nodal.assemble(None)
assert nodal_system.right_hand_side.size > 0

def solve_assembled(system):
    matrix = system.matrix
    if hasattr(matrix, "tocsr"):
        matrix = matrix.tocsr()
    elif hasattr(matrix, "to_scipy_sparse"):
        matrix = matrix.to_scipy_sparse()
    rhs = system.right_hand_side
    solved = spsolve(matrix, rhs) if issparse(matrix) else np.linalg.solve(matrix, rhs)
    np.testing.assert_allclose(matrix @ solved, rhs, atol=1e-12)
    return solved

report = {"provider_id": "demo.direct", "converged": True}
nodal_solution = solve_assembled(nodal_system)
nodal_state = nodal.state_from_solution(nodal_solution, nodal_system, report)
nodal_fields = nodal.reconstruction(nodal_state, None)
np.testing.assert_allclose(nodal_fields[nodal.field_id], rotation, atol=1e-12)
nodal_curl_z = np.asarray(nodal_fields[nodal.curl_field_id])[:, 2]
np.testing.assert_allclose(nodal_curl_z, 1.0, atol=1e-12)

expected = np.array([1+0.3j, 0.2j, -0.2+0.1j])
mass_coefficient = 2.0+0.4j
edge_request = NedelecCurlCurlRequest(
    points, tetrahedra, curl_coefficient=1.0, mass_coefficient=mass_coefficient,
    source=mass_coefficient*expected, essential_edge_ids=(),
)
edge_configuration = NedelecCurlCurlConfiguration()
edge = em_ops.nedelec_curl_curl_operator(edge_request, edge_configuration)
edge_system = edge.assemble(None)
edge_solution = solve_assembled(edge_system)
edge_fields = edge.reconstruction(edge.state_from_solution(edge_solution, edge_system,
    report), None)
```

```
edge_field = np.asarray(edge_fields[edge.field_id])
np.testing.assert_allclose(edge_field, np.broadcast_to(expected, edge_field.shape),
    atol=1e-12)
np.testing.assert_allclose(edge_fields[edge.curl_field_id], 0.0, atol=1e-12)

constraint = em_ops.nedelec_maxwell_dae_constraint(
    coordinates=points, cells=tetrahedra, carrier_id="demo-tetrahedra",
    source=mass_coefficient*expected, essential_edge_ids=(),
        mass_coefficient=mass_coefficient,
    configuration=edge_configuration,
)
snapshot = StateSnapshot("native-maxwell-edges", 0.0, dict(zip(
    constraint.field_ids, (edge_solution.real, edge_solution.imag), strict=True,
)))
np.testing.assert_allclose(constraint(snapshot, 0.0), 0.0, atol=1e-12)
print(f"Nodal free coordinates {nodal_solution.size}, complex edge coordinates
    {edge_solution.size}")

centres = points[tetrahedra].mean(axis=1)
fig, (a, b) = plt.subplots(2, 1, sharex=True, layout="constrained")
a.plot(centres[:, 0], nodal_curl_z, ".", label="Nodal FEM")
a.axhline(1.0, color=theme.COPPER, linestyle="--")
a.set(ylabel="Curl H, z [A/m²]", ylim=(0.95, 1.05))
b.plot(centres[:, 0], edge_field[:, 0].real, ".", label="Nédélec FEM")
b.axhline(expected[0].real, color=theme.COPPER, linestyle="--")
b.set(xlabel="Cell centre x [m]", ylabel="Re Ex [V/m]", ylim=(0.95, 1.05))
plt.show()
```



6.8 Electrostatics recipes

The uniform field in the first-case chapter plots the potential and electric field from a fixed voltage trace. The mixed-element recipe in Section 9.4 compares two H(div) carriers on a second Poisson problem.

6.9 Geotechnics recipes

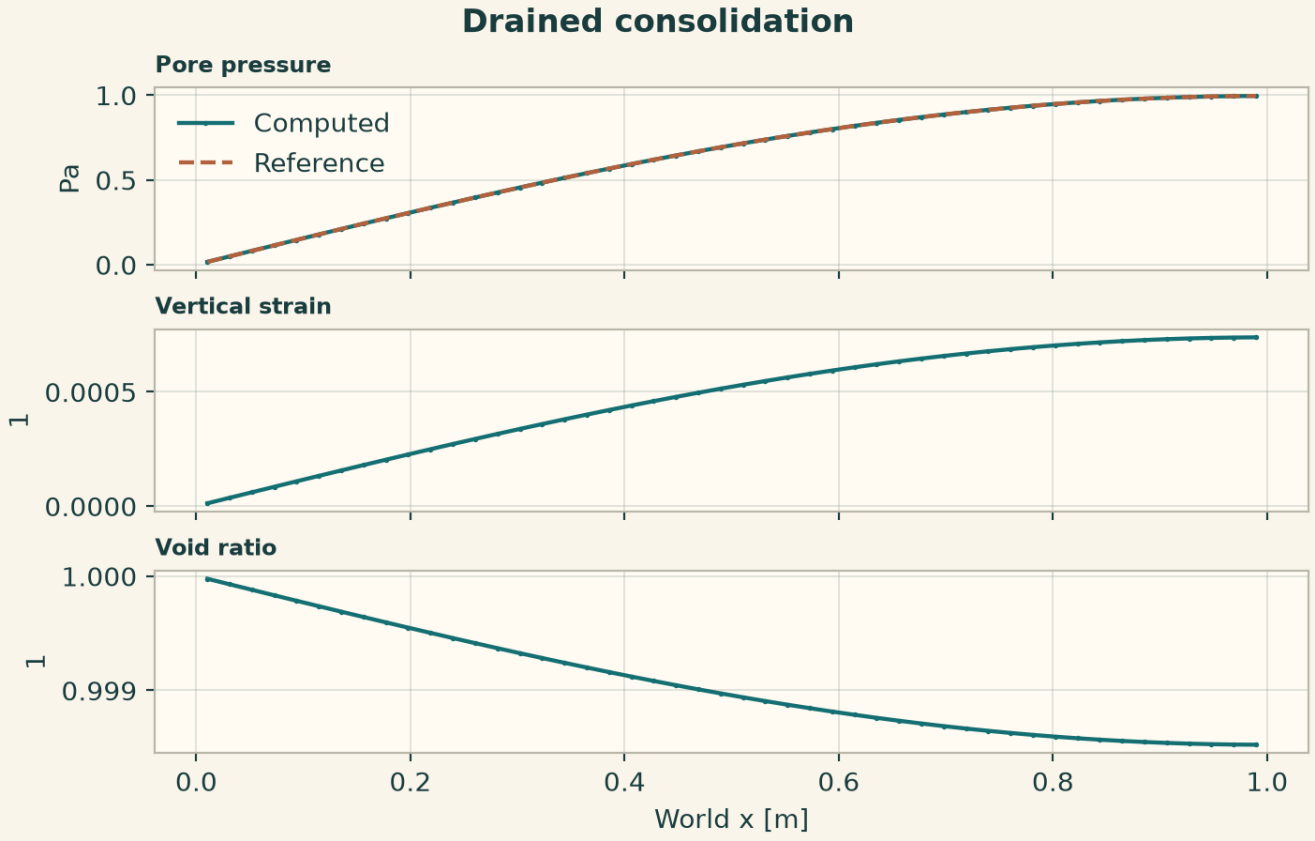
This section includes the field-based declarations and the independently callable geotechnics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.9.1 One-dimensional drainage and consolidation

Drain one end and make the opposite end impermeable. The half-sine pressure mode satisfies both conditions. Keep pore pressure, strain and void ratio as separate initial fields.

```
from onesim.physics.geotechnics import (
    DrainedBoundary,
    ImpermeableBoundary,
    TerzaghiConsolidation,
)

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = TerzaghiConsolidation(
```

```
    domain=line.line,
    permeability=0.01,
    fluid_viscosity=1,
    constrained_storage=1,
    constrained_compressibility=0.15,
    total_stress=2,
)
conditions = (
    DrainedBoundary(physics=model, boundary=line.boundaries.start),
    ImpermeableBoundary(physics=model, boundary=line.boundaries.end),
)
wave = lambda x: np.sin(np.pi * x / 2)
end = 0.2
result = line_experiment(
    line,
    model,
    dict(pore_pressure=wave, vertical_strain=0.0, void_ratio=1.0),
    FiniteDifference(gradient=CentralDifference(), diffusion=CentralLaplacian()),
    cells=48,
    end=end,
    dt=0.001,
    conditions=conditions,
)
exact = lambda x: wave(x) * np.exp(-0.01 * (np.pi / 2) ** 2 * end)
p = result.field(model.pore_pressure)
np.testing.assert_allclose(p.values, exact(p.space.dof_coordinates[:, 0]), atol=4e-4)
fig = profile_figure(
    "Drained consolidation",
    result,
    model,
    ("pore_pressure", "vertical_strain", "void_ratio"),
    reference={"pore_pressure": exact},
)
plt.show()
```

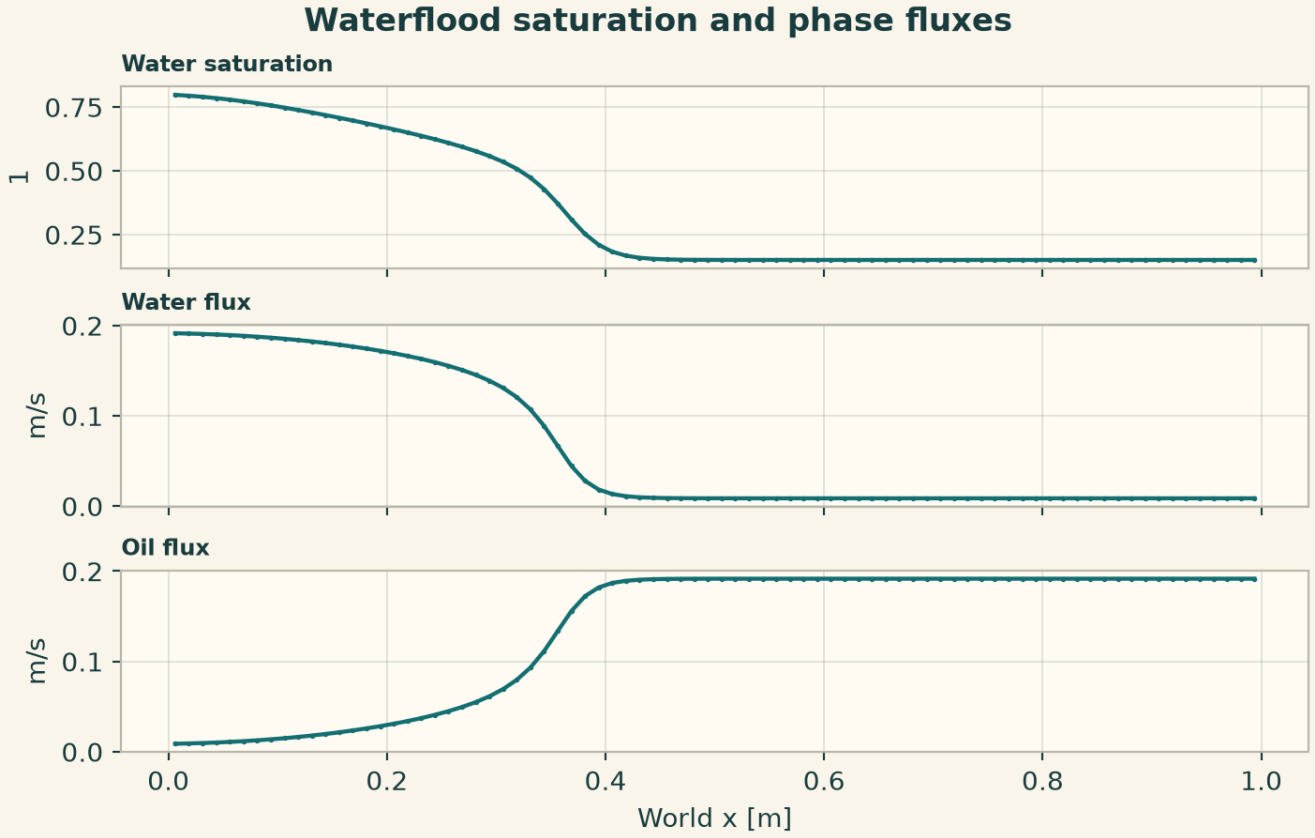


6.9.2 Waterflood with quadratic Corey fractional flow

Feed water into a domain with low initial saturation. Water and oil fluxes partition the prescribed Darcy velocity. The saturation history advances with porosity in the storage term.

```
from onesim.physics.geotechnics import (
    BuckleyLeverettFlow,
    QuadraticCoreyFractionalFlow,
    SaturationInlet,
    SaturationOutflow,
)

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = BuckleyLeverettFlow(
    domain=line.line,
    porosity=0.25,
    darcy_velocity=0.2,
    fractional_flow=QuadraticCoreyFractionalFlow(viscosity_ratio=0.7),
)
conditions = (
    SaturationInlet(physics=model, boundary=line.boundaries.start, value=0.8),
    SaturationOutflow(physics=model, boundary=line.boundaries.end),
)
result = line_experiment(
    line,
    model,
    dict(water_saturation=0.15),
    FiniteVolume(convection=Rusanov()),
    cells=80,
```

```
    end=0.25,
    dt=0.001,
    conditions=conditions,
)
s = result.field(model.water_saturation).values
assert s.min() >= 0 and s.max() <= 1 and s[0] > s[-1]
np.testing.assert_allclose(
    result.field(model.water_flux).values + result.field(model.oil_flux).values, 0.2
)
fig = profile_figure(
    "Waterflood saturation and phase fluxes",
    result,
    model,
    ("water_saturation", "water_flux", "oil_flux"),
)
plt.show()
```



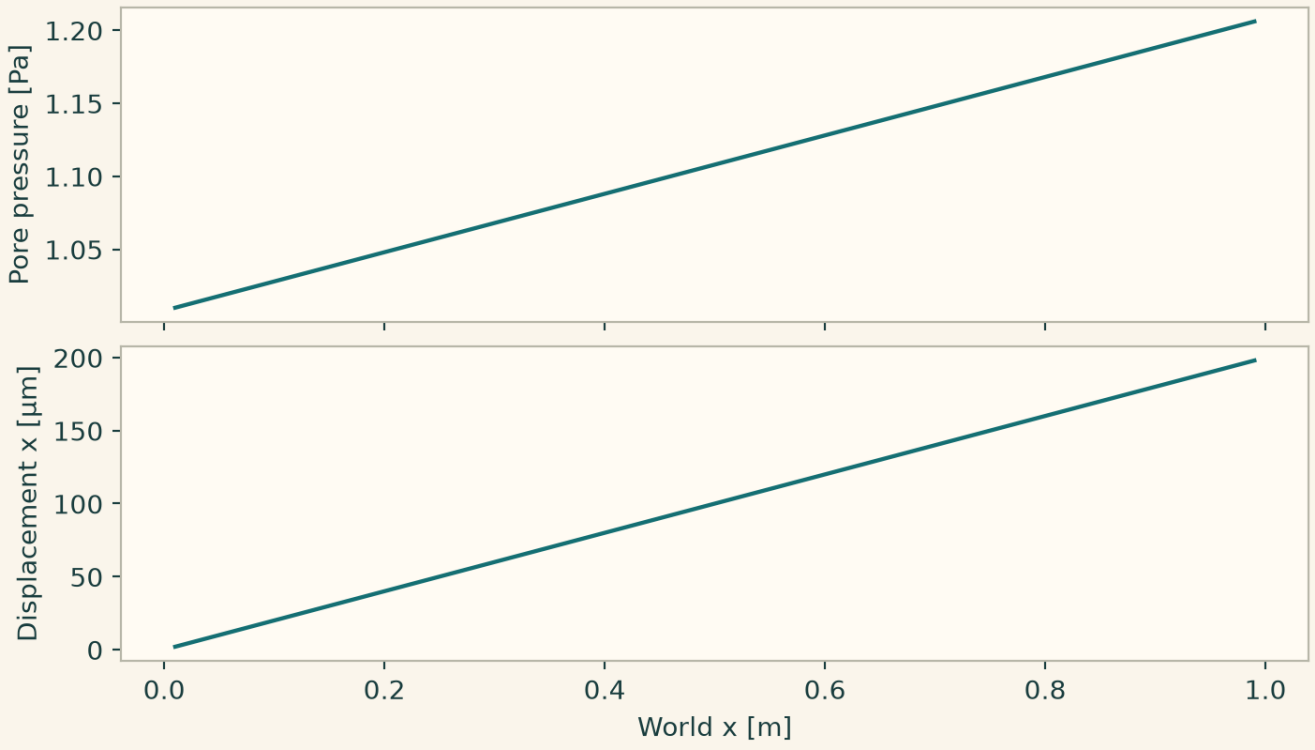
6.9.3 Manufactured three-dimensional Biot patch

The pressure trace is in pascals, while the initial state is fluid content. Apply a time-varying displacement and compare the recovered pore pressure and displacement to a manufactured field. The body force and supply make this a check of the full balance.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
```

```
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange, Nitsche
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.geotechnics import BiotConsolidation, PrescribedPorePressure
from onesim.physics.solid_mechanics import FixedDisplacement
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

box = Box(length=1, width=1, height=1)
model = BiotConsolidation(
    domain=box.volume,
    youngs_modulus=2.0,
    poisson_ratio=0.25,
    biot_coefficient=0.8,
    specific_storage=0.7,
    permeability=0.01,
    fluid_viscosity=1.0,
    fluid_supply=0.304,
    body_force=(0.16, 0, 0),
)


def displacement(x, y, z, t):
    return 0.01 * t * np.column_stack((x, y, z))


def pressure(time, coordinate):
    return [1 + 0.2 * coordinate[0] + 0.4 * time]


conditions = (
    FixedDisplacement(physics=model, boundary=box.boundaries.all, value=displacement),
    PrescribedPorePressure(
        physics=model, boundary=box.boundaries.all, value=pressure
    ),
)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
sim = Simulation(
    problem=Problem(
        name="Biot patch",
        geometry=box,
        physics=(model,),
        boundary_conditions=conditions,
        initial_conditions=(
            InitialValue(model.fluid_content, lambda x: 0.7 * (1 + 0.2 * x)),
        ),
    ),
    mesh=mesh,
```

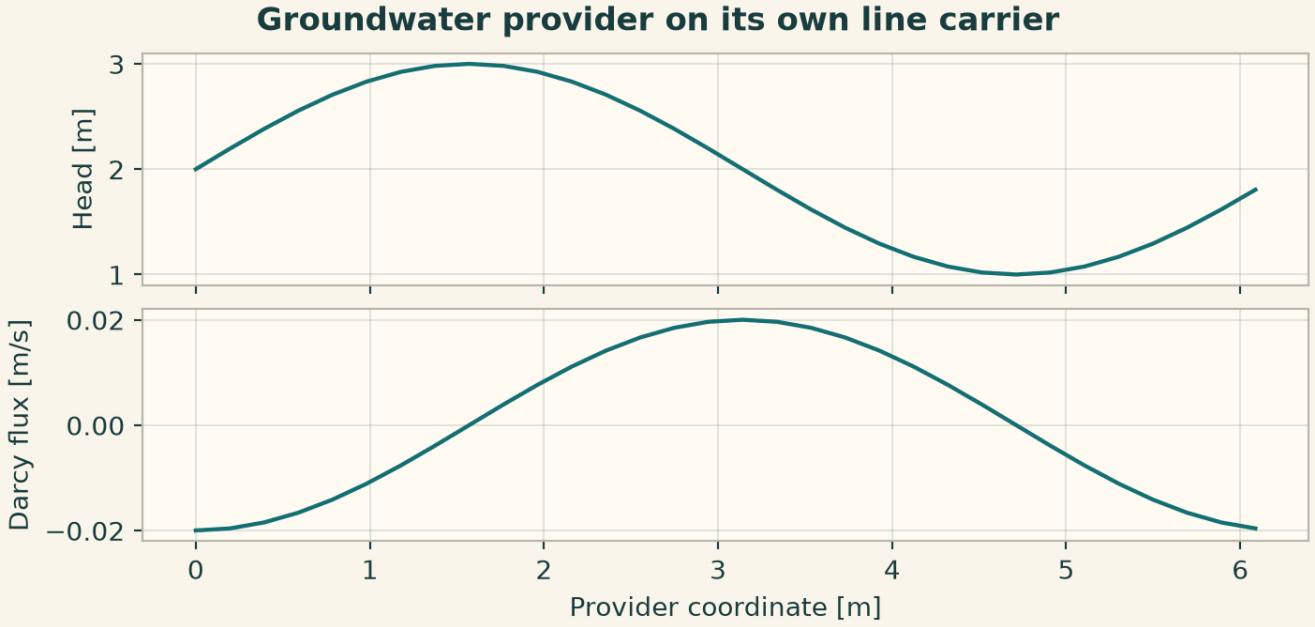
```
    numerics=Numerics(
        spatial=FiniteElement(element=Lagrange(1), boundary_enforcement=Nitsche()),
        linear_solver=SparseDirect(),
        time_integrator=BackwardEuler(),
        nonlinear_solver=Newton(
            linear_solver=SparseDirect(),
            jacobian="analytic",
            absolute_tolerance=1e-12,
            relative_tolerance=1e-11,
        ),
    ),
)
sim.validate().require_valid()
result = sim.run(start_time=0, end_time=0.02, time_step=0.01)
x = np.linspace(0.01, 0.99, 100)
points = np.column_stack((x, np.full_like(x, 0.35), np.full_like(x, 0.55)))
p = result.field(model.pore_pressure).evaluate(points)
u = result.field(model.displacement).evaluate(points)[:, 0]
np.testing.assert_allclose(p, 1 + 0.2 * x + 0.008, atol=2e-10)
np.testing.assert_allclose(u, 0.0002 * x, atol=2e-11)
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
a.plot(x, p)
a.set_ylabel("Pore pressure [Pa]")
b.plot(x, 1e6 * u)
b.set(xlabel="World x [m]", ylabel="Displacement x [µm]")
plt.show()
```



6.9.4 Groundwater provider on its own line carrier

Pass the initial hydraulic head array through the configuration and read the final head and Darcy flux on the returned coordinate array.

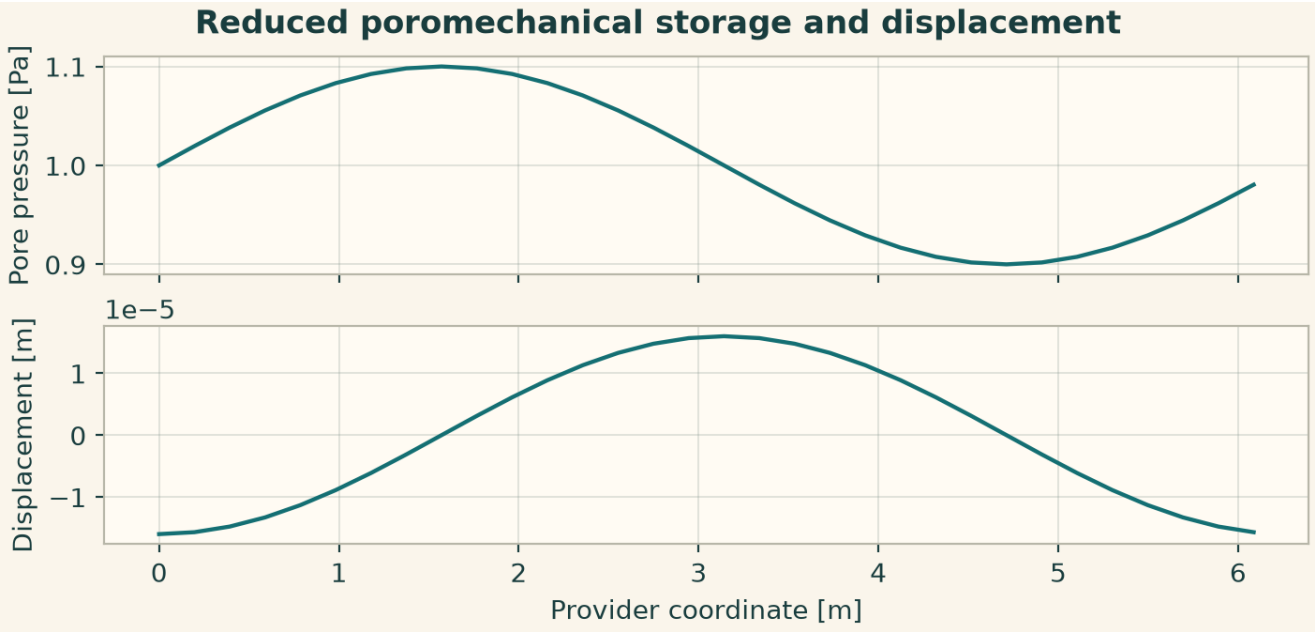
```
from onesim.physics.geotechnics import GroundwaterConfiguration, GroundwaterPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = GroundwaterConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        conductivity=0.02,
        storage=1,
        initial_head=2 + wave,
    )
)
result = run_service(GroundwaterPublicApi(), configuration)
fig = service_figure(
    "Groundwater provider on its own line carrier",
    result.coordinates,
    "Provider coordinate [m]",
    ("Head [m]", result.head),
    ("Darcy flux [m/s]", result.darcy_flux),
)
plt.show()
```



6.9.5 Reduced poromechanical storage and displacement

Configure the pressure field and permeability of the line provider. The result also returns the elastic and storage energy diagnostics.

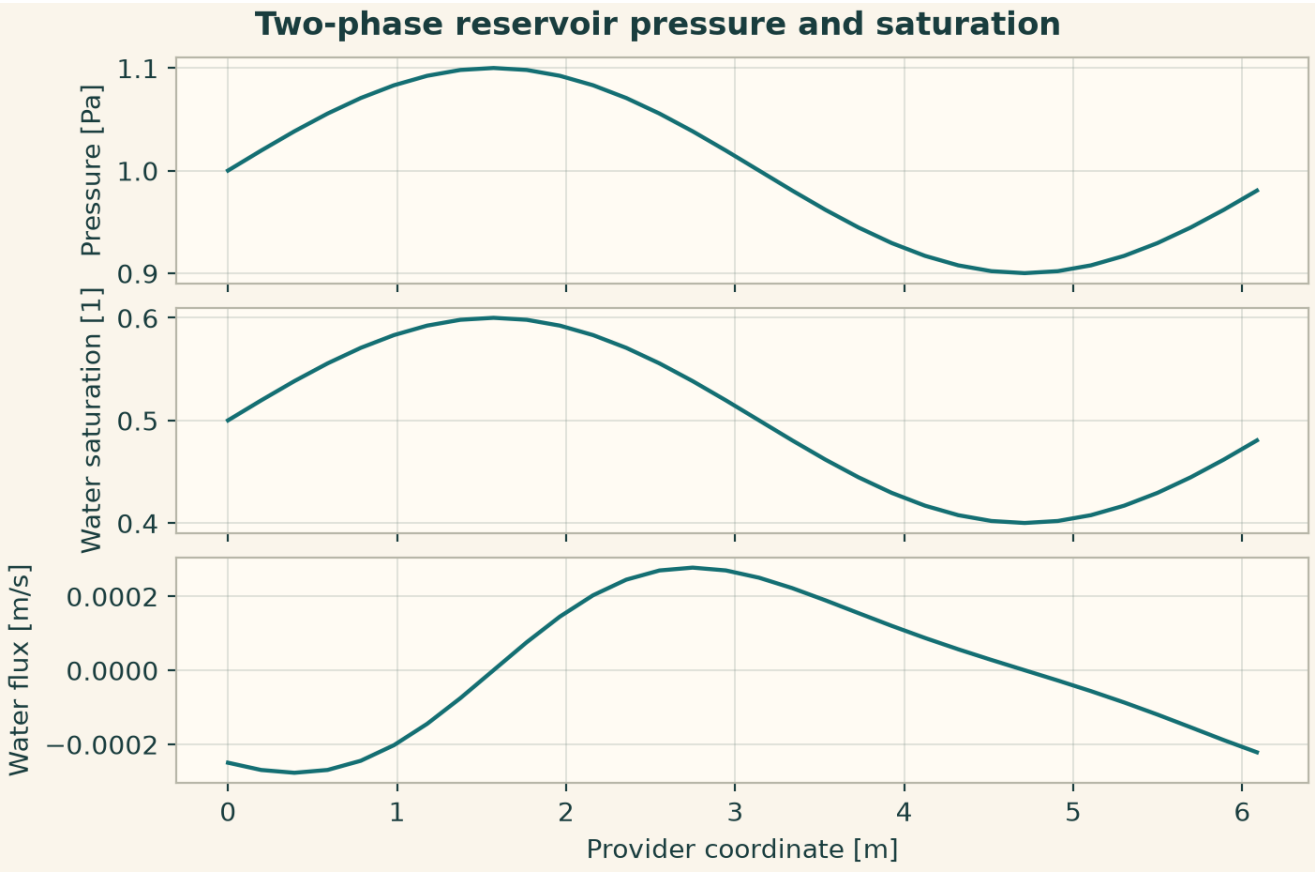
```
from onesim.physics.geotechnics import (
    PoromechanicsConfiguration,
    PoromechanicsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = PoromechanicsConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_pore_pressure=1 + 0.1 * wave,
        permeability=0.01,
        solid_mobility=0.02,
    )
)
result = run_service(PoromechanicsPublicApi(), configuration)
fig = service_figure(
    "Reduced poromechanical storage and displacement",
    result.coordinates,
    "Provider coordinate [m]",
    ("Pore pressure [Pa]", result.pore_pressure),
    ("Displacement [m]", result.displacement),
)
plt.show()
```



6.9.6 Two-phase reservoir pressure and saturation

Supply pressure and physical water saturation separately. The result preserves physical saturation and also reports effective saturation and phase fluxes.

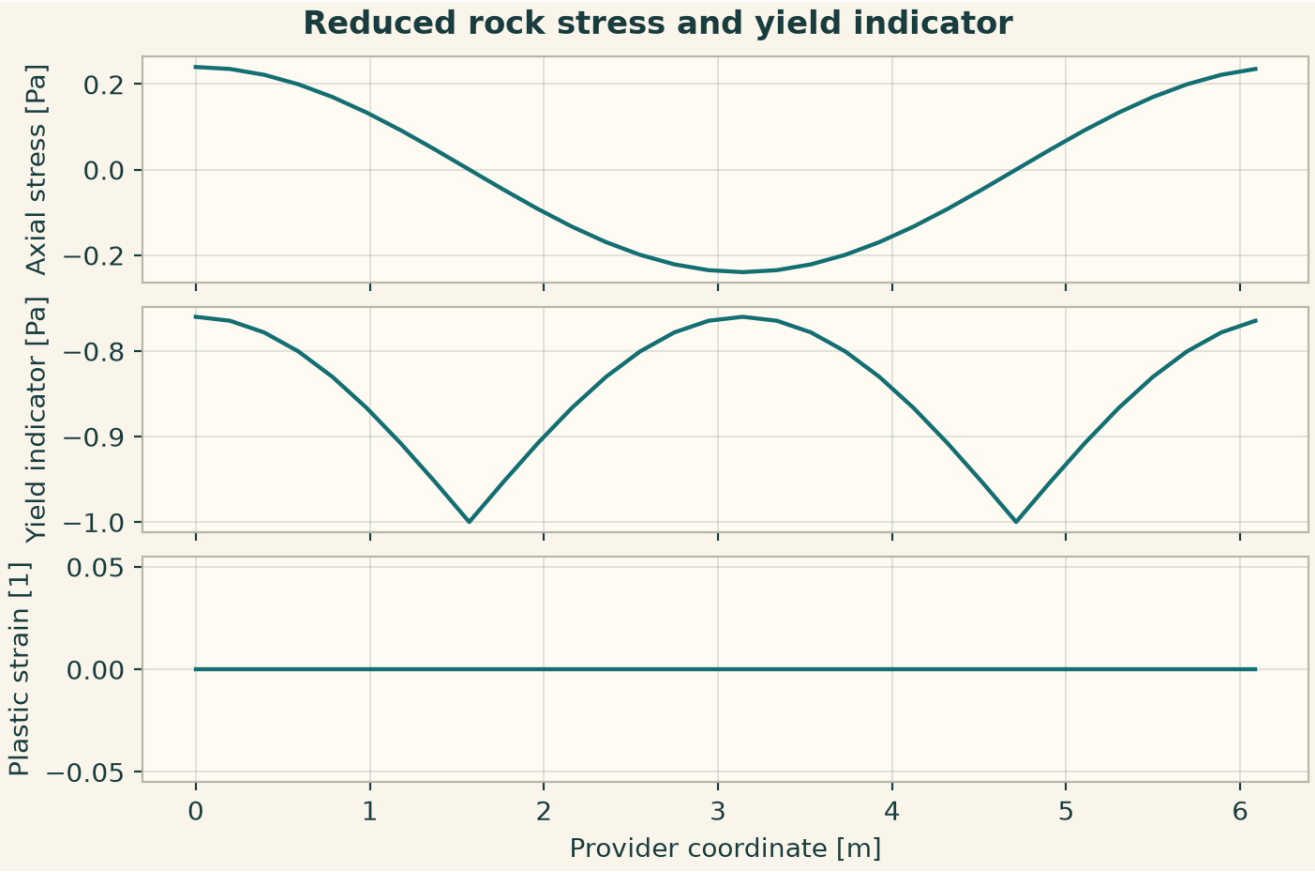
```
from onesim.physics.geotechnics import ReservoirConfiguration, ReservoirPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = ReservoirConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_pressure=1 + 0.1 * wave,
        initial_water_saturation=0.5 + 0.1 * wave,
        permeability=0.01,
    )
)
result = run_service(ReservoirPublicApi(), configuration)
fig = service_figure(
    "Two-phase reservoir pressure and saturation",
    result.coordinates,
    "Provider coordinate [m]",
    ("Pressure [Pa]", result.pressure),
    ("Water saturation [1]", result.water_saturation),
    ("Water flux [m/s]", result.water_flux),
)
plt.show()
```



6.9.7 Reduced rock stress and yield indicator

A prescribed displacement field and confinement determine the axial stress and yield indicator. Use the plastic-strain field to track material evolution under loading.

```
from onesim.physics.geotechnics import (
    RockMechanicsConfiguration,
    RockMechanicsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = RockMechanicsConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_displacement=0.02 * wave,
        confining_pressure=0.2,
        mobility=0.02,
    )
)
result = run_service(RockMechanicsPublicApi(), configuration)
fig = service_figure(
    "Reduced rock stress and yield indicator",
    result.coordinates,
    "Provider coordinate [m]",
    ("Axial stress [Pa]", result.axial_stress),
    ("Yield indicator [Pa]", result.yield_function),
    ("Plastic strain [1]", result.equivalent_plastic_strain),
)
plt.show()
```

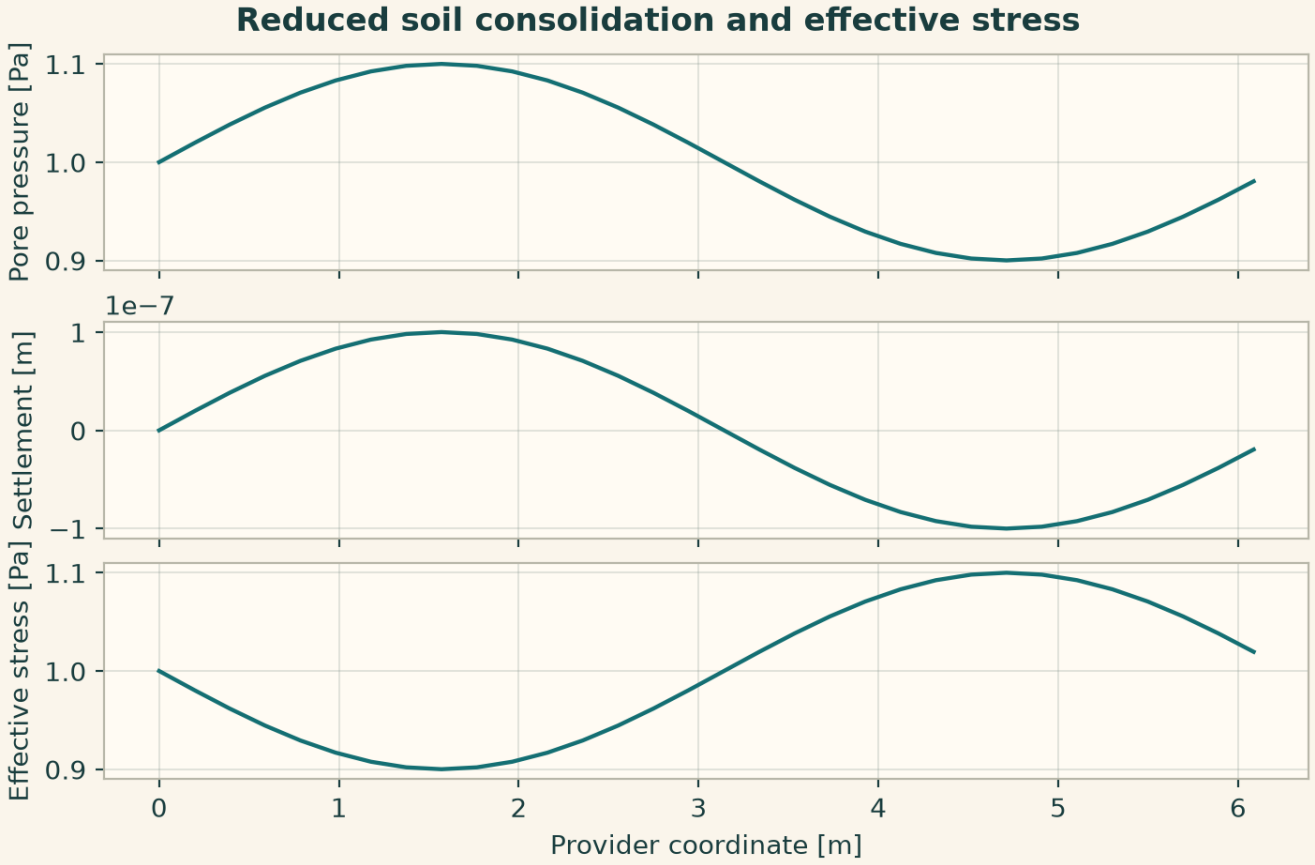


6.9.8 Reduced soil consolidation and effective stress

Initialize pore pressure and specify the total stress. The output separates pore pressure, effective stress and settlement.

```
from onesim.physics.geotechnics import (
    SoilMechanicsConfiguration,
    SoilMechanicsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = SoilMechanicsConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_pore_pressure=1 + 0.1 * wave,
        total_stress=2.0,
        permeability=0.01,
    )
)
result = run_service(SoilMechanicsPublicApi(), configuration)
fig = service_figure(
    "Reduced soil consolidation and effective stress",
    result.coordinates,
    "Provider coordinate [m]",
```

```
    ("Pore pressure [Pa]", result.pore_pressure),
    ("Settlement [m]", result.settlement),
    ("Effective stress [Pa]", result.effective_stress),
)
plt.show()
```



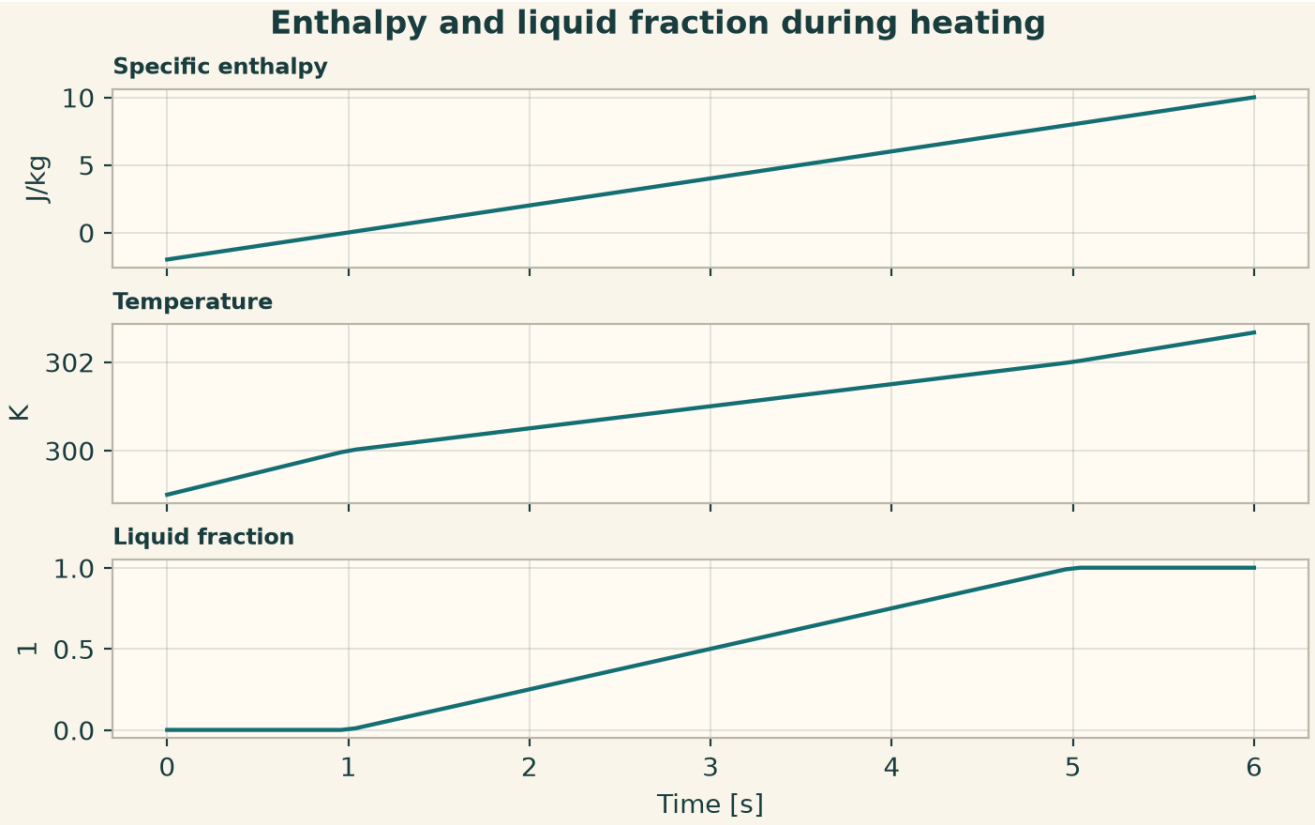
6.10 Heat transfer recipes

6.10.1 Enthalpy and liquid fraction during heating

Use specific enthalpy as the initial state and derive temperature and liquid fraction from the phase law. The mushy interval spreads the latent heat across a finite temperature range.

```
from onesim.physics.heat_transfer import PhaseChangeHeating, ThermalPhaseChangeLawSet

domain = LumpedDomain()
model = PhaseChangeHeating(
    domain=domain.system,
    mass=2,
    heating_power=4,
    phase_law=ThermalPhaseChangeLawSet("mushy", 300, 302, 4, 2, 3),
)
t, data = history_case(
    domain,
    model,
    dict(specific_enthalpy=-2.0),
    ("specific_enthalpy", "temperature", "liquid_fraction"),
```

```
    end=6,
    dt=0.02,
)
np.testing.assert_allclose(data["specific_enthalpy"], -2 + 2 * t, atol=1e-10)
fig = history_plot("Enthalpy and liquid fraction during heating", model, t, data)
plt.show()
```

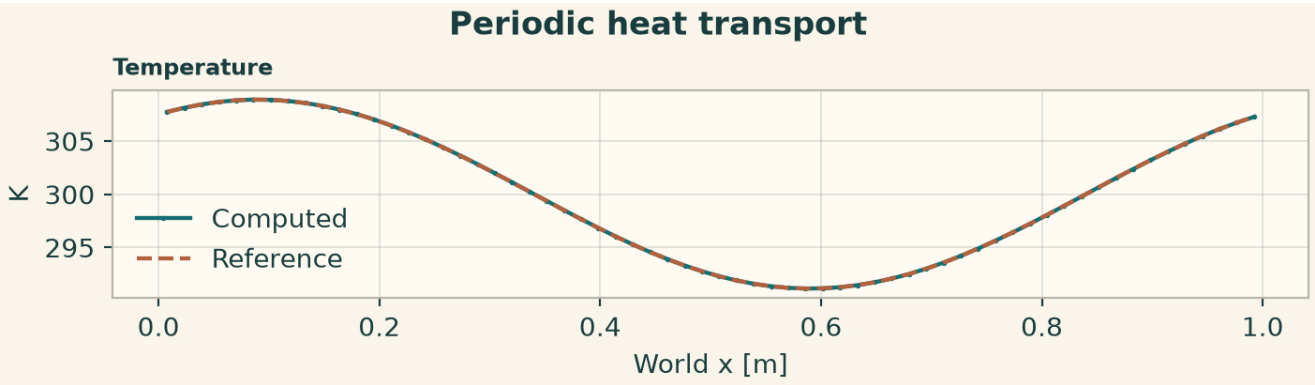


6.10.2 Advection and decay of a temperature wave

Use a periodic Fourier mode to distinguish phase transport from amplitude decay. The carrier here stores cell averages. The analytic curve includes the cell-average sinc factor.

```
from onesim.physics.heat_transfer import HeatAdvectionDiffusion

line = Line(start=(0, 0, 0), end=(1, 0, 0))
speed, diffusivity = 0.6, 0.02
model = HeatAdvectionDiffusion(
    domain=line.line, velocity=speed, thermal_diffusivity=diffusivity
)
n, end = 64, 0.15
result = line_experiment(
    line,
    model,
    dict(temperature=lambda x: 300 + 10 * np.cos(2 * np.pi * x)),
    FiniteVolume(
        convection=Rusanov(), reconstruction=MUSCL(), diffusion=TwoPointDiffusion()
    ),
    cells=n,
```

```
    end=end,
    dt=0.0005,
)
exact = lambda x: 300 + 10 * np.sinc(1 / n) * np.exp(
    -diffusivity * (2 * np.pi) ** 2 * end
) * np.cos(2 * np.pi * (x - speed * end))
field = result.field(model.temperature)
np.testing.assert_allclose(
    field.values, exact(field.space.dof_coordinates[:, 0]), atol=0.06, rtol=0
)
fig = profile_figure(
    "Periodic heat transport",
    result,
    model,
    ("temperature",),
    reference={"temperature": exact},
)
plt.show()
```



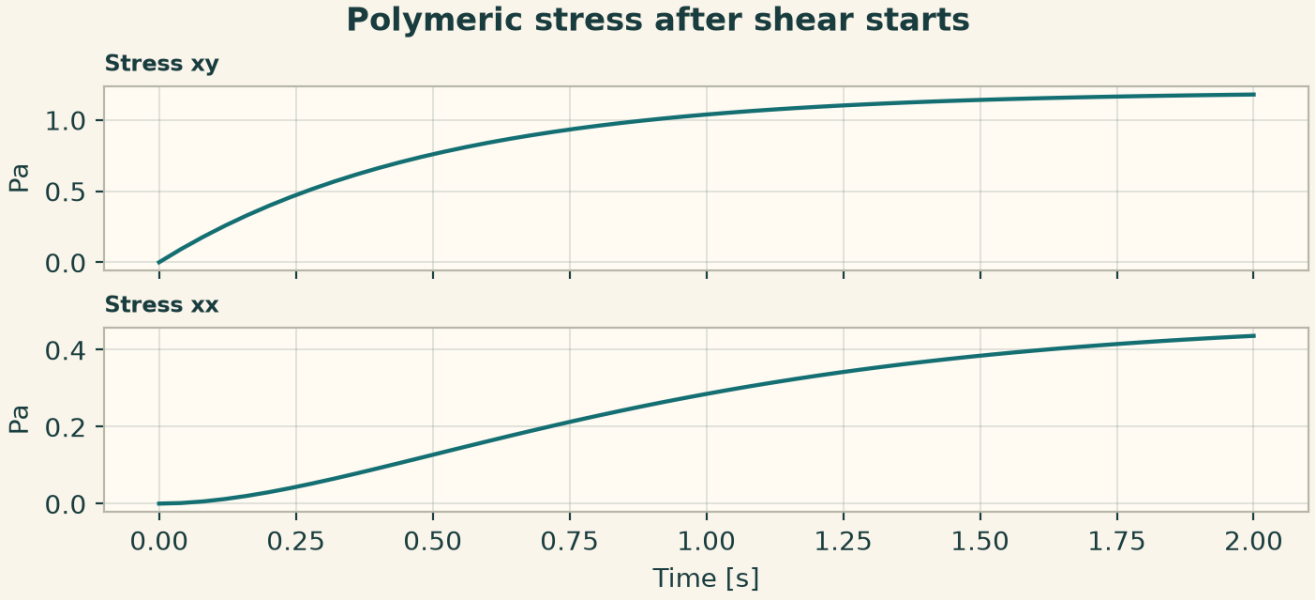
6.11 Hydrodynamics recipes

6.11.1 Polymeric stress after shear starts

Prescribe a fixed shear rate. The shear stress approaches polymer viscosity times shear rate and the normal stress develops as a separate observable.

```
from onesim.physics.hydrodynamics import OldroydBShear, ViscoelasticStressLaw

domain = LumpedDomain()
model = OldroydBShear(
    domain=domain.system,
    shear_rate=0.4,
    stress_law=ViscoelasticStressLaw(relaxation_time=0.5, polymer_viscosity=3),
)
t, data = history_case(
    domain,
    model,
    dict(stress_xx=0.0, stress_xy=0.0, stress_yy=0.0),
    ("stress_xy", "stress_xx"),
    end=2,
```

```
    dt=0.01,
)
np.testing.assert_allclose(data["stress_xy"], 1.2 * (1 - np.exp(-2 * t)), atol=1e-7)
fig = history_plot("Polymeric stress after shear starts", model, t, data)
plt.show()
```

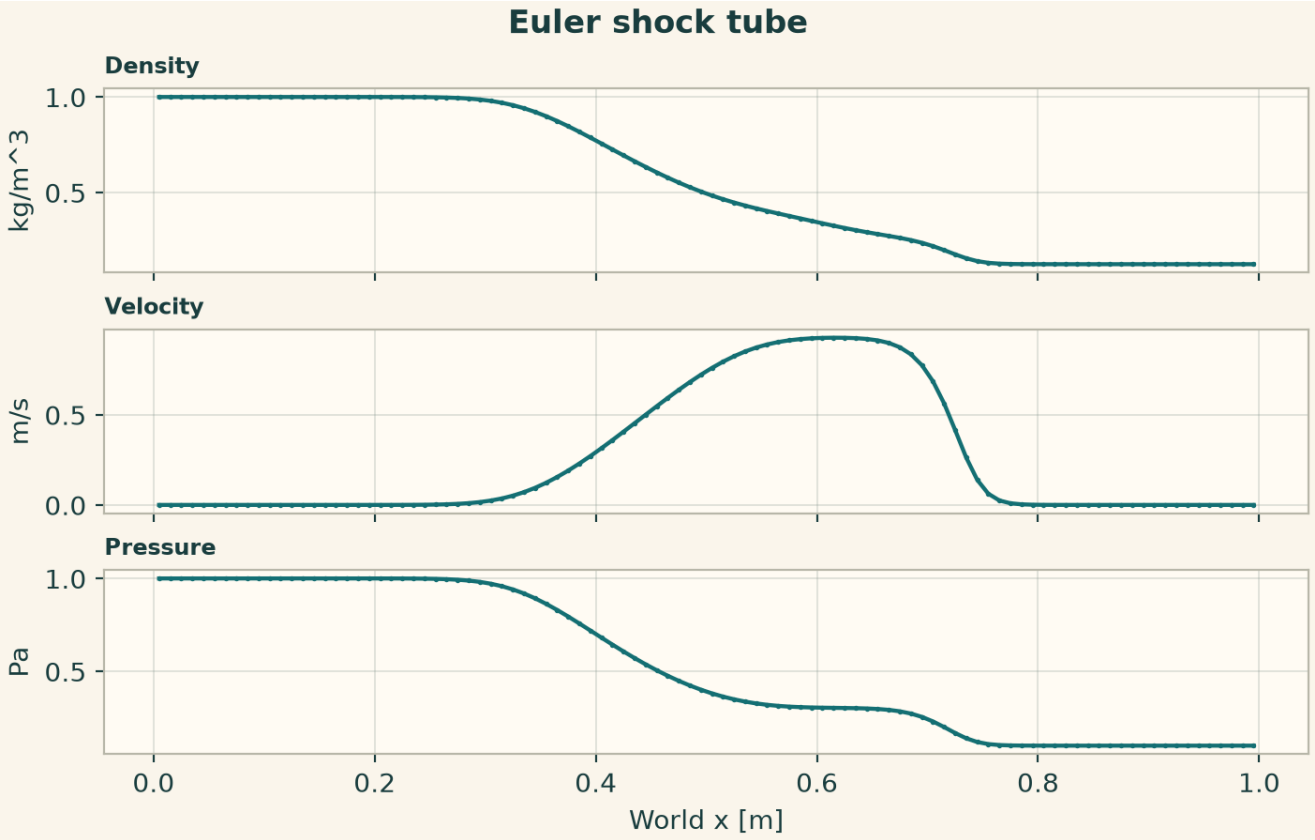


6.11.2 Compressible shock tube

Set conservative density, momentum and total energy. Use the gas law to derive pressure and require positive density and pressure in the physical result.

```
from onesim.physics.hydrodynamics import (
    CompressibleEuler,
    IdealGasEquationOfState,
    TransmissiveBoundary,
)

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = CompressibleEuler(
    domain=line.line, equation_of_state=IdealGasEquationOfState()
)
conditions = (TransmissiveBoundary(physics=model, boundary=line.boundaries.all),)
result = line_experiment(
    line,
    model,
    dict(
        density=lambda x: np.where(x < 0.5, 1.0, 0.125),
        momentum=0.0,
        total_energy=lambda x: np.where(x < 0.5, 2.5, 0.25),
    ),
    FiniteVolume(convection=Rusanov()),
    cells=100,
    end=0.12,
    dt=0.0005,
```

```
    conditions=conditions,
)
assert result.field(model.density).values.min() > 0
assert result.field(model.pressure).values.min() > 0
fig = profile_figure(
    "Euler shock tube", result, model, ("density", "velocity", "pressure")
)
plt.show()
```

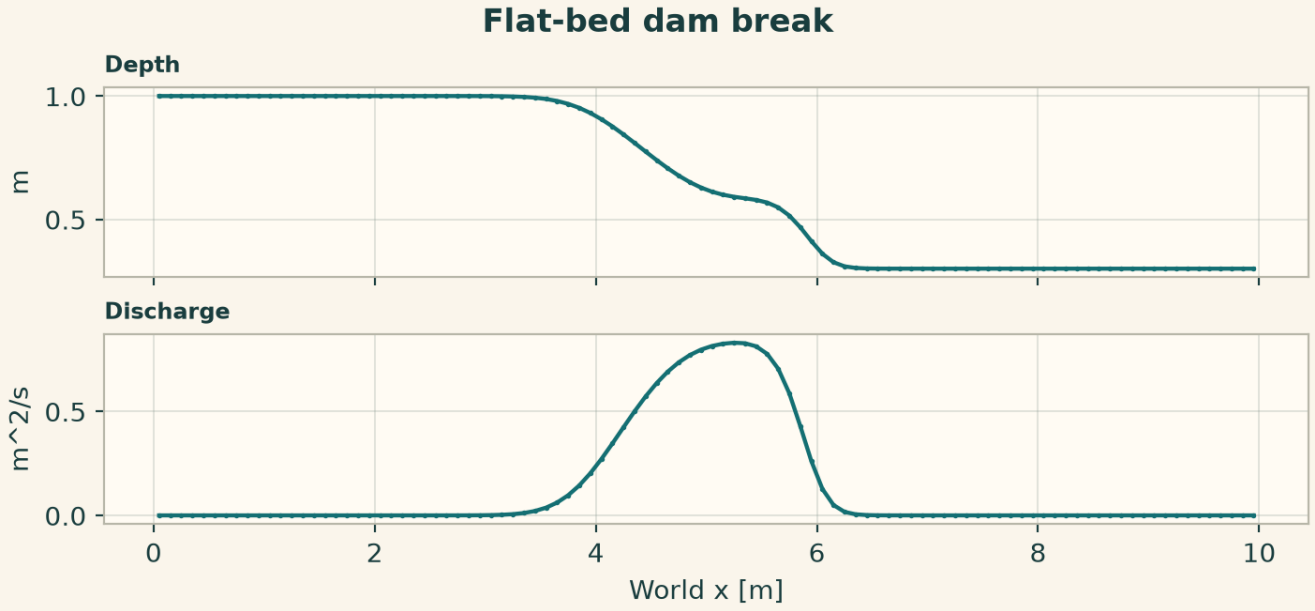


6.11.3 Shallow-water dam break on a flat bed

Specify water depth and discharge on the conservative carrier. For this short run the disturbance stays inside the domain. Check positive depth and total water volume.

```
from onesim.physics.hydrodynamics import ShallowWater
from onesim.simulation.problem_definition.boundaries import ReflectingWall

line = Line(start=(0, 0, 0), end=(10, 0, 0))
model = ShallowWater(domain=line.line)
result = line_experiment(
    line,
    model,
    dict(depth=lambda x: np.where(x < 5, 1.0, 0.3), discharge=0.0),
    FiniteVolume(convection=Rusanov()),
    cells=100,
    end=0.3,
    dt=0.001,
```

```
    conditions=(ReflectingWall(physics=model, boundary=line.boundaries.all),),
)
depth = result.field(model.depth)
assert depth.values.min() > 0
np.testing.assert_allclose(depth.integrate().value, 6.5, atol=1e-10)
fig = profile_figure("Flat-bed dam break", result, model, ("depth", "discharge"))
plt.show()
```

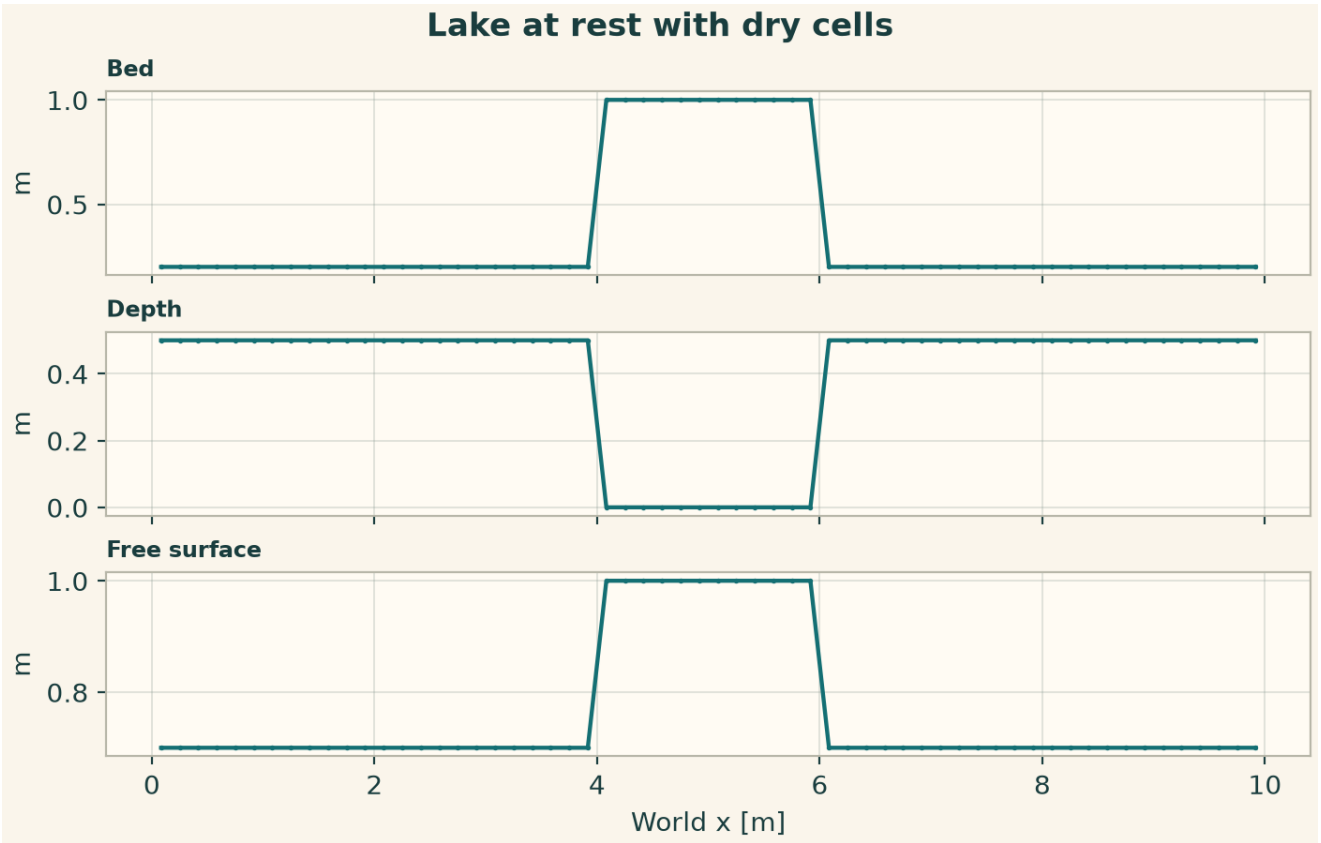


6.11.4 Well-balanced flow over an emerged bed

The initial water level lies below the bed in the middle of the domain. Use hydrostatic reconstruction to preserve the wet/dry equilibrium and check zero discharge.

```
from onesim.physics.hydrodynamics import TopographicShallowWater
from onesim.numerics.fvm import HydrostaticReconstruction
from onesim.simulation.problem_definition.boundaries import ReflectingWall

line = Line(start=(0, 0, 0), end=(10, 0, 0))
bed = lambda x: np.where((x > 4) & (x < 6), 1.0, 0.2)
depth = lambda x: np.maximum(0, 0.7 - bed(x))
model = TopographicShallowWater(domain=line.line, bed_elevation=bed)
result = line_experiment(
    line,
    model,
    dict(depth=depth, discharge=0.0),
    FiniteVolume(
        convection=Rusanov(),
        provider=HydrostaticReconstruction(),
        cell_quadrature_order=4,
    ),
    cells=60,
    end=0.12,
    dt=0.002,
    conditions=(ReflectingWall(physics=model, boundary=line.boundaries.all),),
```

```
)
np.testing.assert_allclose(result.field(model.discharge).values, 0, atol=1e-12)
fig = profile_figure(
    "Lake at rest with dry cells", result, model, ("bed", "depth", "free_surface")
)
plt.show()
```

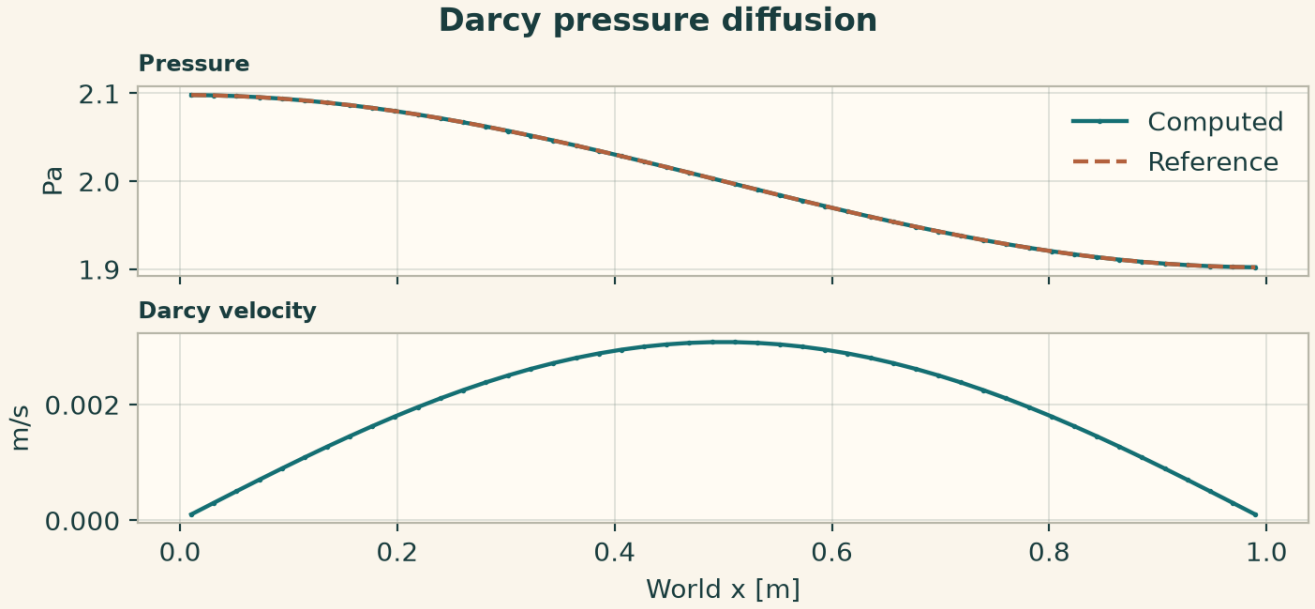


6.11.5 Pressure storage and no-flow Darcy endpoints

Use outward flux conditions at both ends. A cosine pressure profile is compatible with no flow through the walls. Darcy velocity is derived from the pressure gradient.

```
from onesim.physics.hydrodynamics import DarcyPressureFlow, DarcyFluxBoundary

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = DarcyPressureFlow(
    domain=line.line, permeability=0.01, viscosity=1, pressure_storage=1
)
conditions = tuple(
    DarcyFluxBoundary(physics=model, boundary=getattr(line.boundaries, side))
    for side in ("start", "end")
)
end = 0.2
result = line_experiment(
    line,
    model,
    dict(pressure=lambda x: 2 + 0.1 * np.cos(np.pi * x)),
```

```
    FiniteDifference(gradient=CentralDifference(), diffusion=CentralLaplacian()),
    cells=48,
    end=end,
    dt=0.001,
    conditions=conditions,
)
exact = lambda x: 2 + 0.1 * np.cos(np.pi * x) * np.exp(-0.01 * np.pi**2 * end)
p = result.field(model.pressure)
np.testing.assert_allclose(p.values, exact(p.space.dof_coordinates[:, 0]), atol=1e-4)
fig = profile_figure(
    "Darcy pressure diffusion",
    result,
    model,
    ("pressure", "darcy_velocity"),
    reference={"pressure": exact},
)
plt.show()
```

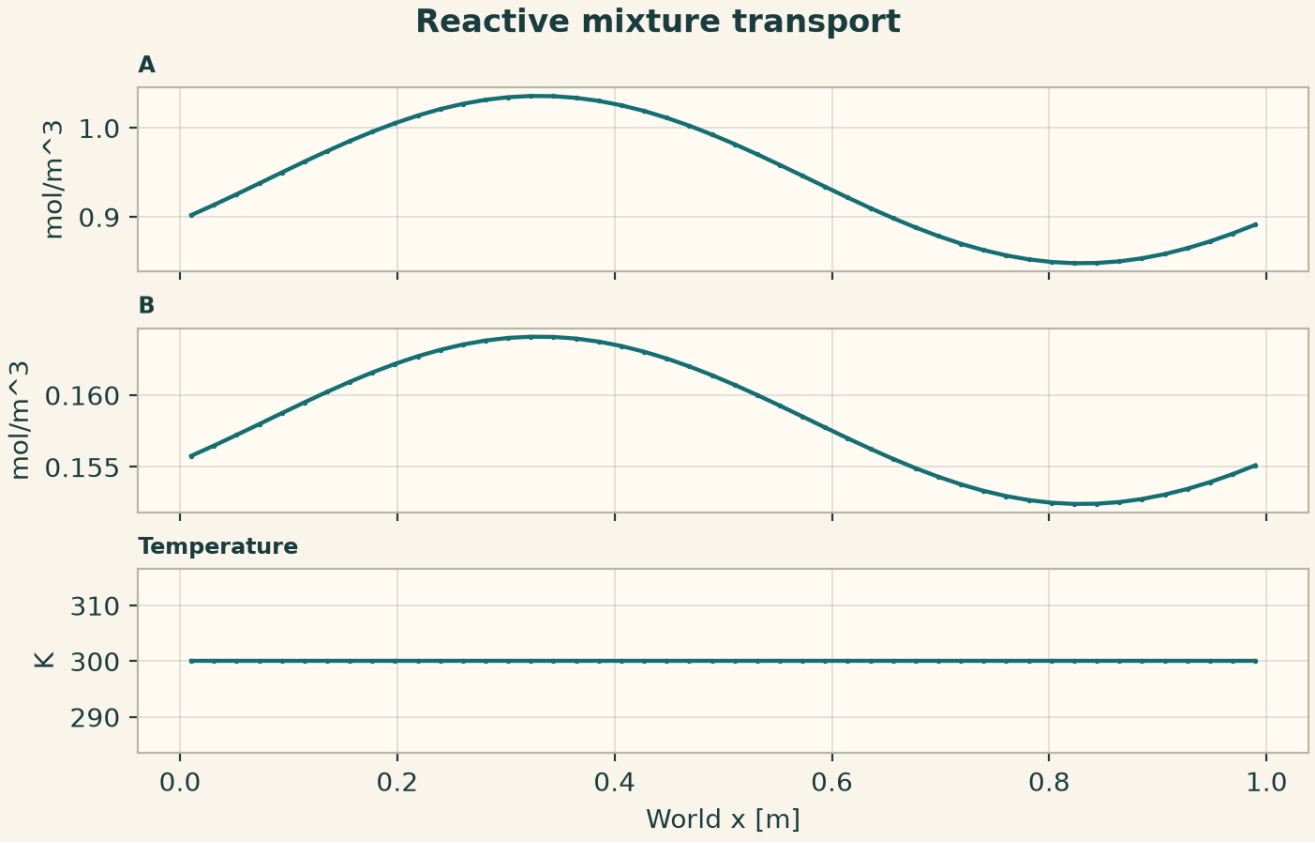


6.11.6 Advection of a reacting mixture

Share the reaction network between the transported species and temperature balances. A periodic total concentration wave translates while the reaction redistributes population between species.

```
from onesim.physics.chemistry import ReactionIR, ReactionNetworkIR, SpeciesIR
from onesim.physics.hydrodynamics import ReactiveMixtureTransport

line = Line(start=(0, 0, 0), end=(1, 0, 0))
species = tuple(
    SpeciesIR(n, 0, 1.66054e-27, composition=(("H", 1),)) for n in ("A", "B")
)
reaction = ReactionIR(
    "conversion", (("A", 1),), (("B", 1),), {"kind": "constant", "value": 0.3}
```

```
)
network = ReactionNetworkIR("advecting isomers", "1", species, (reaction,))
model = ReactiveMixtureTransport(
    domain=line.line, network=network, velocity=0.4, volumetric_heat_capacity=2.0
)
a, b = model.concentration("A").name, model.concentration("B").name
result = line_experiment(
    line,
    model,
    {a: lambda x: 1 + 0.1 * np.sin(2 * np.pi * x), b: 0.1, "temperature": 300.0},
    FiniteVolume(convection=Rusanov(), reconstruction=MUSCL()),
    cells=48,
    end=0.2,
    dt=0.001,
)
integral = sum(result.field(model.field(n)).integrate().value for n in (a, b))
np.testing.assert_allclose(integral, 1.1, atol=1e-9)
fig = profile_figure(
    "Reactive mixture transport", result, model, (a, b, "temperature")
)
plt.show()
```

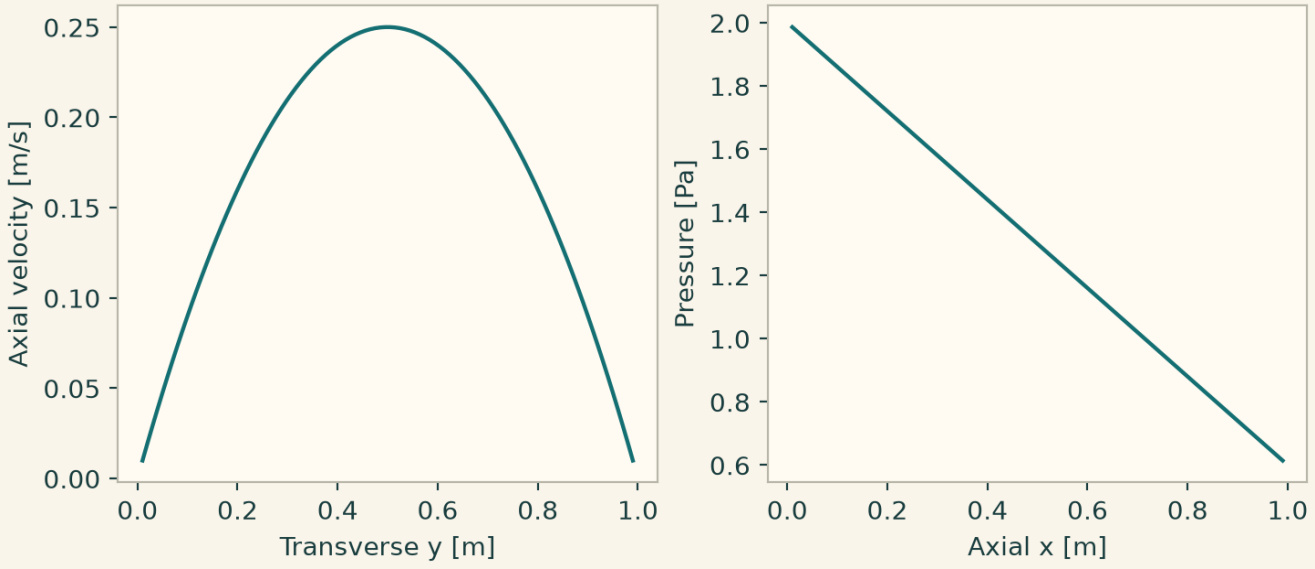


6.11.7 Laminar channel with physical traction

Prescribe the exact no-slip profile on the transverse faces, including the spanwise faces, and use the full Cauchy traction at both ends. This fixes the absolute pressure through the load and checks the mixed Taylor–Hood formulation.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import NewtonianFluid
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, TaylorHood
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.physics.hydrodynamics import (
    FixedVelocity,
    FlowTraction,
    IncompressibleNavierStokes,
)
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
mu = 0.7
flow = IncompressibleNavierStokes(
    domain=box.volume,
    material=NewtonianFluid(density=1, dynamic_viscosity=mu),
)


def velocity(y):
    return np.column_stack((y * (1 - y), np.zeros((len(y), 2))))


def inlet(y):
    return np.column_stack((2 * np.ones_like(y), -mu * (1 - 2 * y), np.zeros_like(y)))


def outlet(y):
    return np.column_stack(
        (-(2 - 2 * mu) * np.ones_like(y), mu * (1 - 2 * y), np.zeros_like(y))
    )


walls = (
    box.boundaries.y_min
    | box.boundaries.y_max
    | box.boundaries.z_min
    | box.boundaries.z_max
)
conditions = (
    FixedVelocity(physics=flow, boundary=walls, value=velocity),
    FlowTraction(physics=flow, boundary=box.boundaries.x_min, value=inlet),
    FlowTraction(physics=flow, boundary=box.boundaries.x_max, value=outlet),
)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
```

```
sim = Simulation(
    problem=Problem(
        name="Open channel",
        geometry=box,
        physics=(flow,),
        boundary_conditions=conditions,
    ),
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteElement(element=TaylorHood()),
        nonlinear_solver=Newton(
            linear_solver=SparseDirect(),
            absolute_tolerance=2e-11,
            relative_tolerance=0.0,
        ),
        initial_guess={flow.velocity: (0, 0, 0)},
    ),
)
sim.validate().require_valid()
result = sim.solve_stationary()
s = np.linspace(0.01, 0.99, 100)
cross = np.column_stack((np.full_like(s, 0.5), s, np.full_like(s, 0.5)))
axis = np.column_stack((s, np.full_like(s, 0.5), np.full_like(s, 0.5)))
u = result.field(flow.velocity).evaluate(cross)[:, 0]
p = result.field(flow.pressure).evaluate(axis)
np.testing.assert_allclose(u, s * (1 - s), atol=3e-10)
np.testing.assert_allclose(p, 2 - 2 * mu * s, atol=1e-8)
theme.use_theme()
fig, (a, b) = plt.subplots(1, 2, figsize=(7.4, 3.2), layout="constrained")
a.plot(s, u)
a.set(xlabel="Transverse y [m]", ylabel="Axial velocity [m/s]")
b.plot(s, p)
b.set(xlabel="Axial x [m]", ylabel="Pressure [Pa]")
plt.show()
```

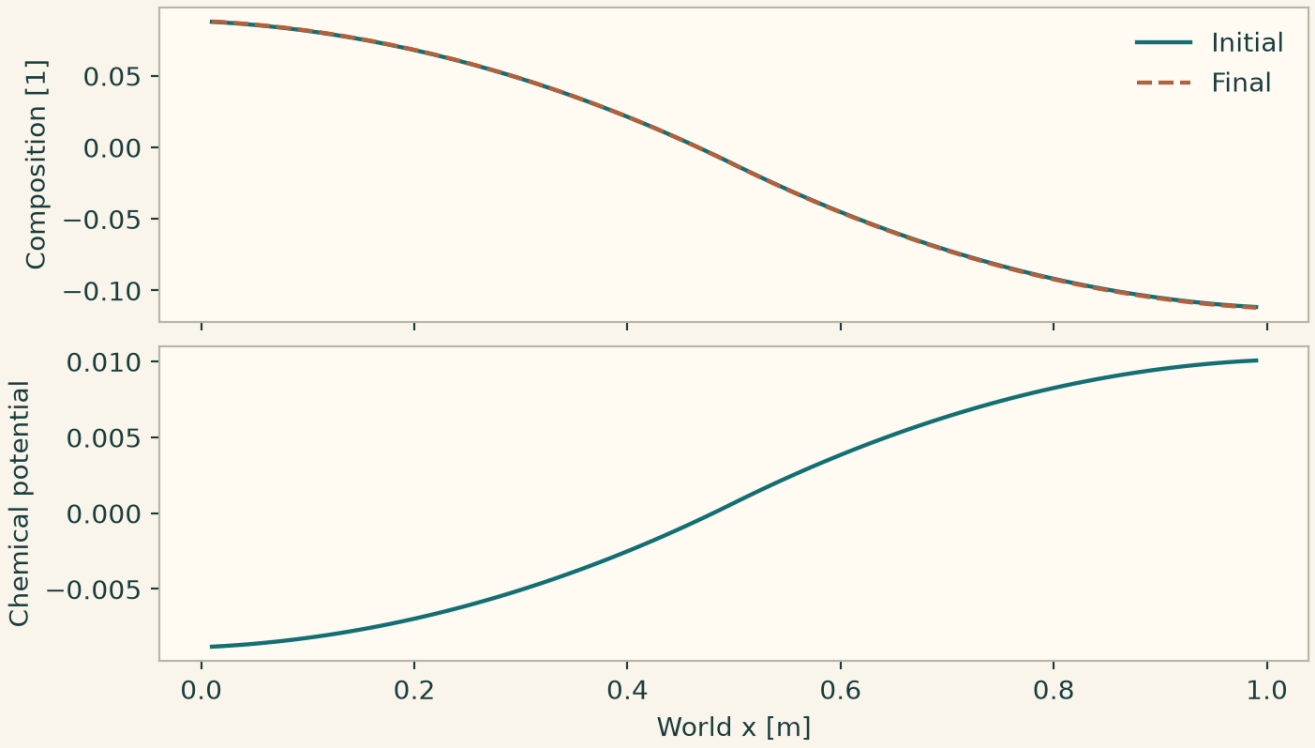


6.11.8 Mass-conserving diffuse-interface evolution

Select an interface width, surface tension and mobility. Zero outward phase flux conserves composition, while chemical potential evolves as an algebraic field coupled to the composition rate.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import (
    FiniteElement,
    GalerkinAdvection,
    GalerkinDiffusion,
    Lagrange,
)
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.hydrodynamics import (
    CahnHilliard,
    CompositionGradient,
    DoubleWellFreeEnergy,
    PhaseFlux,
)
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

box = Box(length=1, width=1, height=1)
model = CahnHilliard(
    domain=box.volume,
    free_energy=DoubleWellFreeEnergy(surface_tension=0.01, interface_width=0.1),
    mobility=0.1,
    velocity=(0.0, 0.0, 0.0),
)


def initial(x, y, z):
    return 0.1 * np.cos(np.pi * x) + 0.03 * np.cos(2 * np.pi * y)


mesh = StructuredHexahedralMesher(cells=(2, 2, 2)).generate(box)
sim = Simulation(
    problem=Problem(
        name="Diffuse interface",
        geometry=box,
        physics=(model,),
        initial_conditions=(InitialValue(model.composition, initial),),
        boundary_conditions=(
            PhaseFlux(physics=model, boundary=box.boundaries.all, value=0.0),
```

```
            CompositionGradient(
                physics=model, boundary=box.boundaries.all, value=0.0
            ),
        ),
    ),
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteElement(
            element=Lagrange(2),
            quadrature_order=5,
            convection=GalerkinAdvection(),
            diffusion=GalerkinDiffusion(),
        ),
        linear_solver=SparseDirect(),
        time_integrator=BackwardEuler(),
        nonlinear_solver=Newton(
            linear_solver=SparseDirect(),
            jacobian="analytic",
            absolute_tolerance=1e-12,
            relative_tolerance=0.0,
        ),
    ),
)
sim.validate().require_valid()
result = sim.run(start_time=0, end_time=0.04, time_step=0.01)
c = result.field(model.composition)
assert abs(c.integrate(time="last").value) < 1e-9
x = np.linspace(0.01, 0.99, 100)
points = np.column_stack((x, np.full_like(x, 0.35), np.full_like(x, 0.55)))
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
a.plot(x, c.evaluate(points, time=0), label="Initial")
a.plot(x, c.evaluate(points, time="last"), "--", label="Final")
a.set_ylabel("Composition [1]")
a.legend(frameon=False)
b.plot(x, result.field(model.chemical_potential).evaluate(points))
b.set(xlabel="World x [m]", ylabel="Chemical potential")
plt.show()
```



6.12 Magnetohydrodynamics recipes

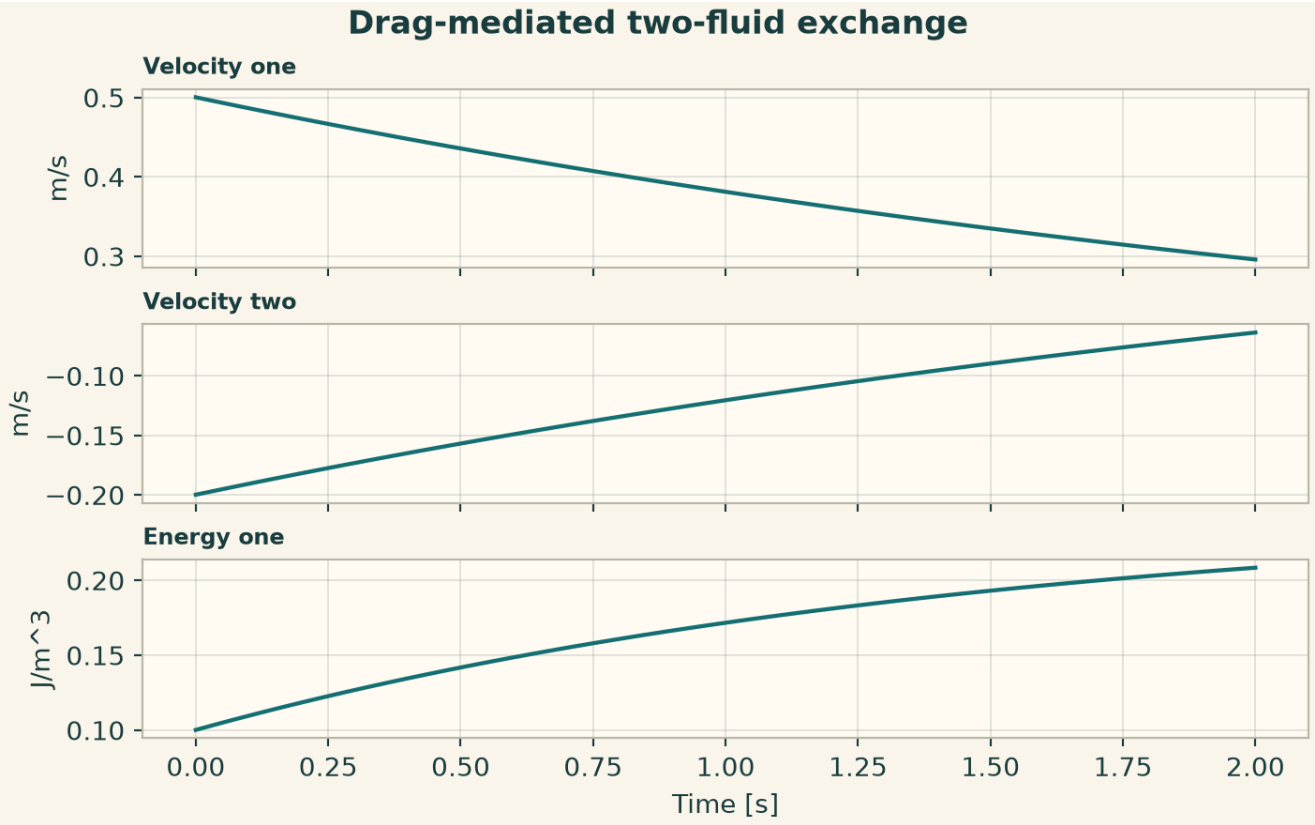
This section includes the field-based declarations and the independently callable magnetohydrodynamics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.12.1 Drag-mediated two-fluid exchange

Two homogeneous components exchange momentum through drag. Without external forces the density-weighted velocity sum stays fixed.

```
from onesim.physics.magnetohydrodynamics import TwoFluidExchange

domain = LumpedDomain()
model = TwoFluidExchange(
    domain=domain.system,
    density_one=2,
    density_two=3,
    drag_coefficient=0.4,
)
t, data = history_case(
    domain,
    model,
    dict(velocity_one=0.5, velocity_two=-0.2, energy_one=0.1, energy_two=0.1),
    ("velocity_one", "velocity_two", "energy_one"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(
    2 * data["velocity_one"] + 3 * data["velocity_two"], 0.4, atol=1e-10
)
```

```
fig = history_plot("Drag-mediated two-fluid exchange", model, t, data)
plt.show()
```

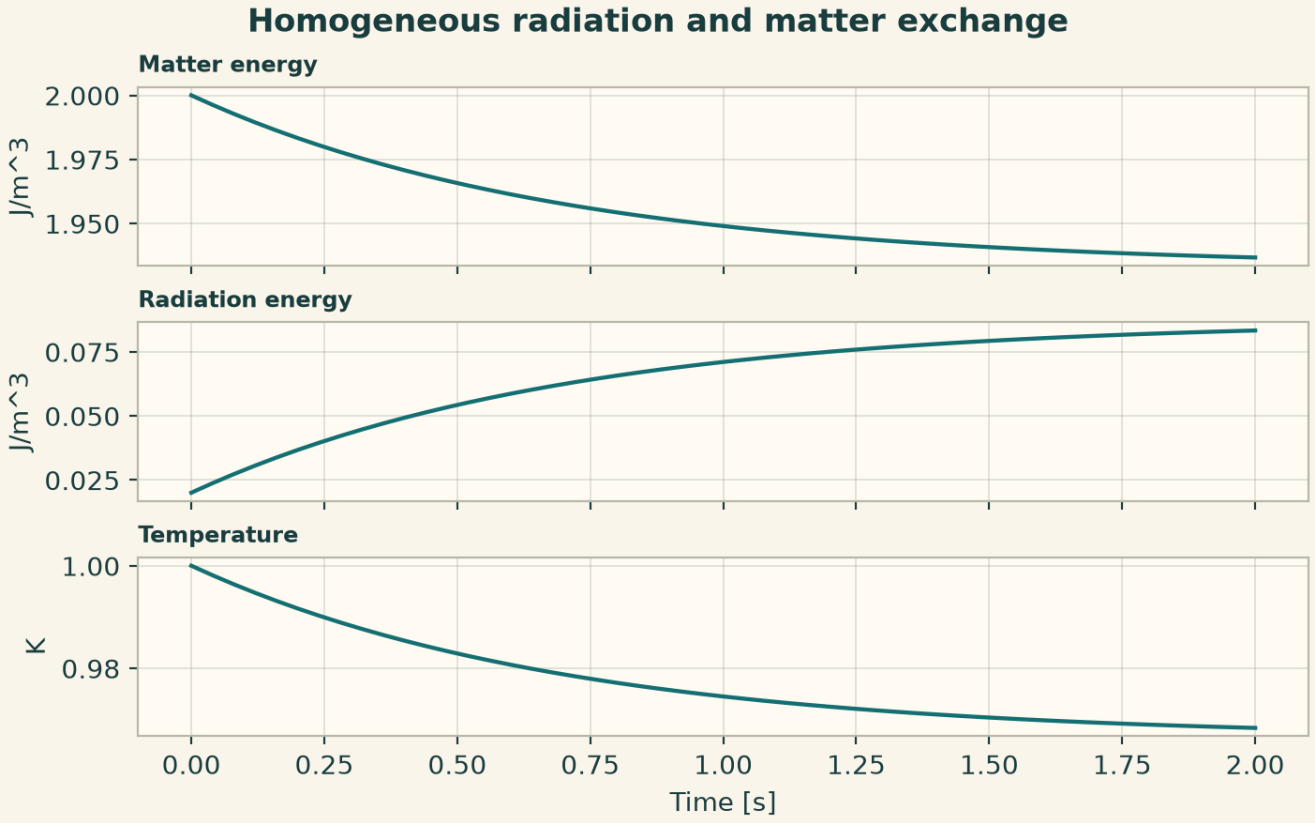


6.12.2 Homogeneous radiation and matter exchange

Initialize matter and radiation energies independently. Their sum remains constant in the isolated model and temperature follows the matter energy.

```
from onesim.physics.magnetohydrodynamics import RadiationMatterExchange

domain = LumpedDomain()
model = RadiationMatterExchange(
    domain=domain.system,
    heat_capacity=2,
    absorption=0.3,
    light_speed=4,
    radiation_constant=0.1,
)
t, data = history_case(
    domain,
    model,
    dict(matter_energy=2.0, radiation_energy=0.02),
    ("matter_energy", "radiation_energy", "temperature"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(
    data["matter_energy"] + data["radiation_energy"], 2.02, atol=1e-10
```

```
)
fig = history_plot("Homogeneous radiation and matter exchange", model, t, data)
plt.show()
```

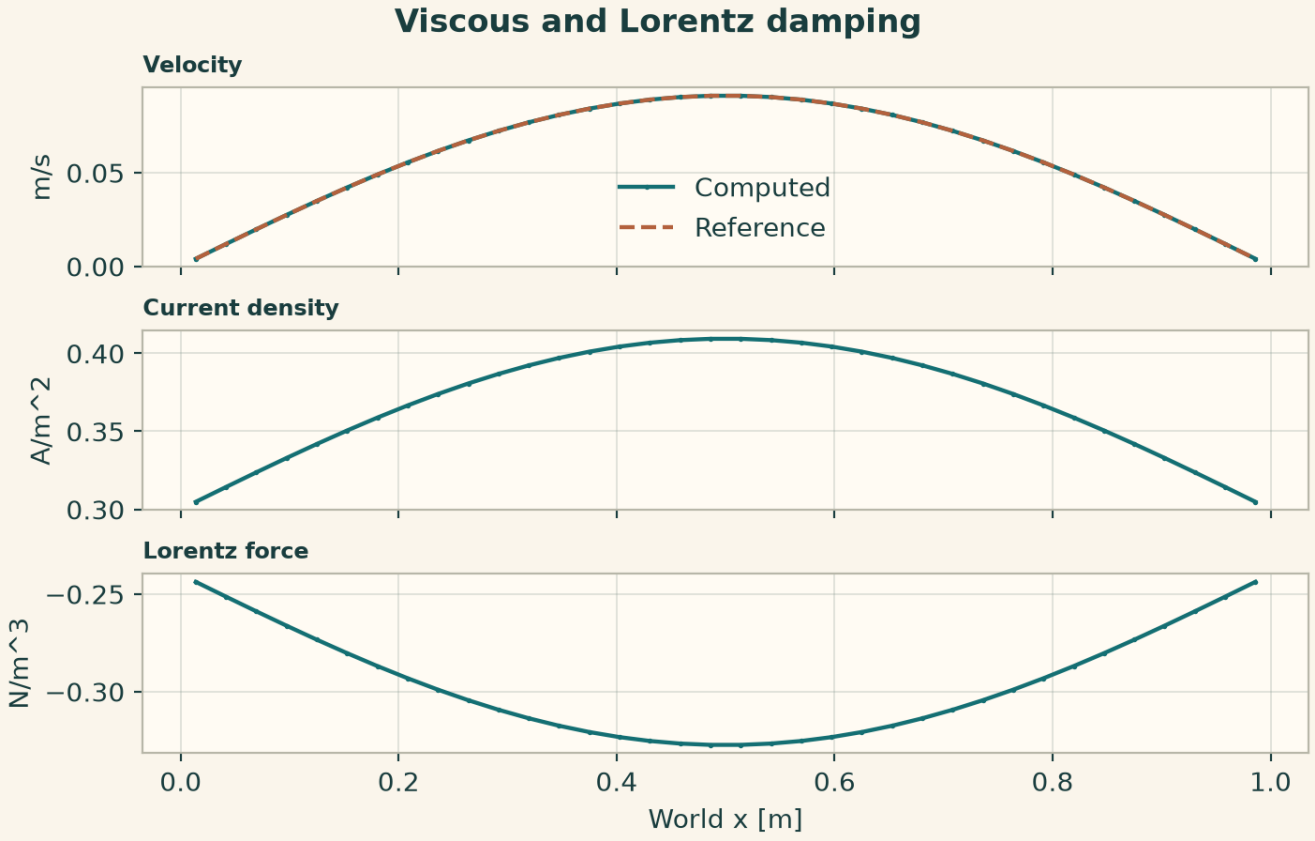


6.12.3 Decay of velocity with Lorentz damping

The selected pressure gradient and applied electric field cancel the constant part of the force. A sinusoidal profile decays through viscosity and electromagnetic drag.

```
from onesim.physics.magnetohydrodynamics import (
    InductionlessMomentum,
    InductionlessNoSlipWall,
)
from onesim.numerics.fdm import PolynomialLaplacian

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = InductionlessMomentum(
    domain=line.line,
    density=2,
    kinematic_viscosity=0.03,
    conductivity=1.5,
    magnetic_field=0.8,
    electric_field=0.2,
    pressure_gradient=0.24,
)
conditions = tuple(
    InductionlessNoSlipWall(physics=model, boundary=getattr(line.boundaries, side))
    for side in ("start", "end")
```

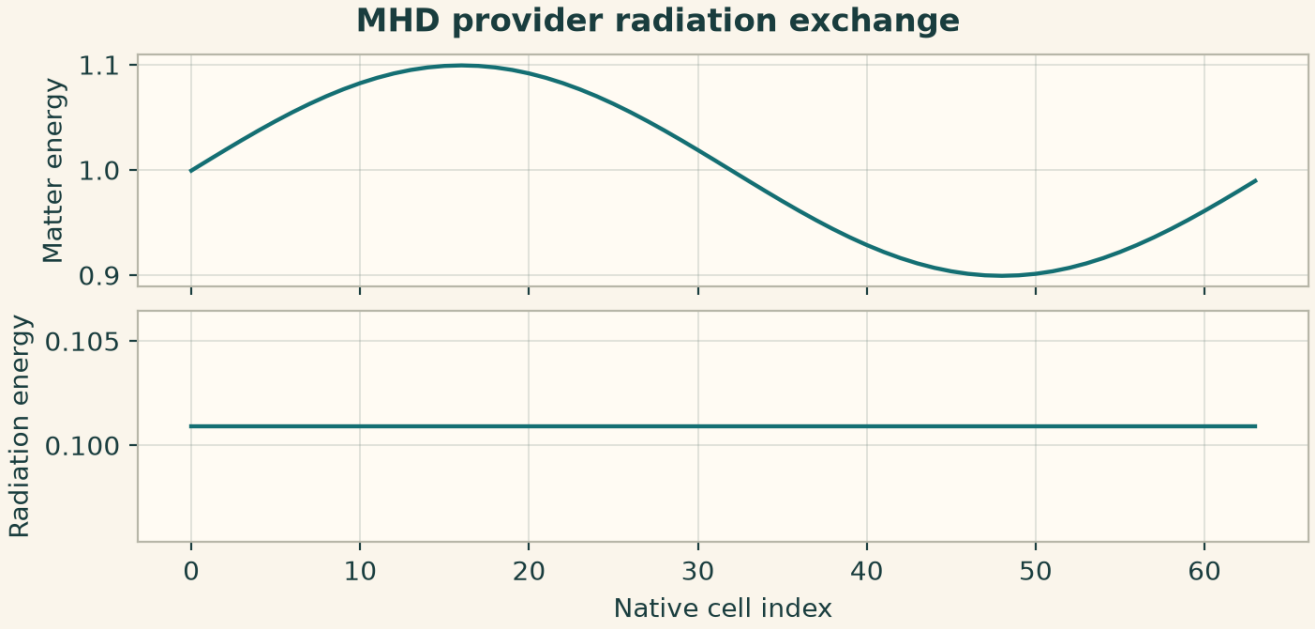
```
)
end = 0.12
result = line_experiment(
    line,
    model,
    dict(velocity=lambda x: 0.1 * np.sin(np.pi * x)),
    FiniteDifference(diffusion=PolynomialLaplacian()),
    cells=36,
    end=end,
    dt=0.0005,
    conditions=conditions,
)
u = result.field(model.velocity).values
np.testing.assert_allclose(
    result.field(model.current_density).values, 1.5 * (0.2 + 0.8 * u)
)
exact = (
    lambda x: 0.1
    * np.sin(np.pi * x)
    * np.exp(-(0.03 * np.pi**2 + 1.5 * 0.8**2 / 2) * end)
)
fig = profile_figure(
    "Viscous and Lorentz damping",
    result,
    model,
    ("velocity", "current_density", "lorentz_force"),
    reference={"velocity": exact},
)
plt.show()
```



6.12.4 Provider-selected radiation and matter exchange

Use the public provider ID to select a reduced MHD-family kernel. This example reads the provider's cell arrays and checks exchange conservation. Provider IDs advertised as pending describe extension slots.

```
from onesim.physics.magnetohydrodynamics import MagnetohydrodynamicPublicApi

api = MagnetohydrodynamicPublicApi()
key = "radiation_matter_exchange_1d"
assert key in api.provider_ids()
provider = api.create_provider(key)
result = provider.run(
    matter_energy=1.0 + 0.1 * np.sin(np.linspace(0, 2 * np.pi, 64, endpoint=False)),
    radiation_energy=np.full(64, 0.1),
)
np.testing.assert_allclose(result.total_energy_residual, 0, atol=1e-8)
fig = service_figure(
    "MHD provider radiation exchange",
    np.arange(64),
    "Native cell index",
    ("Matter energy", result.matter_energy),
    ("Radiation energy", result.radiation_energy),
)
plt.show()
```



6.12.5 The provider catalogue and its execution status

The registry advertises 74 provider IDs. This checkout marks 27 implemented, 1 partial and 46 pending. The galleries below exercise the implemented entries and the executable reduced relativistic variant. Each panel plots a real result array with its own scale. The shared conservative seed is in the order density, two momenta, total energy and two magnetic components. It uses a ratio of specific heats of 1.4. The routes with their own default state use that state and expose the selected native diagnostic.

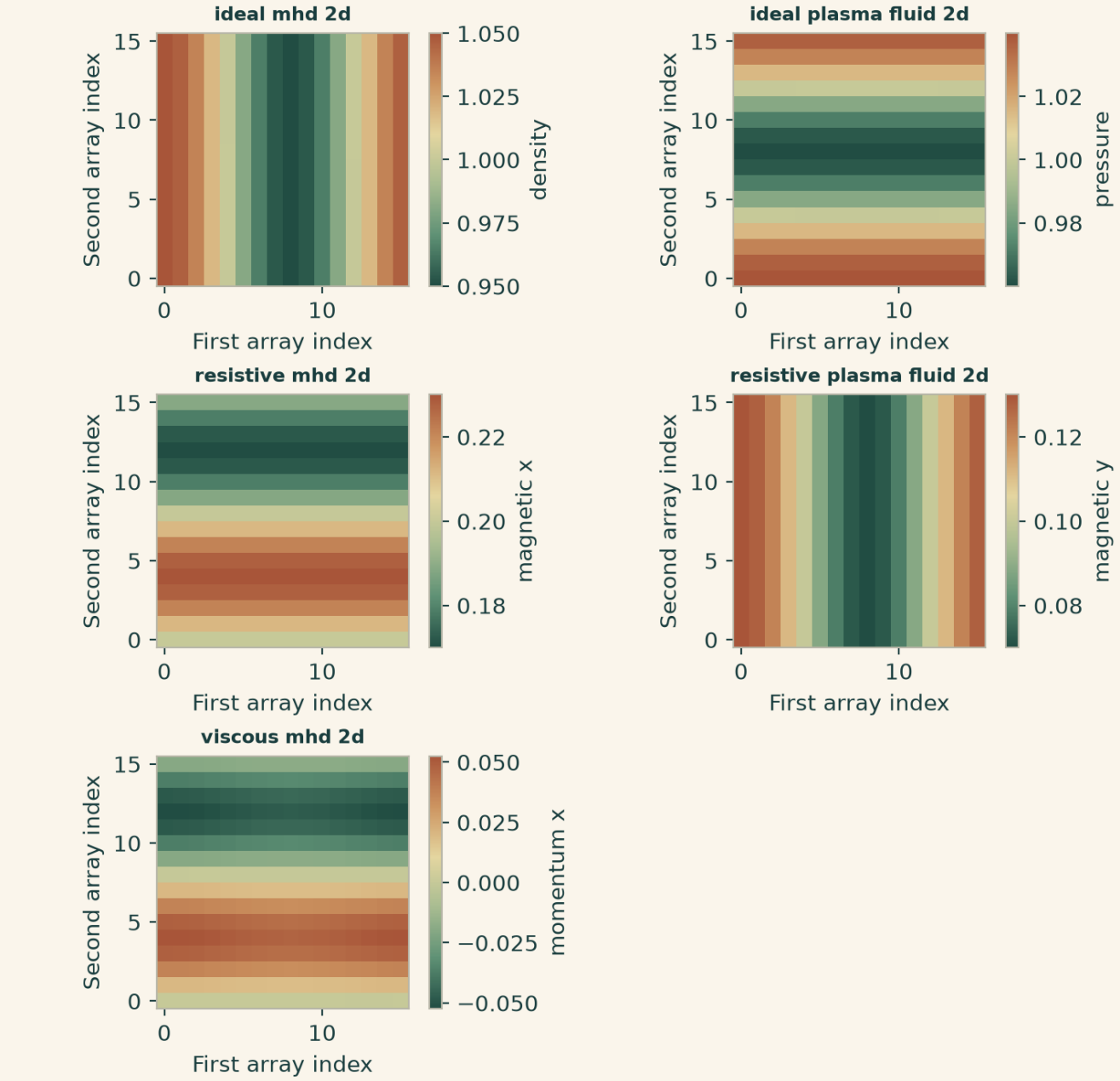
```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.physics.magnetohydrodynamics import MagnetohydrodynamicPublicApi


def conservative_mhd_seed(n):
    x, y = np.meshgrid(
        np.linspace(0, 1, n, endpoint=False),
        np.linspace(0, 1, n, endpoint=False),
        indexing="ij",
    )
    density = 1 + 0.05 * np.cos(2 * np.pi * x)
    pressure = 1 + 0.04 * np.cos(2 * np.pi * y)
    mx = density * 0.05 * np.sin(2 * np.pi * y)
    my = density * 0.03 * np.cos(2 * np.pi * x)
    bx = 0.2 + 0.03 * np.sin(2 * np.pi * y)
    by = 0.1 + 0.03 * np.cos(2 * np.pi * x)
    energy = (
        pressure / 0.4 + (mx * mx + my * my) / (2 * density) + (bx * bx + by * by) / 2
    )
    return density, mx, my, energy, bx, by


def provider_gallery(quantities, *, seeded=False):
```

```
    theme.use_theme()
    rows = (len(quantities) + 1) // 2
    fig, axes = plt.subplots(
        rows, 2, squeeze=False, figsize=(7.3, 2.35 * rows), layout="constrained"
    )
    for ax, (key, quantity) in zip(axes.ravel(), quantities.items()):
        if seeded:
            p = MagnetohydrodynamicPublicApi.create_provider(
                key, mesh_size=16, final_time=0.002, time_step=0.0002
            )
            result = p.run(initial_state=conservative_mhd_seed(16))
        else:
            p = MagnetohydrodynamicPublicApi.create_provider(key)
            result = p.run()
        values = np.asarray(getattr(result, quantity))
        assert values.ndim in (1, 2, 3) and np.isfinite(values).all()
        label = quantity.replace("_", " ")
        if values.ndim == 3:
            values = values[:, :, values.shape[2] // 2]
            label += " (mid-plane)"
        if values.ndim == 1:
            ax.plot(np.arange(values.size), values)
            ax.set(xlabel="Native cell index", ylabel=label)
            ax.grid(alpha=0.25)
        else:
            image = ax.imshow(
                values.T, origin="lower", cmap=theme.FIELD, interpolation="nearest"
            )
            ax.set(xlabel="First array index", ylabel="Second array index")
            fig.colorbar(image, ax=ax, label=label)
        ax.set_title(key.replace("_", " "), fontsize=9)
    for ax in axes.ravel()[len(quantities) :]:
        ax.set_visible(False)
    return fig
```

6.12.6 Ideal, resistive and viscous MHD variants

Use the same six-component periodic seed for the ideal and resistive routes, their plasma-fluid entry points, and viscous MHD. The initial total energy includes the kinetic and magnetic parts.

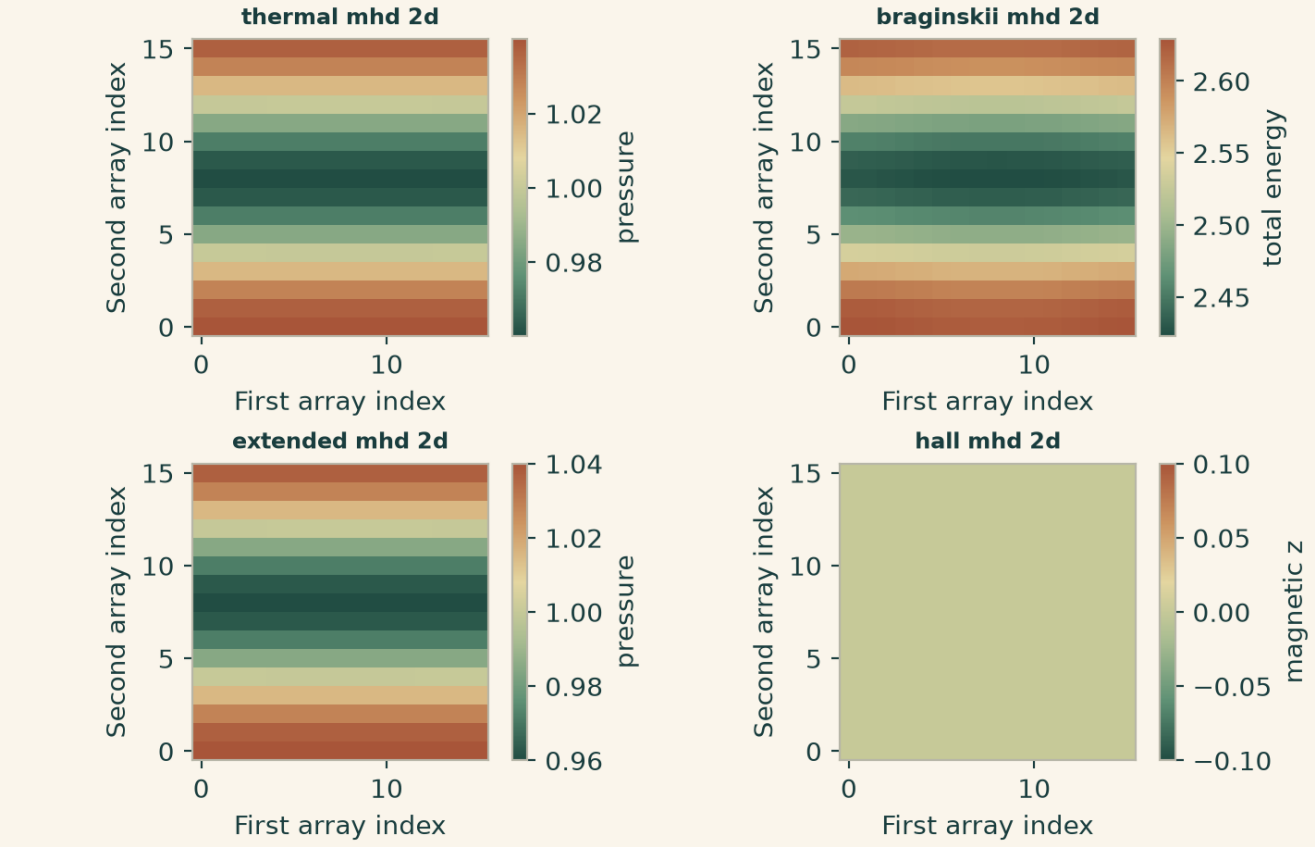
```
quantities = {
    "ideal_mhd_2d": "density",
    "ideal_plasma_fluid_2d": "pressure",
    "resistive_mhd_2d": "magnetic_x",
    "resistive_plasma_fluid_2d": "magnetic_y",
    "viscous_mhd_2d": "momentum_x",
}
fig = provider_gallery(quantities, seeded=True)
plt.show()
```



6.12.7 Thermal, anisotropic, extended and Hall variants

The common seed contains transverse magnetic and momentum variation. The Hall provider exposes a separate magnetic result. Select the named quantity from each result rather than assuming identical layouts.

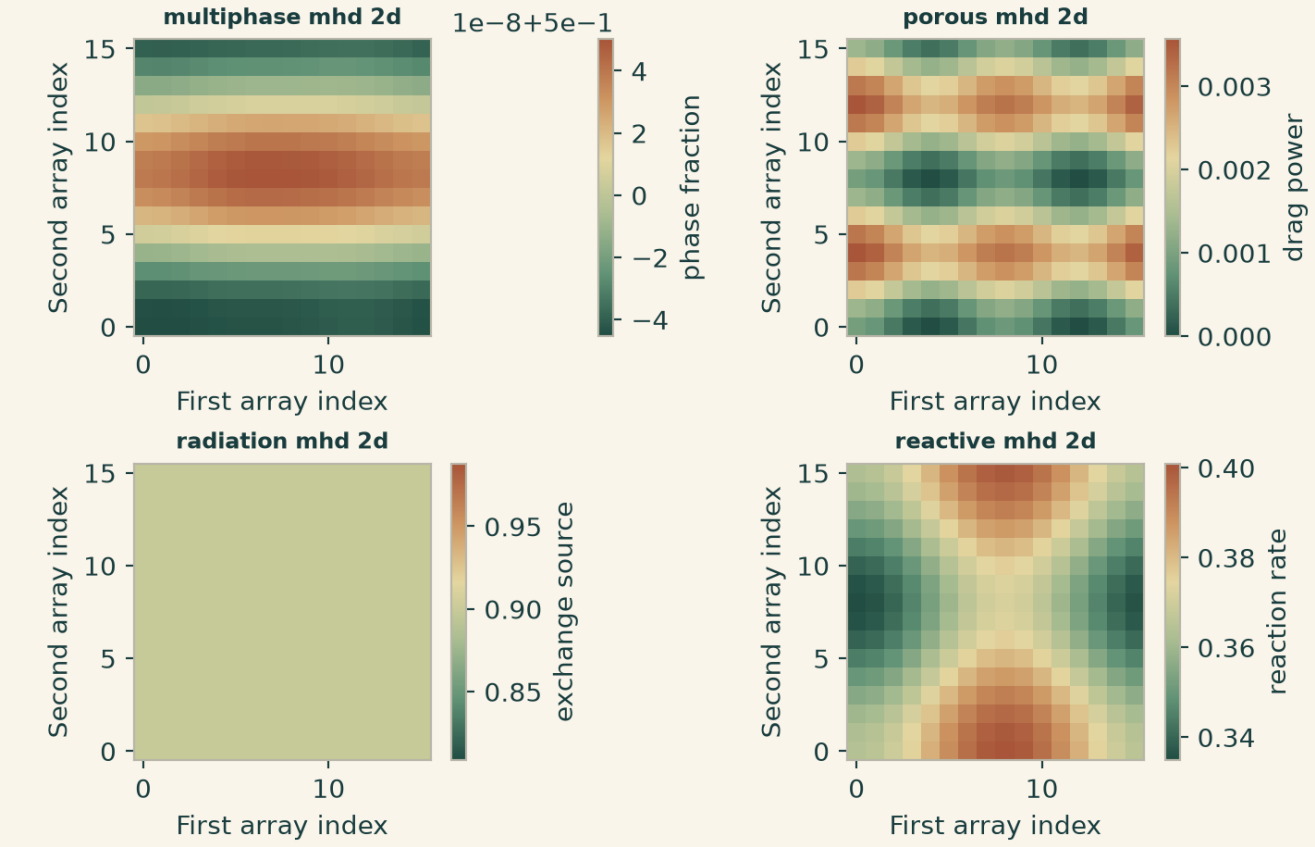
```
quantities = {
    "thermal_mhd_2d": "pressure",
    "braginskii_mhd_2d": "total_energy",
    "extended_mhd_2d": "pressure",
    "hall_mhd_2d": "magnetic_z",
}
fig = provider_gallery(quantities, seeded=True)
plt.show()
```



6.12.8 MHD with phases, porous drag, radiation and reactions

Each provider adds diagnostics owned by its closure. The short smoke cases share a fluid state and use the respective provider's defaults for its additional closure parameters.

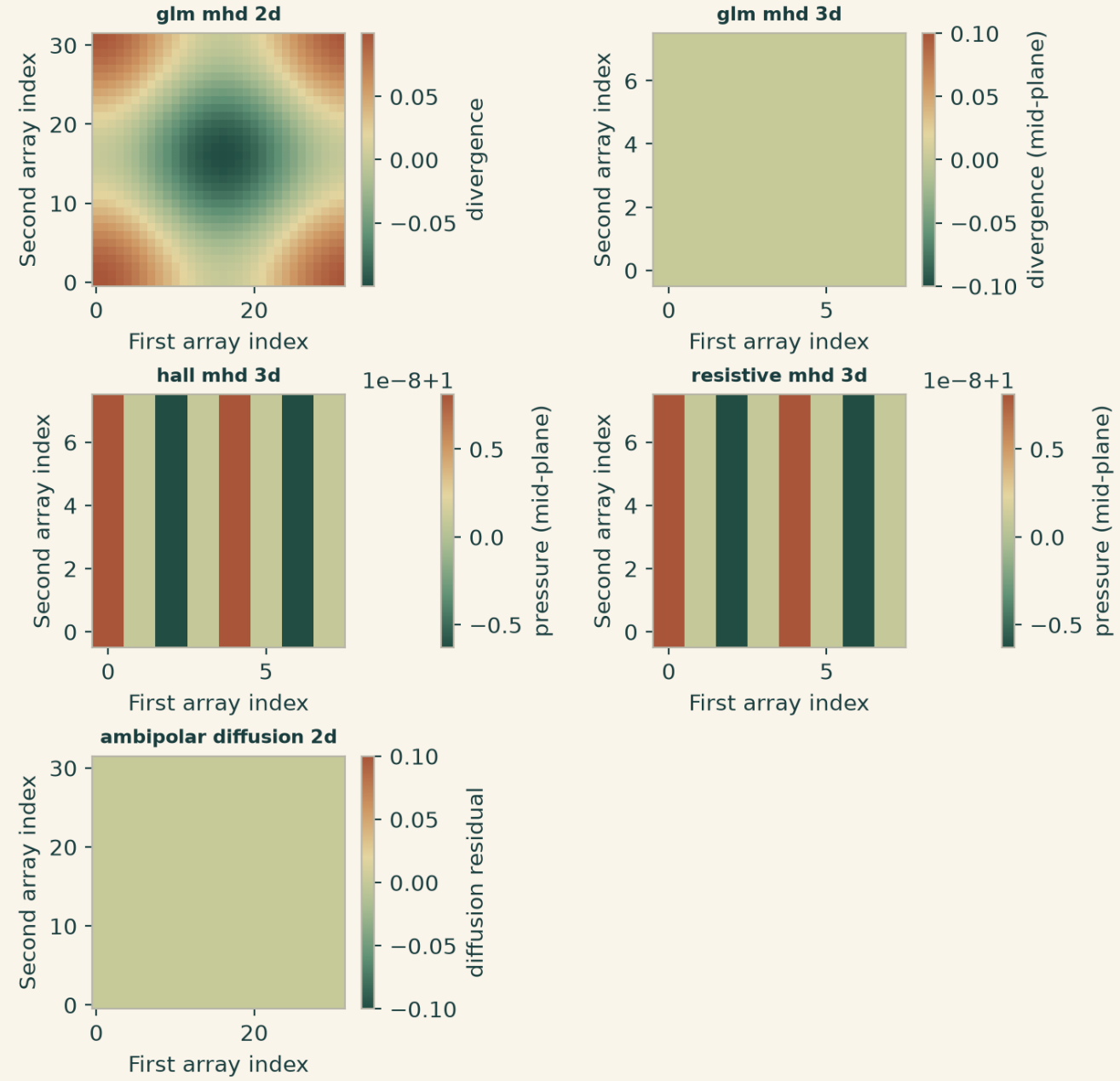
```
quantities = {
    "multiphase_mhd_2d": "phase_fraction",
    "porous_mhd_2d": "drag_power",
    "radiation_mhd_2d": "exchange_source",
    "reactive_mhd_2d": "reaction_rate",
}
fig = provider_gallery(quantities, seeded=True)
plt.show()
```



6.12.9 Divergence cleaning and three-dimensional induction

These providers define their own default initial state. Inspect divergence and pressure on native slices, with independent color scales for distinct fields.

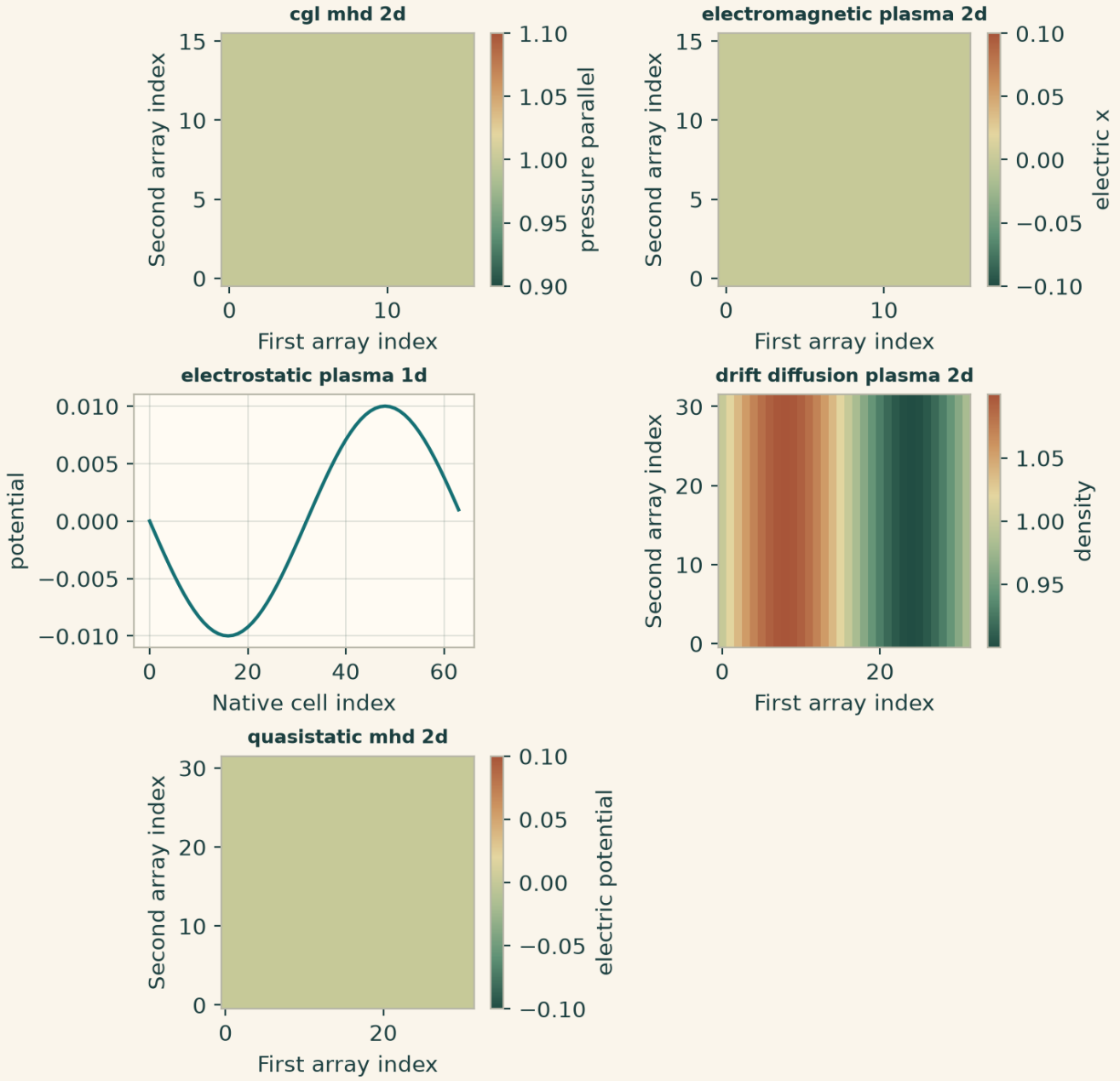
```
quantities = {
    "glm_mhd_2d": "divergence",
    "glm_mhd_3d": "divergence",
    "hall_mhd_3d": "pressure",
    "resistive_mhd_3d": "pressure",
    "ambipolar_diffusion_2d": "diffusion_residual",
}
fig = provider_gallery(quantities, seeded=False)
plt.show()
```



6.12.10 Anisotropic pressure and electromagnetic plasma providers

The CGL, electrostatic, electromagnetic, drift-diffusion and quasistatic routes expose different diagnostic carriers. The example uses each route's built-in demonstration state.

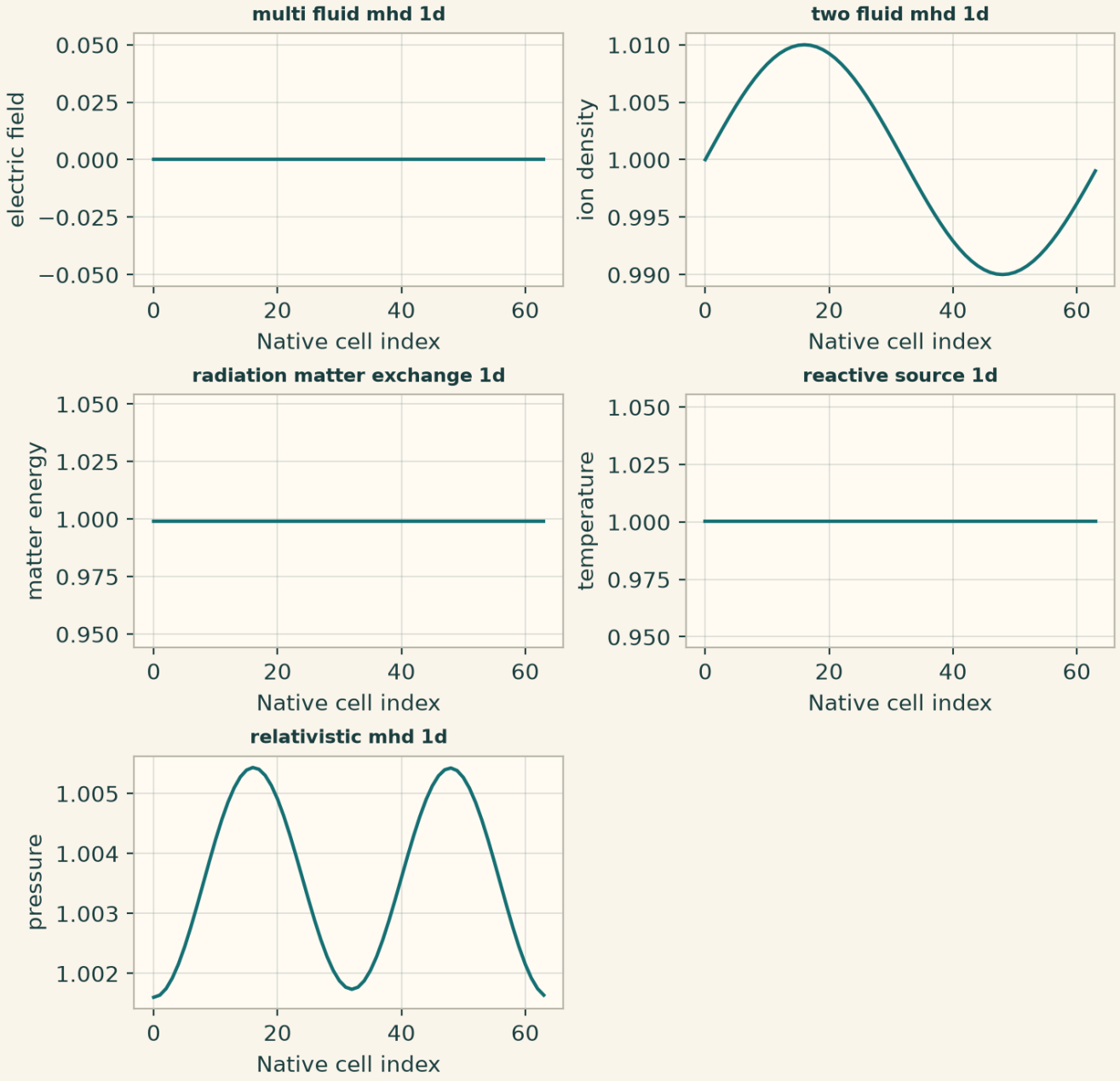
```
quantities = {
    "cgl_mhd_2d": "pressure_parallel",
    "electromagnetic_plasma_2d": "electric_x",
    "electrostatic_plasma_1d": "potential",
    "drift_diffusion_plasma_2d": "density",
    "quasistatic_mhd_2d": "electric_potential",
}
fig = provider_gallery(quantities, seeded=False)
plt.show()
```



6.12.11 One-dimensional multi-fluid, radiation and reactive kernels

Use the reduced one-dimensional variants for direct result inspection. The catalogue marks the relativistic route as partial. The plotted pressure is from that executable reduced case.

```
quantities = {
    "multi_fluid_mhd_1d": "electric_field",
    "two_fluid_mhd_1d": "ion_density",
    "radiation_matter_exchange_1d": "matter_energy",
    "reactive_source_1d": "temperature",
    "relativistic_mhd_1d": "pressure",
}
fig = provider_gallery(quantities, seeded=False)
plt.show()
```



6.12.12 Pending provider identifiers

The following registry entries have pending status in the inspected checkout. The executable MHD galleries use the implemented or partial entries described above.

`pending_ale`, `pending_bingham_fluid`, `pending_boussinesq`, `pending_brinkman_flow`, `pending_carreau_fluid`, `pending_combustion_flow`, `pending_compressible_navier_stokes`, `pending_cross_fluid`, `pending_cut_cell`, `pending_darcy_brinkman_flow`, `pending_darcy_flow`, `pending_dpd`, `pending_dual_permeability`, `pending_dual_porosity`, `pending_embedded_boundary`, `pending_euler_euler_multiphase`, `pending_euler_lagrange_multiphase`, `pending_forchheimer_flow`, `pending_green_naghdi`, `pending_herschel_bulkley_fluid`, `pending_hybrid_kinetic_fluid`, `pending_hydrodynamics_euler`, `pending_immersed_boundary`, `pending_incompressible_navier_stokes`, `pending_lbm`, `pending_level_set`, `pending_low_mach_reactive_flow`, `pending_lubrication_flow`, `pending_maxwell_fluid`, `pending_mixture_model`, `pending_multispecies_reactive_flow`, `pending_nonlinear_shallow_water`, `pending_nonpremixed_flame`, `pending_oldroyd_b_fluid`, `pending_phase_field`, `pending_potential_flow`, `pending_power_law_fluid`, `pending_premixed_flame`, `pending_richards_flow`, `pending_saint_venant`, `pending_shallow_water`, `pending_sph`,

`pending_stokes_flow`, `pending_viscoelastic_fluid`, `pending_viscous_hydrodynamics`, `pending_vof`.

6.13 Magnetostatics recipes

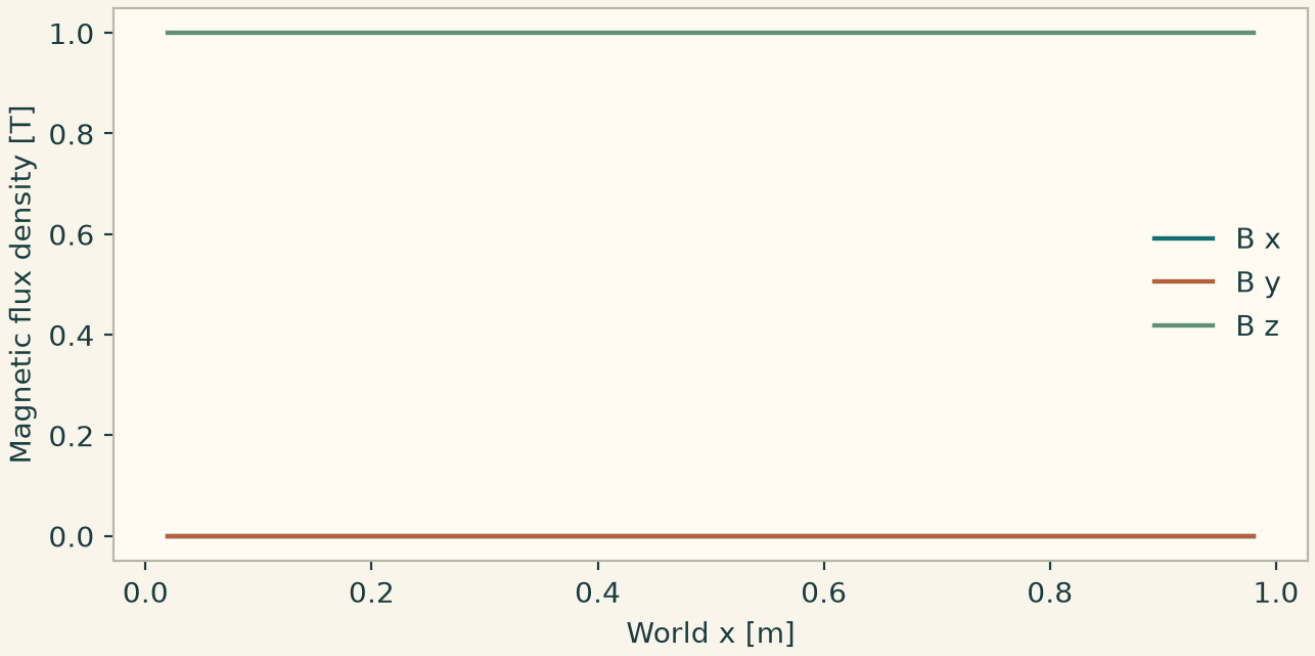
The solenoid chapter exercises the current source, embedded winding support and magnetic insulation. The manufactured trace below isolates the tangential condition and the gauge from winding geometry.

6.13.1 Imposed tangential potential and a uniform magnetic field

Use a known divergence-free vector potential as tangential trace. The resulting flux density is the curl of that field. This example exercises a nonzero trace and the Coulomb constraint.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import VacuumPermeability
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Nedelec
from onesim.numerics.linear import SparseDirect
from onesim.physics.magnetostatics import (
    CoulombGauge,
    MagneticPotentialBoundary,
    MagneticVectorPotential,
    Magnetostatics,
)
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
model = Magnetostatics(
    domain=box.volume,
    permeability=VacuumPermeability(),
    formulation=MagneticVectorPotential(
        gauge=CoulombGauge(multiplier_reference_dof=0)
    ),
)


def vector_potential(x, y, z):
    return np.column_stack((-0.5 * y, 0.5 * x, np.zeros_like(x)))


problem = Problem(
    name="Uniform B patch",
    geometry=box,
    physics=(model,),
    boundary_conditions=(
        MagneticPotentialBoundary(
            physics=model, boundary=box.boundaries.all, value=vector_potential
        ),
```

```
    ),
)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
result = Simulation(
    problem=problem,
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteElement(element=Nedelec(1)), linear_solver=SparseDirect()
    ),
).solve_stationary()
x = np.linspace(0.02, 0.98, 101)
p = np.column_stack((x, np.full_like(x, 0.37), np.full_like(x, 0.51)))
B = result.field(model.magnetic_flux_density).evaluate(p)
np.testing.assert_allclose(B, np.broadcast_to((0.0, 0.0, 1.0), B.shape), atol=1e-8)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.6, 3.3), layout="constrained")
for i, d in enumerate("xyz"):
    ax.plot(x, B[:, i], label=f"B {d}")
ax.set(xlabel="World x [m]", ylabel="Magnetic flux density [T]")
ax.legend(frameon=False)
plt.show()
```



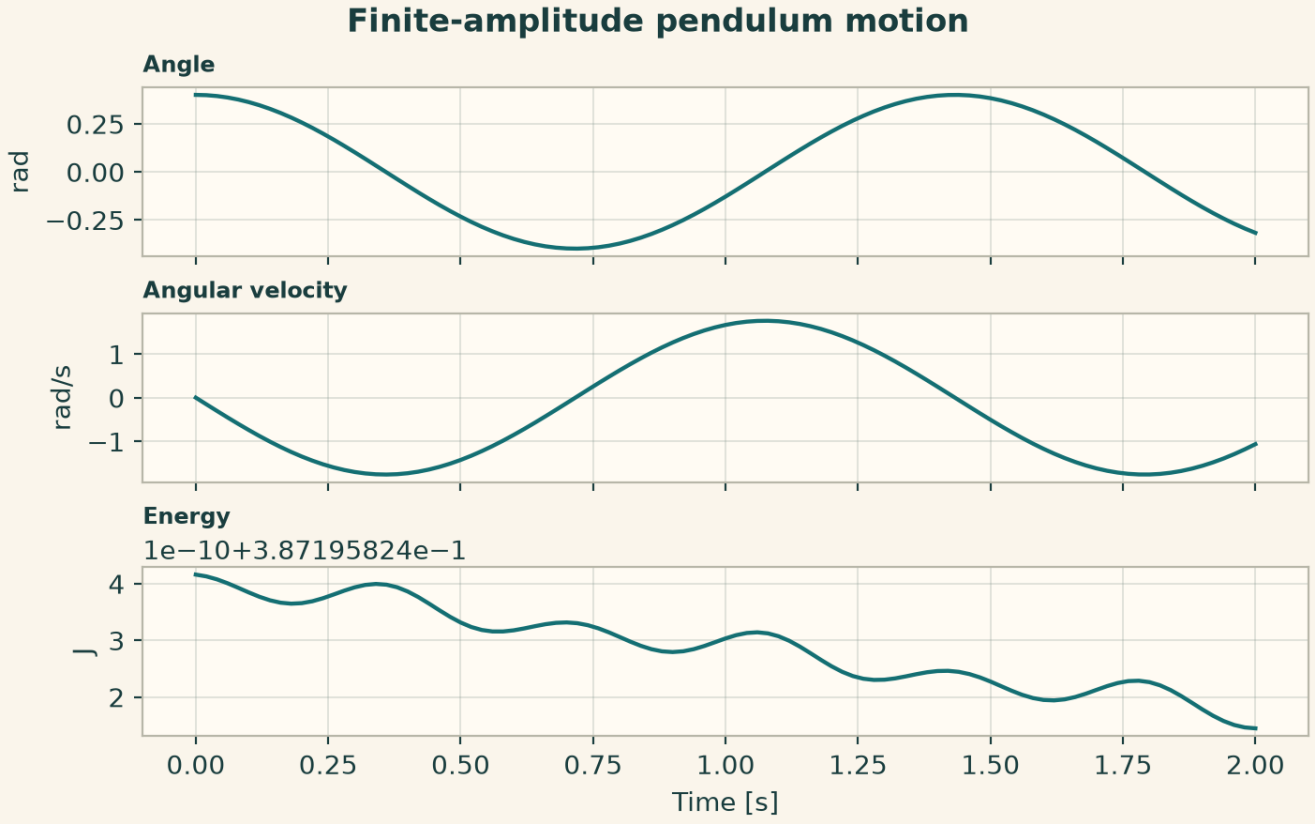
6.14 Multibody dynamics recipes

This section includes the field-based declarations and the independently callable multibody dynamics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.14.1 Finite-amplitude pendulum motion

The angular state is integrated directly. Energy remains constant in the unforced case, even when the motion departs from a small-angle approximation.

```
from onesim.physics.multibody import SimplePendulum

domain = LumpedDomain()
model = SimplePendulum(
    domain=domain.system,
    length=0.5,
)
t, data = history_case(
    domain,
    model,
    dict(angle=0.4, angular_velocity=0.0),
    ("angle", "angular_velocity", "energy"),
    end=2,
    dt=0.005,
)
np.testing.assert_allclose(data["energy"], data["energy"][0], atol=1e-8)
fig = history_plot("Finite-amplitude pendulum motion", model, t, data)
plt.show()
```

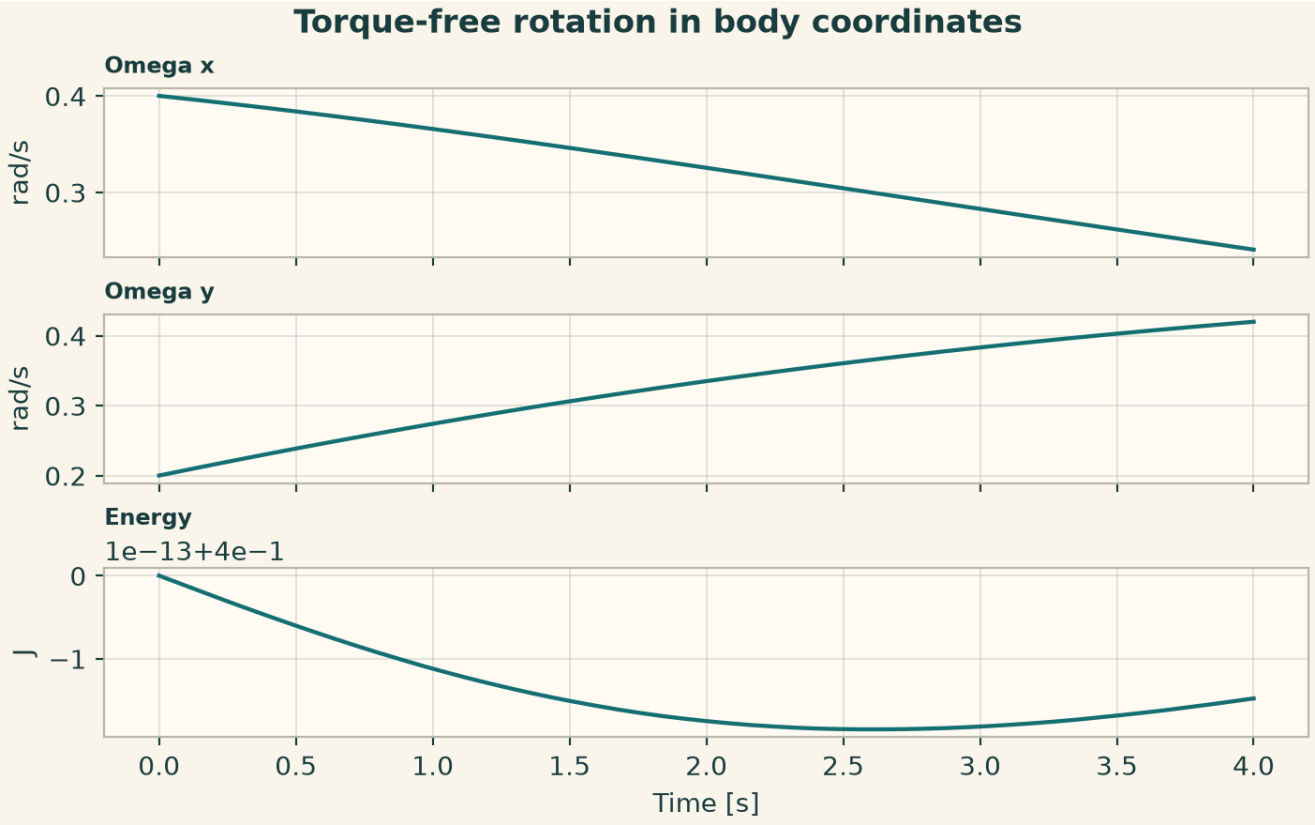


6.14.2 Torque-free rotation in body coordinates

The inertia tuple contains the principal moments. Inspect two body-axis angular velocities and check that the rigid body's kinetic energy is conserved.

```
from onesim.physics.multibody import EulerTop

domain = LumpedDomain()
model = EulerTop(
```

```
    domain=domain.system,
    inertia=(2, 3, 4),
)
t, data = history_case(
    domain,
    model,
    dict(omega_x=0.4, omega_y=0.2, omega_z=0.3),
    ("omega_x", "omega_y", "energy"),
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(data["energy"], data["energy"][0], atol=1e-9)
fig = history_plot("Torque-free rotation in body coordinates", model, t, data)
plt.show()
```

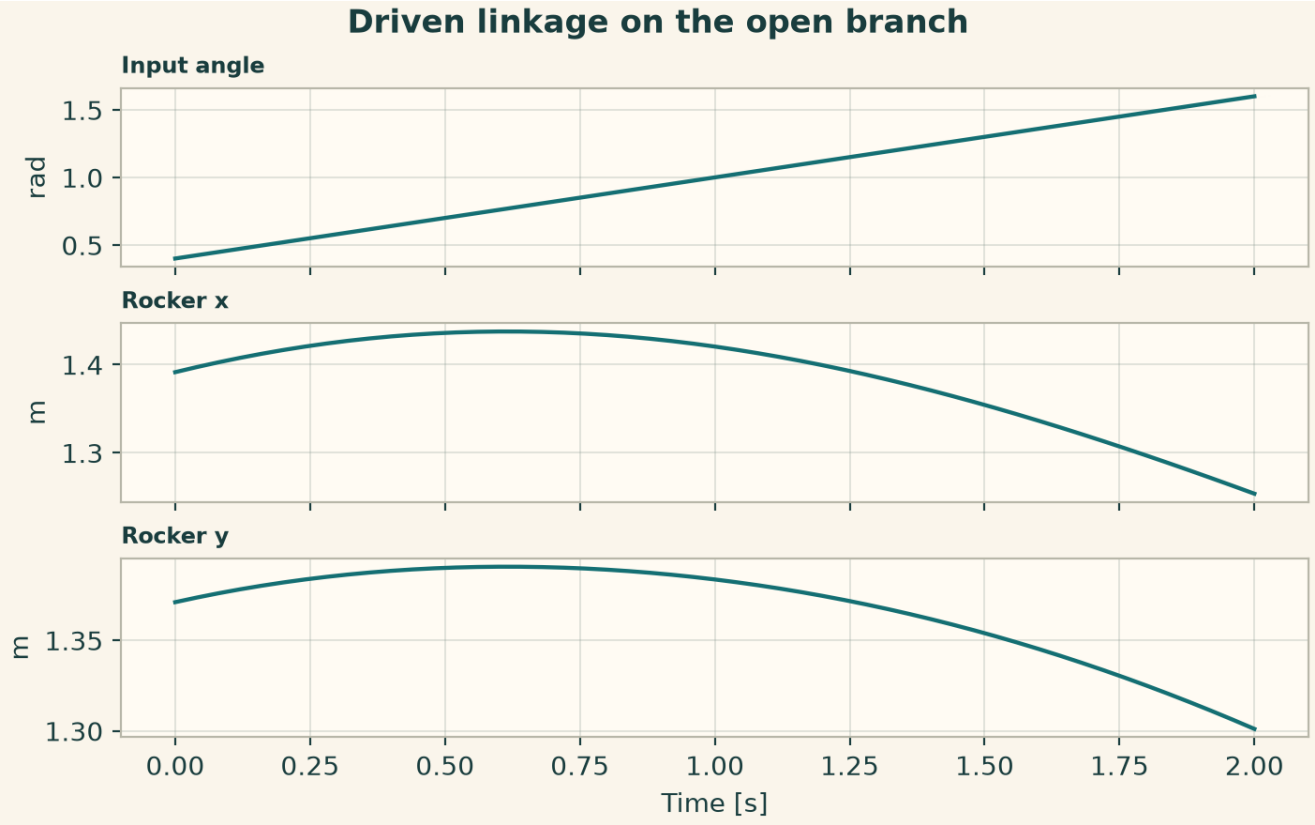


6.14.3 Driven linkage on the open branch

Supply the four link lengths, an assembly branch and a crank speed. Track the crank angle and the output rocker position.

```
from onesim.physics.multibody import DrivenFourBar

domain = LumpedDomain()
model = DrivenFourBar(
    domain=domain.system,
    ground_length=2,
    crank_length=0.5,
    coupler_length=1.5,
```

```
    rocker_length=1.5,
    drive_speed=0.6,
    assembly_branch="open",
)
t, data = history_case(
    domain,
    model,
    dict(input_angle=0.4),
    ("input_angle", "rocker_x", "rocker_y"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["input_angle"], 0.4 + 0.6 * t, atol=1e-9)
fig = history_plot("Driven linkage on the open branch", model, t, data)
plt.show()
```

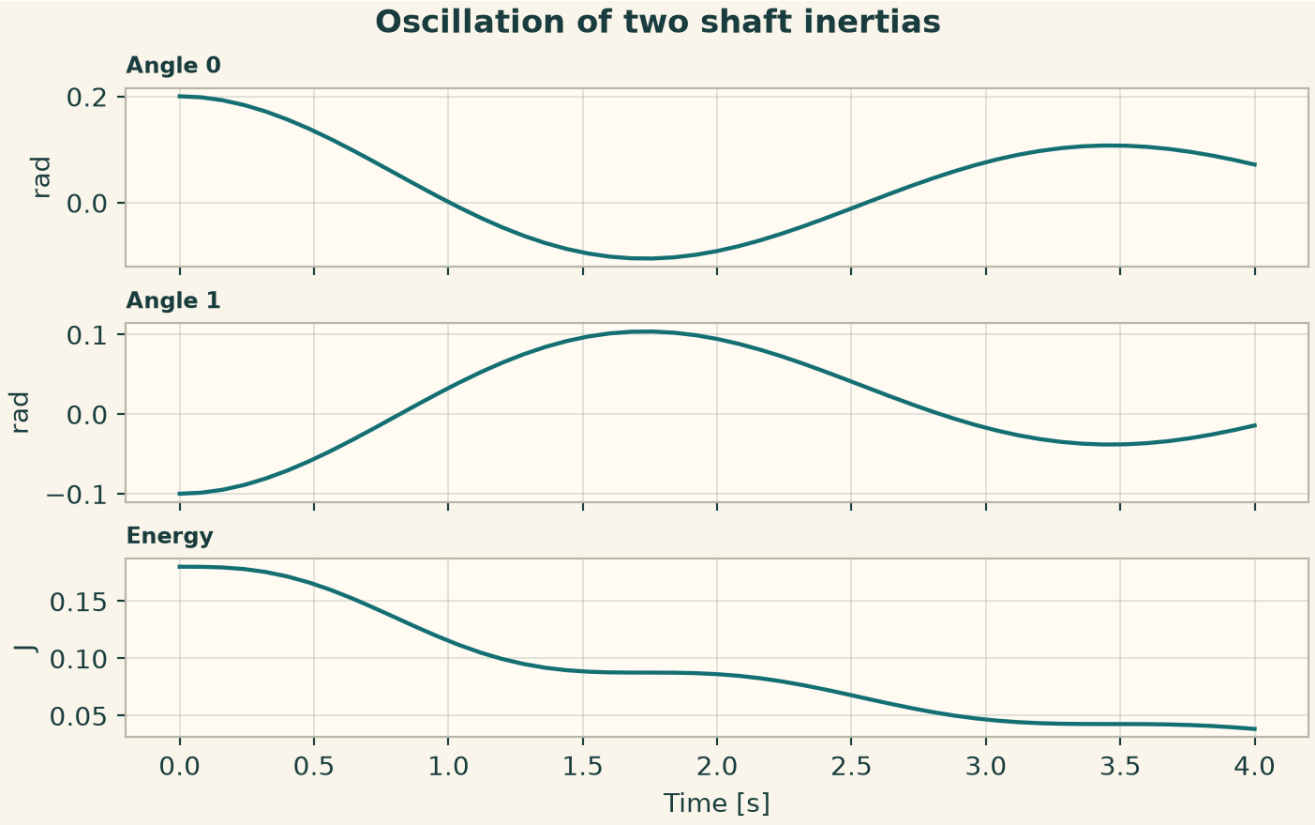


6.14.4 Oscillation of two shaft inertias

Start with a twist across the shaft and apply no external torque. Damping reduces the total mechanical energy while exchanging angular momentum.

```
from onesim.physics.multibody import TorsionalDrivetrain

domain = LumpedDomain()
model = TorsionalDrivetrain(
    domain=domain.system,
    inertias=(2, 3),
    shaft_stiffness=(4,),
```

```
    shaft_damping=(0.5,),
    external_torques=(0, 0),
)
t, data = history_case(
    domain,
    model,
    dict(angle_0=0.2, angle_1=-0.1, omega_0=0.0, omega_1=0.0),
    ("angle_0", "angle_1", "energy"),
    end=4,
    dt=0.02,
)
assert data["energy"][-1] < data["energy"][0]
fig = history_plot("Oscillation of two shaft inertias", model, t, data)
plt.show()
```

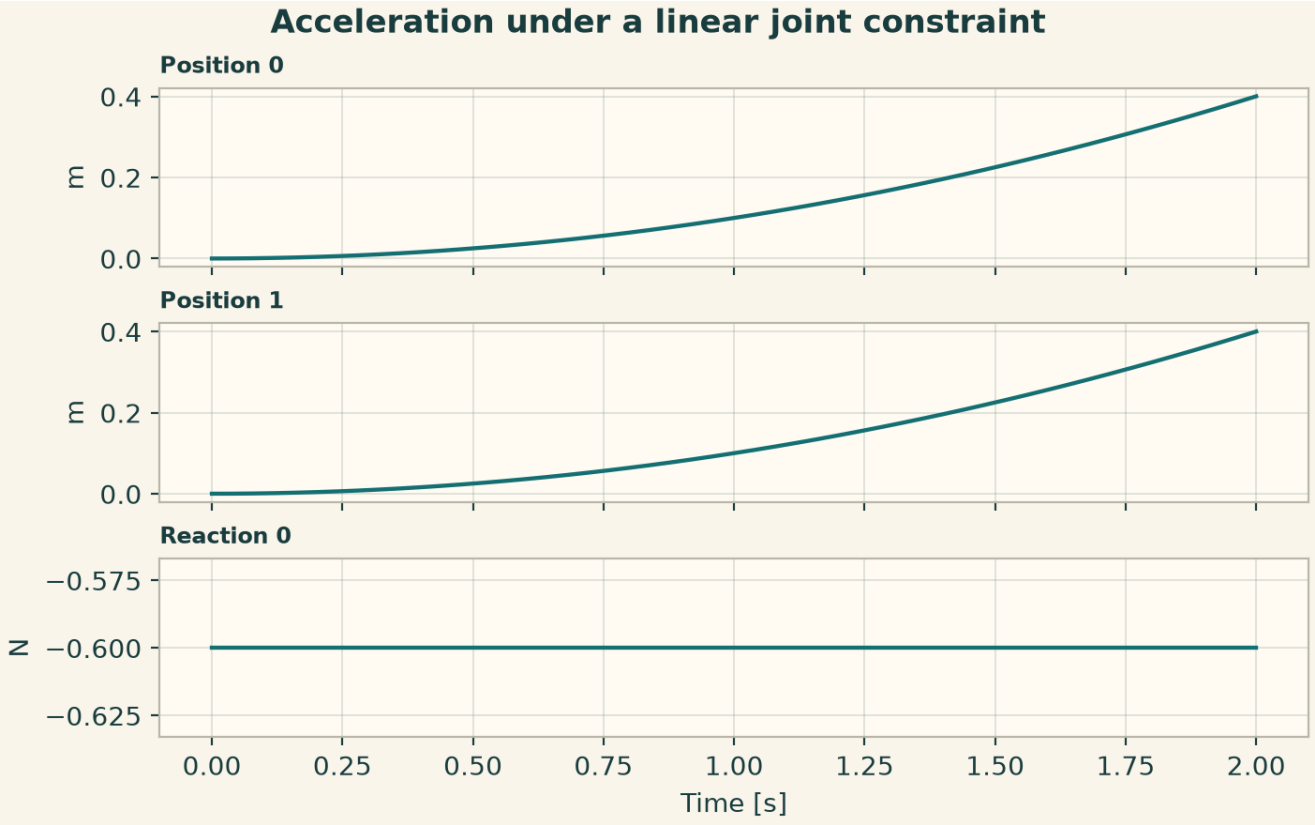


6.14.5 Acceleration under a linear joint constraint

Constrain the two positions to coincide and apply a force to only one mass. Start with a compatible position and velocity and inspect the constraint reaction.

```
from onesim.physics.multibody import LinearJointSystem

domain = LumpedDomain()
model = LinearJointSystem(
    domain=domain.system,
    masses=(2, 3),
    constraint_matrix=((1.0, -1.0),),
    constraint_targets=(0.0,),
```

```
    forces=(1.0, 0.0),
)
t, data = history_case(
    domain,
    model,
    dict(position_0=0.0, position_1=0.0, velocity_0=0.0, velocity_1=0.0),
    ("position_0", "position_1", "reaction_0"),
    end=2,
    dt=0.01,
    linear_solver=SparseDirect(),
)
np.testing.assert_allclose(data["position_0"], data["position_1"], atol=1e-10)
fig = history_plot("Acceleration under a linear joint constraint", model, t, data)
plt.show()
```

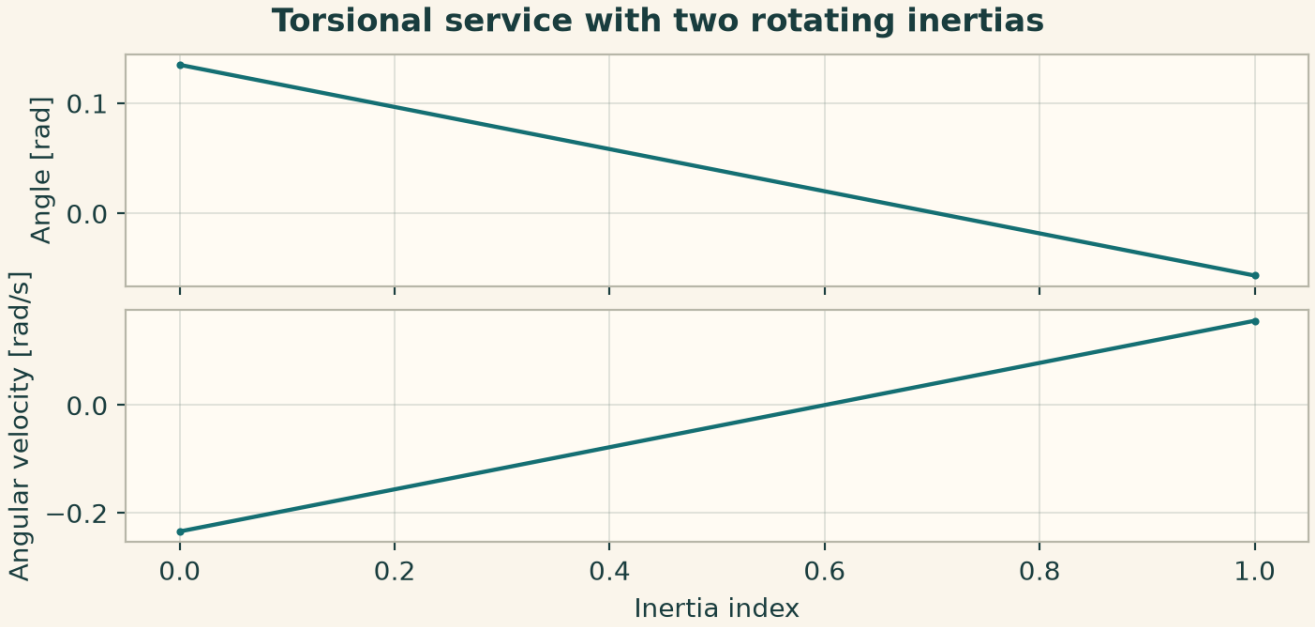


6.14.6 Torsional service with two rotating inertias

Configure a finite final time, an initial shaft twist and damping. The output is indexed by inertia rather than by spatial coordinate.

```
from onesim.physics.multibody import DrivetrainConfiguration, DrivetrainPublicApi

settings = dict(
    inertias=(2.0, 3.0),
    shaft_stiffness=(4.0,),
    shaft_damping=(0.5,),
    initial_angles=(0.2, -0.1),
    final_time=0.5,
```

```
    time_step=0.01,
)
result = run_service(
    DrivetrainPublicApi(), DrivetrainConfiguration(parameters=settings)
)
fig = service_figure(
    "Torsional service with two rotating inertias",
    np.arange(2),
    "Inertia index",
    ("Angle [rad]", result.angles),
    ("Angular velocity [rad/s]", result.angular_velocities),
)
plt.show()
```

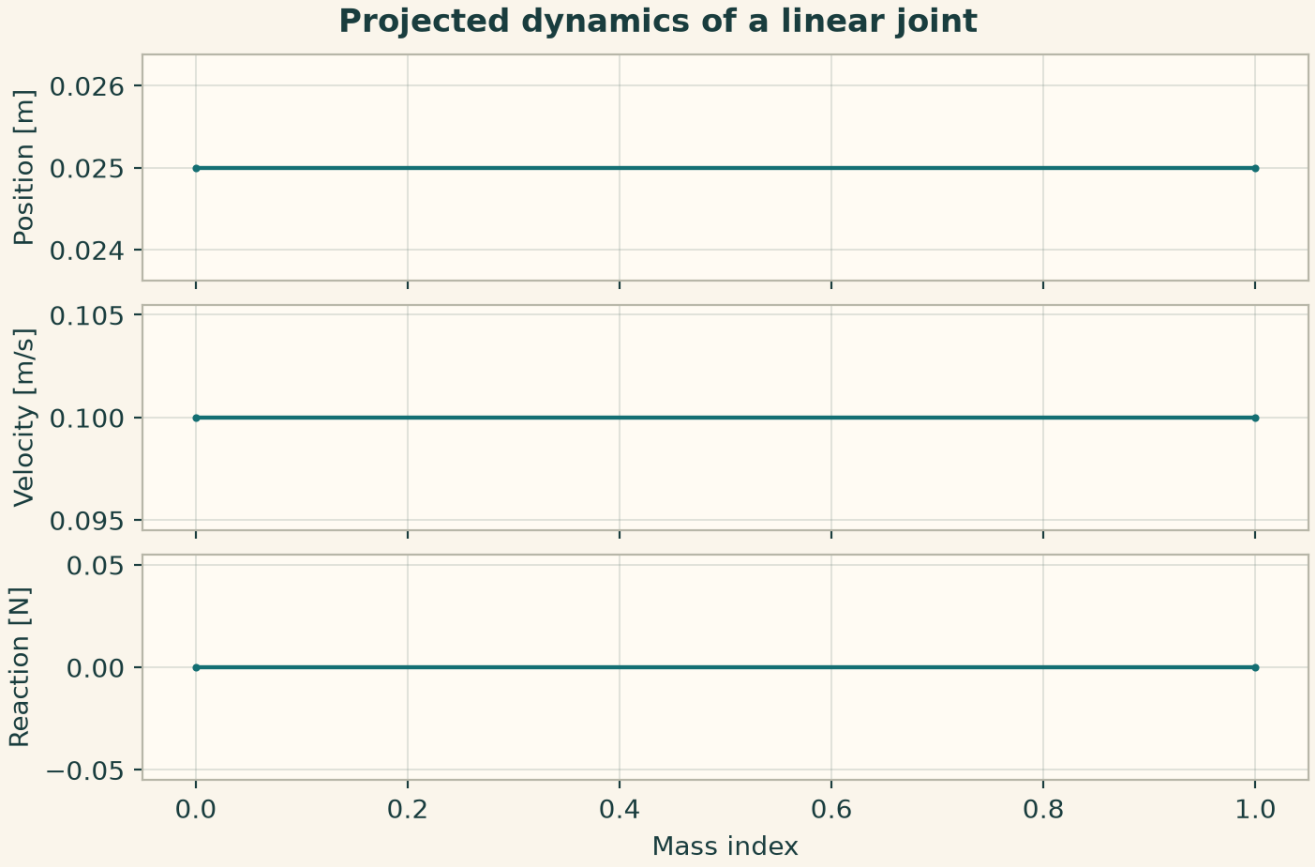


6.14.7 Projected dynamics of a linear joint

Supply a compatible constrained state and choose projection enforcement. The endpoint positions and velocities must satisfy the same linear relation.

```
from onesim.physics.multibody import JointSystemsConfiguration, JointSystemsPublicApi

settings = dict(
    masses=(2.0, 3.0),
    constraint_matrix=((1.0, -1.0),),
    constraint_targets=(0.0,),
    external_force=(1.0, 0.0),
    initial_positions=(0.0, 0.0),
    initial_velocities=(0.0, 0.0),
    final_time=0.5,
    time_step=0.01,
)
result = run_service(
    JointSystemsPublicApi(), JointSystemsConfiguration(parameters=settings)
```

```
)
fig = service_figure(
    "Projected dynamics of a linear joint",
    np.arange(2),
    "Mass index",
    ("Position [m]", result.positions),
    ("Velocity [m/s]", result.velocities),
    ("Reaction [N]", result.reaction_forces),
)
plt.show()
```

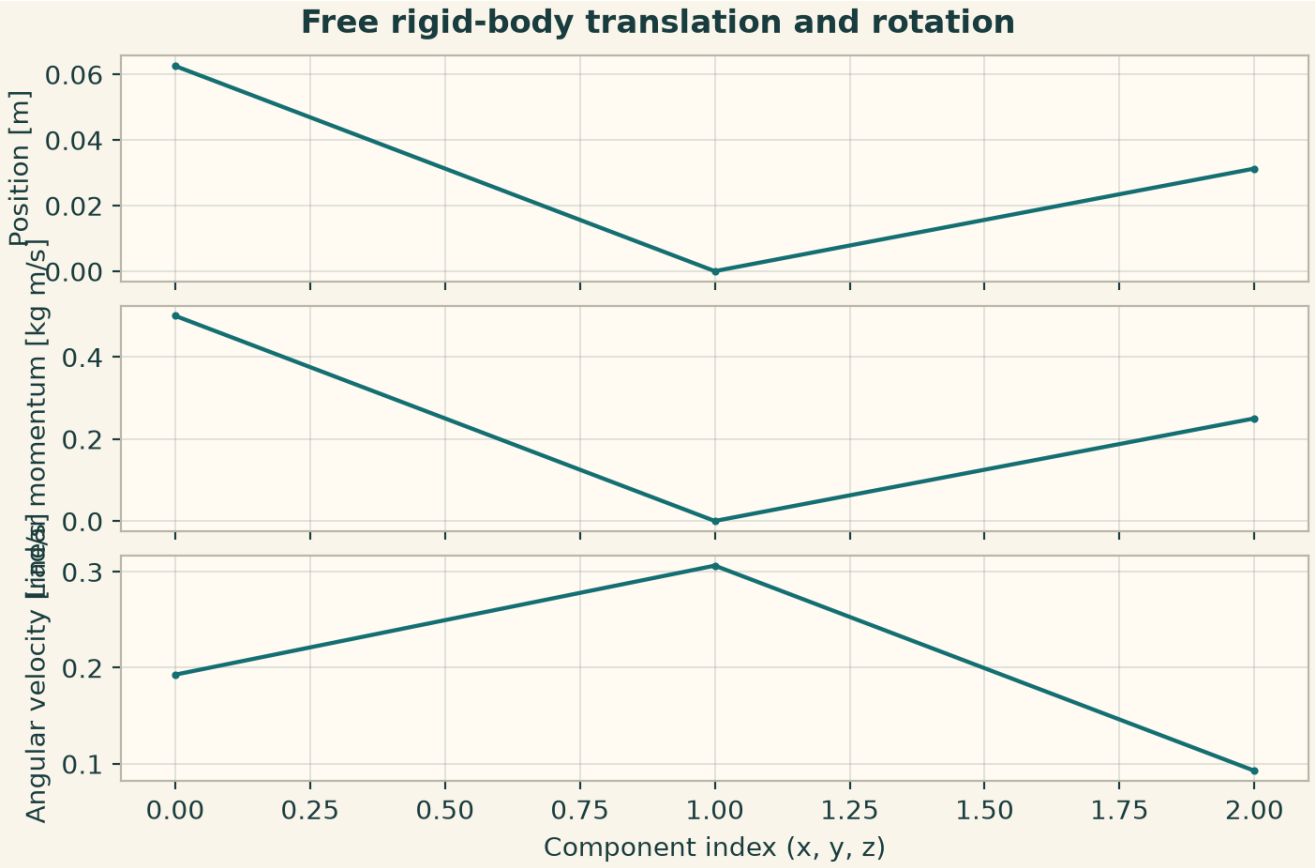


6.14.8 Free rigid-body translation and rotation

An external force and principal inertias advance the six-degree-of-freedom state. The service also returns a quaternion and a rigid transform.

```
from onesim.physics.multibody import (
    RigidBodyDynamicsConfiguration,
    RigidBodyDynamicsPublicApi,
)

settings = dict(
    mass=2,
    inertia=(2.0, 3.0, 4.0),
    external_force=(1.0, 0.0, 0.5),
    initial_angular_velocity=(0.2, 0.3, 0.1),
    final_time=0.5,
```

```
    time_step=0.01,
)
result = run_service(
    RigidBodyDynamicsPublicApi(), RigidBodyDynamicsConfiguration(parameters=settings)
)
fig = service_figure(
    "Free rigid-body translation and rotation",
    np.arange(3),
    "Component index (x, y, z)",
    ("Position [m]", result.position),
    ("Linear momentum [kg m/s]", result.linear_momentum),
    ("Angular velocity [rad/s]", result.angular_velocity),
)
plt.show()
```

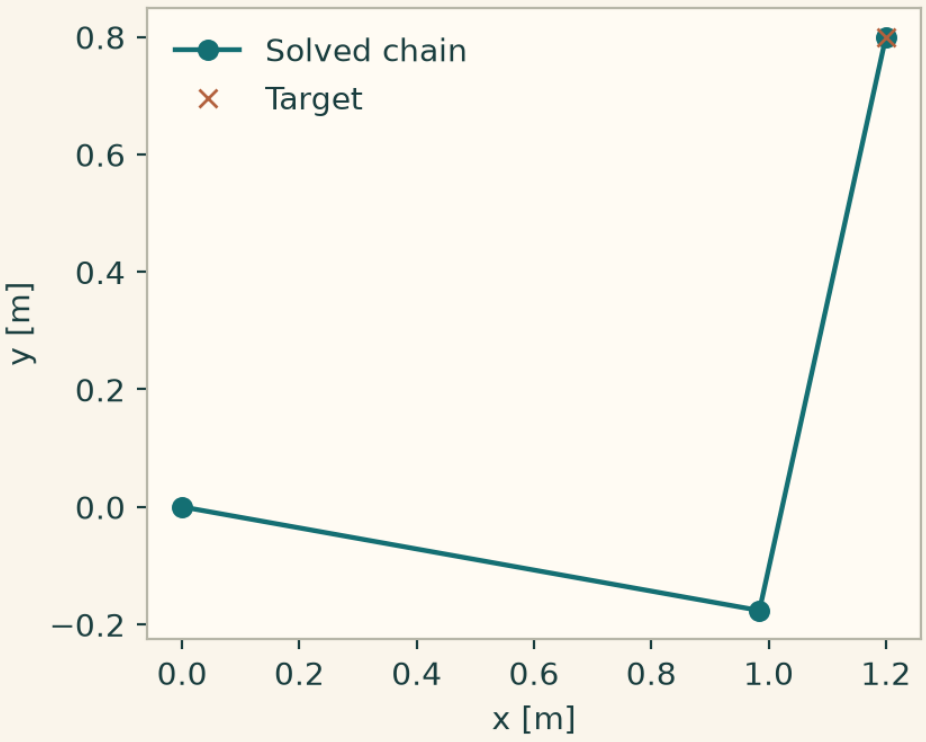


6.14.9 Inverse kinematics of a two-link chain

Provide a reachable Cartesian target and an initial branch seed. Plot the solved joint positions and verify the end effector reaches the target.

```
from onesim.physics.multibody import KinematicsConfiguration, KinematicsPublicApi

target = np.array([1.2, 0.8])
configuration = KinematicsConfiguration(
    parameters=dict(
        link_lengths=(1.0, 1.0),
        initial_joint_angles=(0.1, 0.8),
```

```
        target_position=target,
    )
)
result = run_service(KinematicsPublicApi(), configuration)
np.testing.assert_allclose(result.end_effector, target, atol=1e-8)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6, 3.4), layout="constrained")
ax.plot(*result.joint_positions.T, "o-", label="Solved chain")
ax.plot(*target, "x", color=theme.COPPER, label="Target")
ax.set(xlabel="x [m]", ylabel="y [m]", aspect="equal")
ax.legend(frameon=False)
plt.show()
```

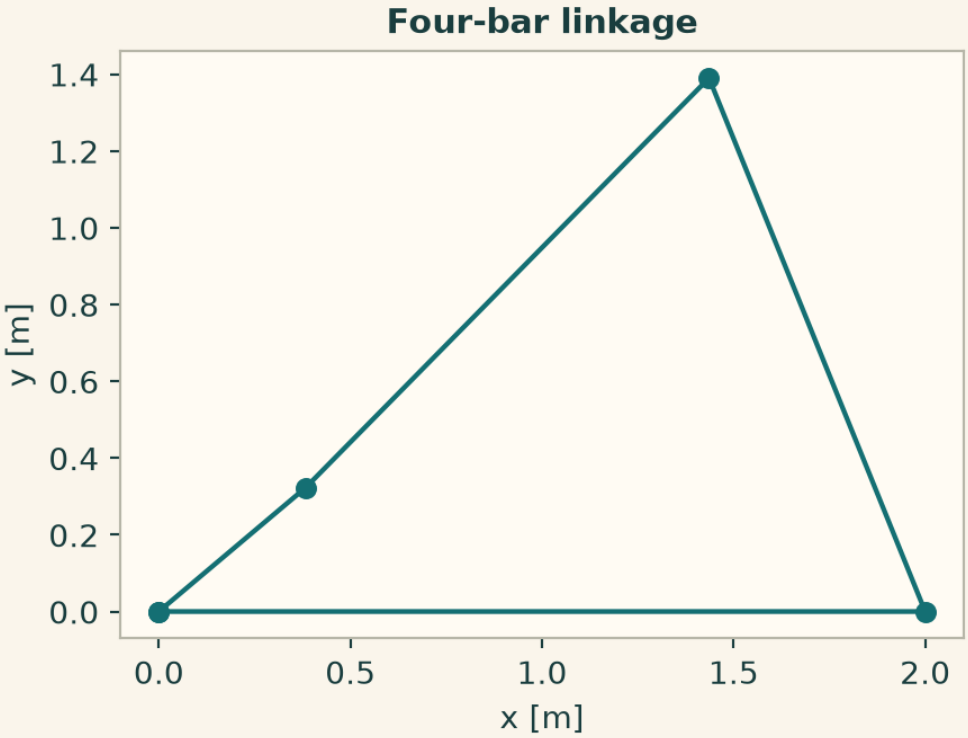


6.14.10 Four-bar service closure and transmission angle

Read the ground pivot, crank and rocker pin after solving the selected assembly branch. The coupler geometry is reconstructed from the returned pins.

```
from onesim.physics.multibody import MechanismsConfiguration, MechanismsPublicApi

configuration = MechanismsConfiguration(
    parameters=dict(
        ground_length=2.0,
        crank_length=0.5,
        coupler_length=1.5,
        rocker_length=1.5,
        initial_input_angle=0.4,
        input_rate=0.6,
        assembly_branch="open",
        final_time=0.5,
```

```
        time_step=0.01,
    )
)
result = run_service(MechanismsPublicApi(), configuration)
assert result.closure_residual < 1e-9
vertices = np.stack(
    ((0, 0), result.crank_pin, result.rocker_pin, result.ground_pivot, (0, 0))
)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6, 3.4), layout="constrained")
ax.plot(*vertices.T, "o-")
ax.set(xlabel="x [m]", ylabel="y [m]", aspect="equal", title="Four-bar linkage")
plt.show()
```



6.15 Multiphysics recipes

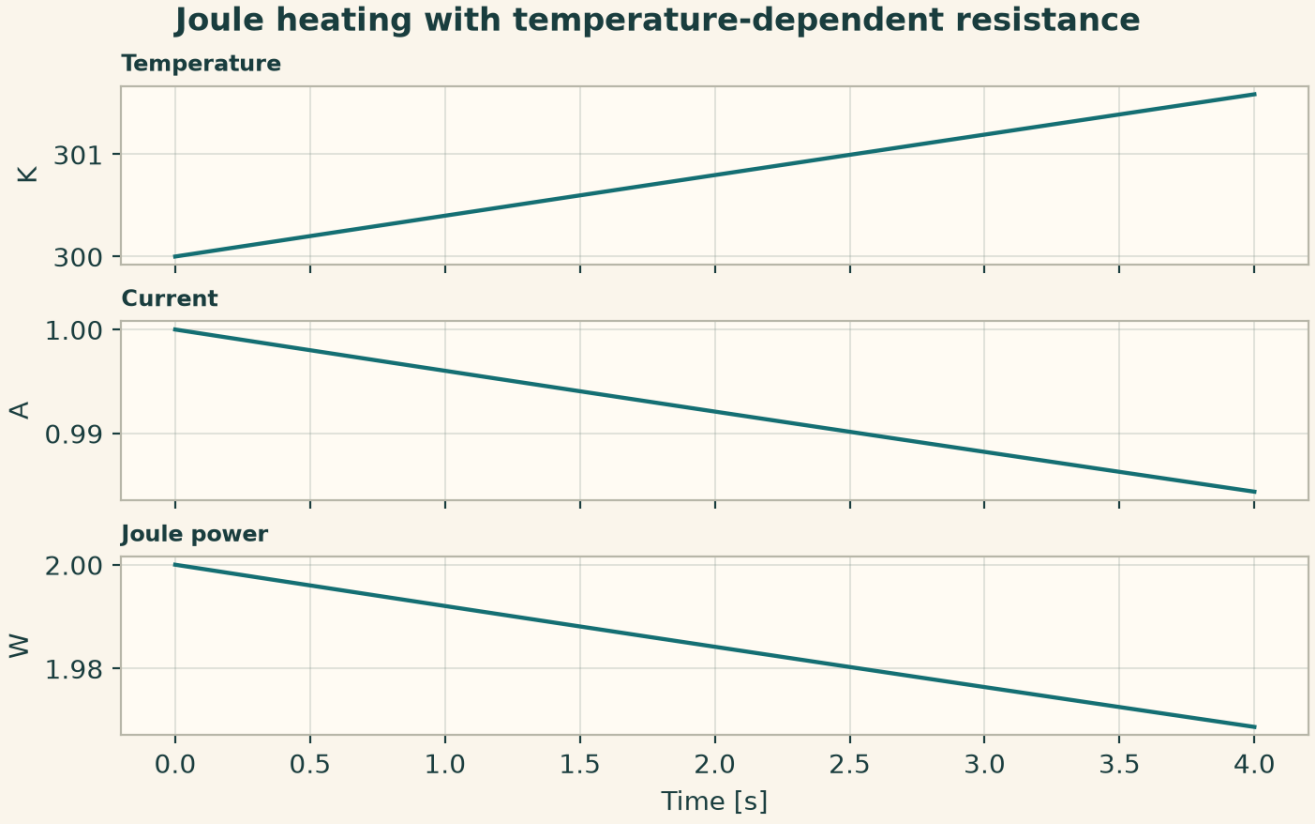
This section includes the field-based declarations and the independently callable multiphysics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.15.1 Joule heating with temperature-dependent resistance

Apply a voltage and use a linear resistance law around a stated reference temperature. The heating and current histories expose the electrical feedback.

```
from onesim.physics.multiphysics import HeatedResistor

domain = LumpedDomain()
model = HeatedResistor(
    domain=domain.system,
```

```
    resistance=2,
    heat_capacity=5,
    applied_voltage=2,
    temperature_coefficient=0.01,
    reference_temperature=300,
)
t, data = history_case(
    domain,
    model,
    dict(temperature=300.0),
    ("temperature", "current", "joule_power"),
    end=4,
    dt=0.02,
)
assert data["temperature"][-1] > 300 and data["current"][-1] < data["current"][0]
fig = history_plot(
    "Joule heating with temperature-dependent resistance", model, t, data
)
plt.show()
```

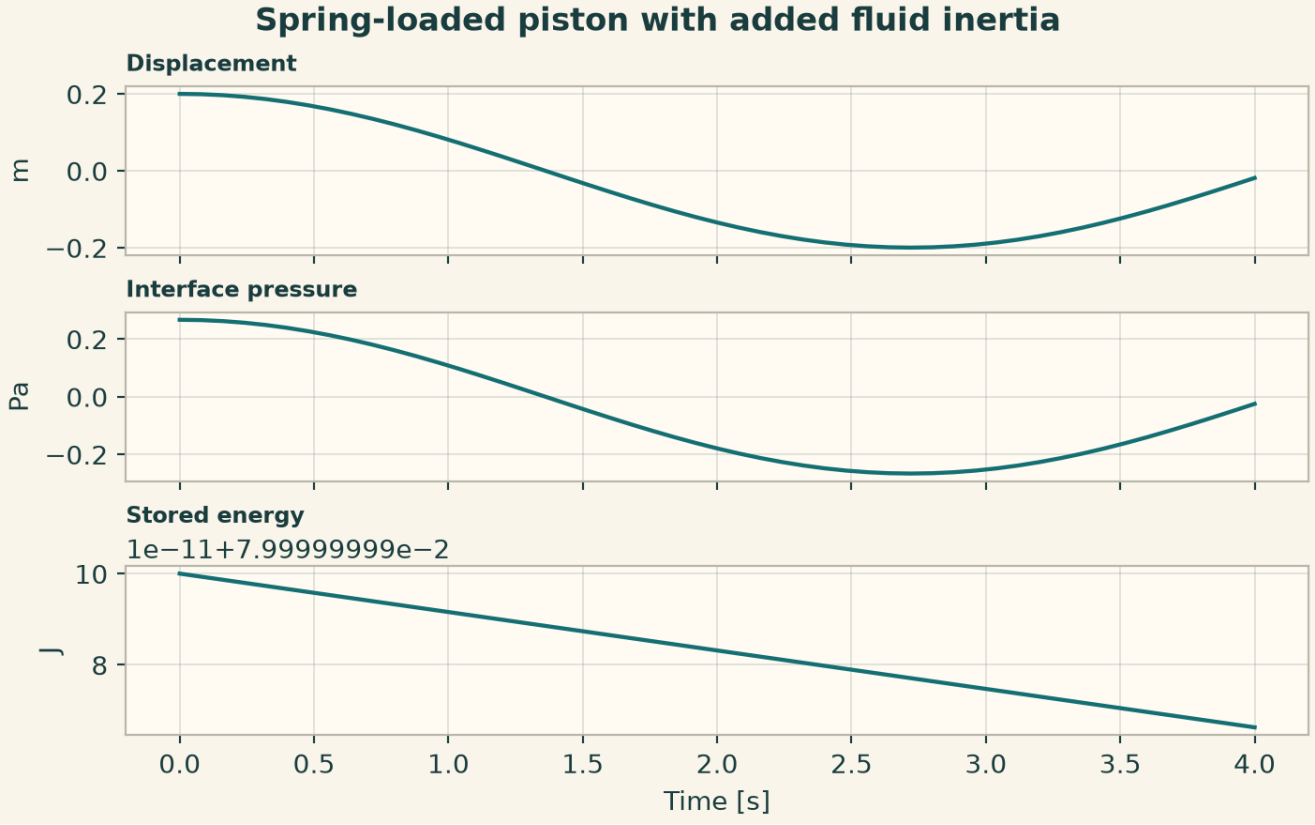


6.15.2 Spring-loaded piston with added fluid inertia

The liquid column contributes inertia to the piston balance. An initial displacement produces periodic motion and a changing interface pressure.

```
from onesim.physics.multiphysics import FluidColumnPiston

domain = LumpedDomain()
```

```
model = FluidColumnPiston(
    domain=domain.system,
    piston_mass=2,
    fluid_density=1,
    column_length=1,
    area=1,
    spring_stiffness=4,
)
t, data = history_case(
    domain,
    model,
    dict(displacement=0.2, velocity=0.0),
    ("displacement", "interface_pressure", "stored_energy"),
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(data["stored_energy"], data["stored_energy"][0], atol=1e-8)
fig = history_plot("Spring-loaded piston with added fluid inertia", model, t, data)
plt.show()
```

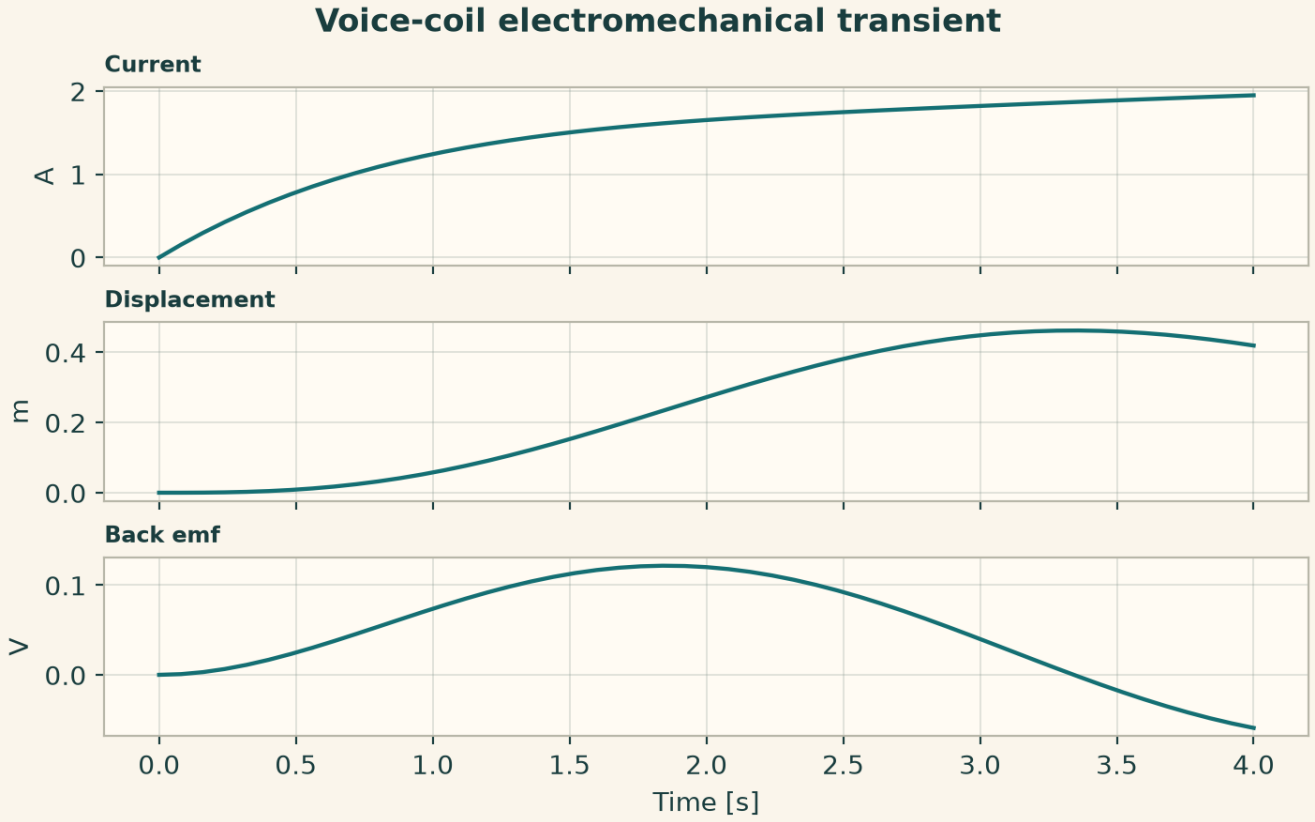


6.15.3 Voice-coil electromechanical transient

A voltage drives current and therefore force on a damped spring mass. Motion generates back EMF through the same force factor.

```
from onesim.physics.multiphysics import VoiceCoil

domain = LumpedDomain()
```

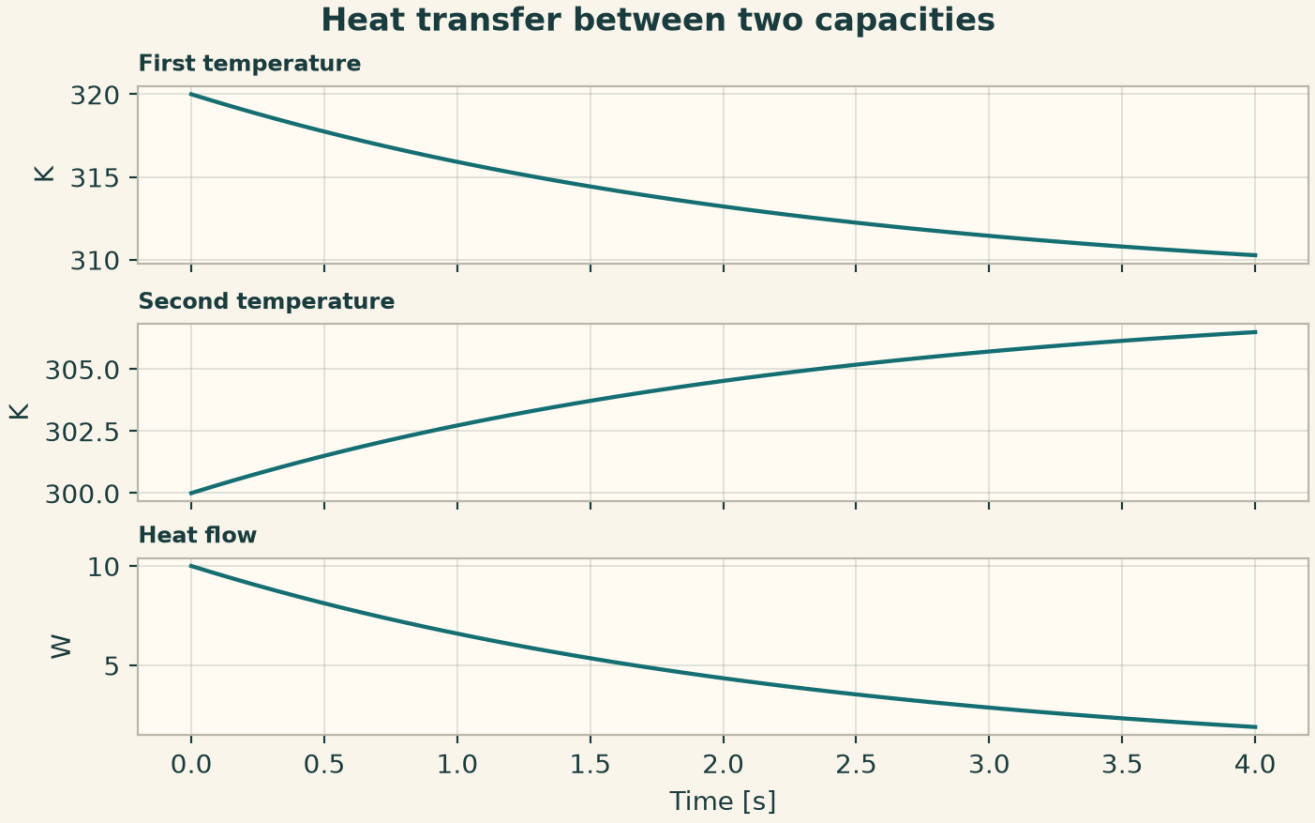
```
model = VoiceCoil(
    domain=domain.system,
    resistance=1,
    inductance=1,
    force_factor=0.5,
    mass=2,
    spring_stiffness=3,
    damping=0.4,
    applied_voltage=2,
)
t, data = history_case(
    domain,
    model,
    dict(current=0.0, displacement=0.0, velocity=0.0),
    ("current", "displacement", "back_emf"),
    end=4,
    dt=0.02,
)
assert data["displacement"].max() > 0
fig = history_plot("Voice-coil electromechanical transient", model, t, data)
plt.show()
```



6.15.4 Heat transfer between two capacities

The conductance transfers energy from the hotter capacity to the cooler one. Check conservation by adding their stored thermal energies.

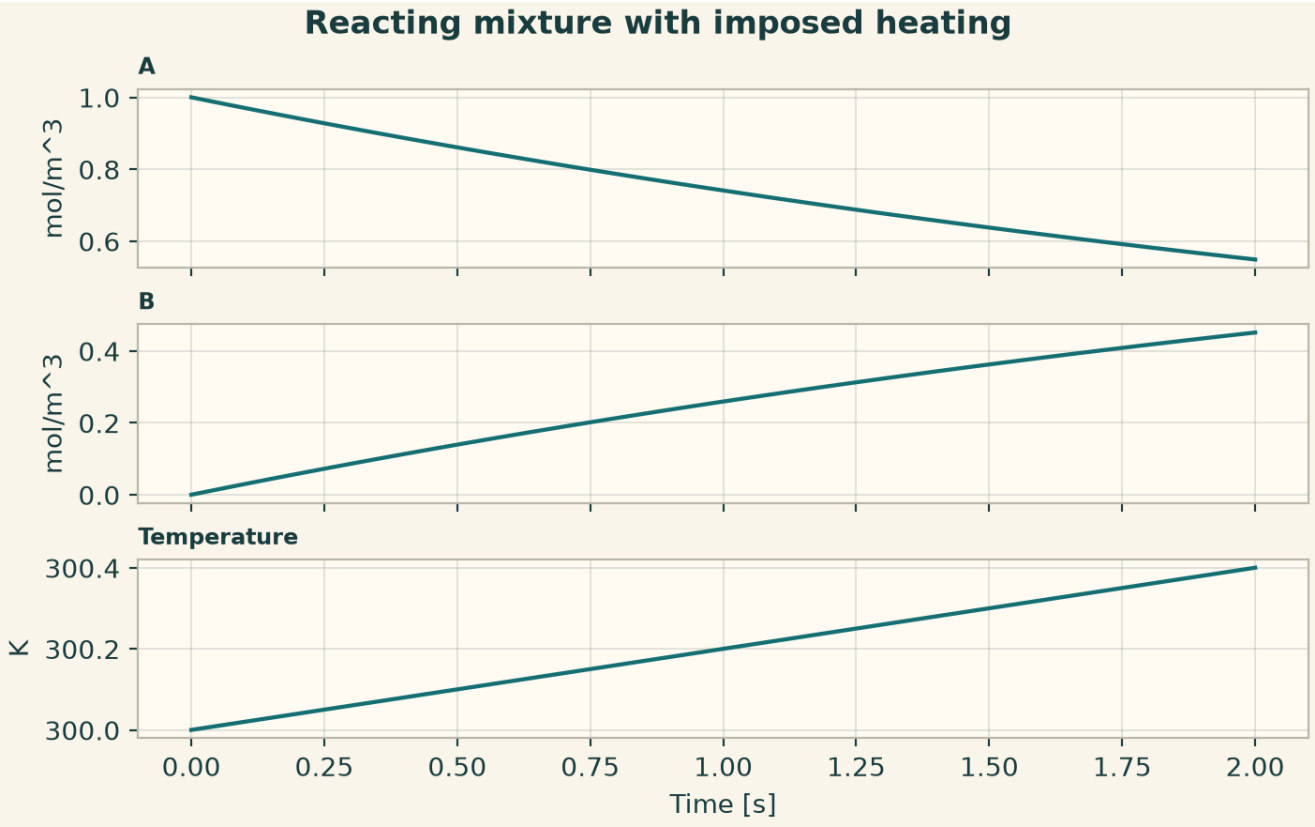
```
from onesim.physics.multiphysics import ThermalExchange

domain = LumpedDomain()
model = ThermalExchange(
    domain=domain.system,
    first_capacity=2,
    second_capacity=3,
    conductance=0.5,
)
t, data = history_case(
    domain,
    model,
    dict(first_temperature=320.0, second_temperature=300.0),
    ("first_temperature", "second_temperature", "heat_flow"),
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(
    2 * data["first_temperature"] + 3 * data["second_temperature"], 1540.0, atol=1e-9
)
fig = history_plot("Heat transfer between two capacities", model, t, data)
plt.show()
```



6.15.5 Heating from a homogeneous reaction

Combine a homogeneous reaction network and a temperature balance. This run specifies an independent volumetric heat source and a reaction with zero specified energy release, so the concentration sum and temperature rise have separate exact checks.

```
from onesim.physics.chemistry import ReactionIR, ReactionNetworkIR, SpeciesIR
from onesim.physics.multiphysics import AdiabaticReaction

domain = LumpedDomain()
species = tuple(
    SpeciesIR(n, 0, 1.66054e-27, composition=(("H", 1),)) for n in ("A", "B")
)
reaction = ReactionIR(
    "conversion", (("A", 1),), (("B", 1),), {"kind": "constant", "value": 0.3}
)
network = ReactionNetworkIR("reactive heating", "1", species, (reaction,))
model = AdiabaticReaction(
    domain=domain.system,
    network=network,
    volumetric_heat_capacity=2.0,
    thermal_source=0.4,
)
a, b = model.concentration("A").name, model.concentration("B").name
t, data = history_case(
    domain,
    model,
    {a: 1.0, b: 0.0, "temperature": 300.0},
    (a, b, "temperature"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data[a] + data[b], 1, atol=1e-12)
np.testing.assert_allclose(data["temperature"], 300 + 0.2 * t, atol=1e-9)
fig = history_plot("Reacting mixture with imposed heating", model, t, data)
plt.show()
```

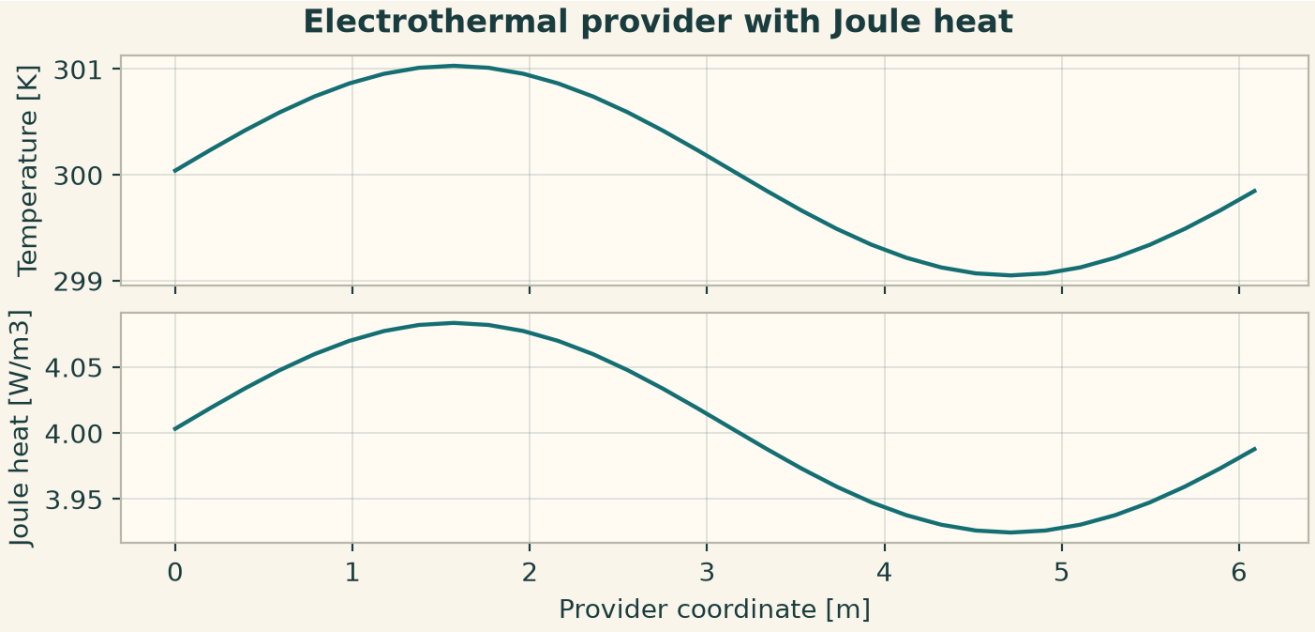


6.15.6 Electrothermal provider with Joule heat

The temperature-dependent conductivity changes the current density and Joule power in a prescribed electric field. Inspect the thermal and electrical outputs on the same coordinates.

```
from onesim.physics.multiphysics import (
    ElectroThermalConfiguration,
    ElectroThermalPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = ElectroThermalConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_temperature=300 + wave,
        applied_electric_field=2,
        conductivity_temperature_coefficient=0.02,
        reference_temperature=300,
    )
)
result = run_service(ElectroThermalPublicApi(), configuration)
fig = service_figure(
    "Electrothermal provider with Joule heat",
    result.coordinates,
    "Provider coordinate [m]",
```

```
    ("Temperature [K]", result.temperature),
    ("Joule heat [W/m3]", result.joule_heat),
)
plt.show()
```

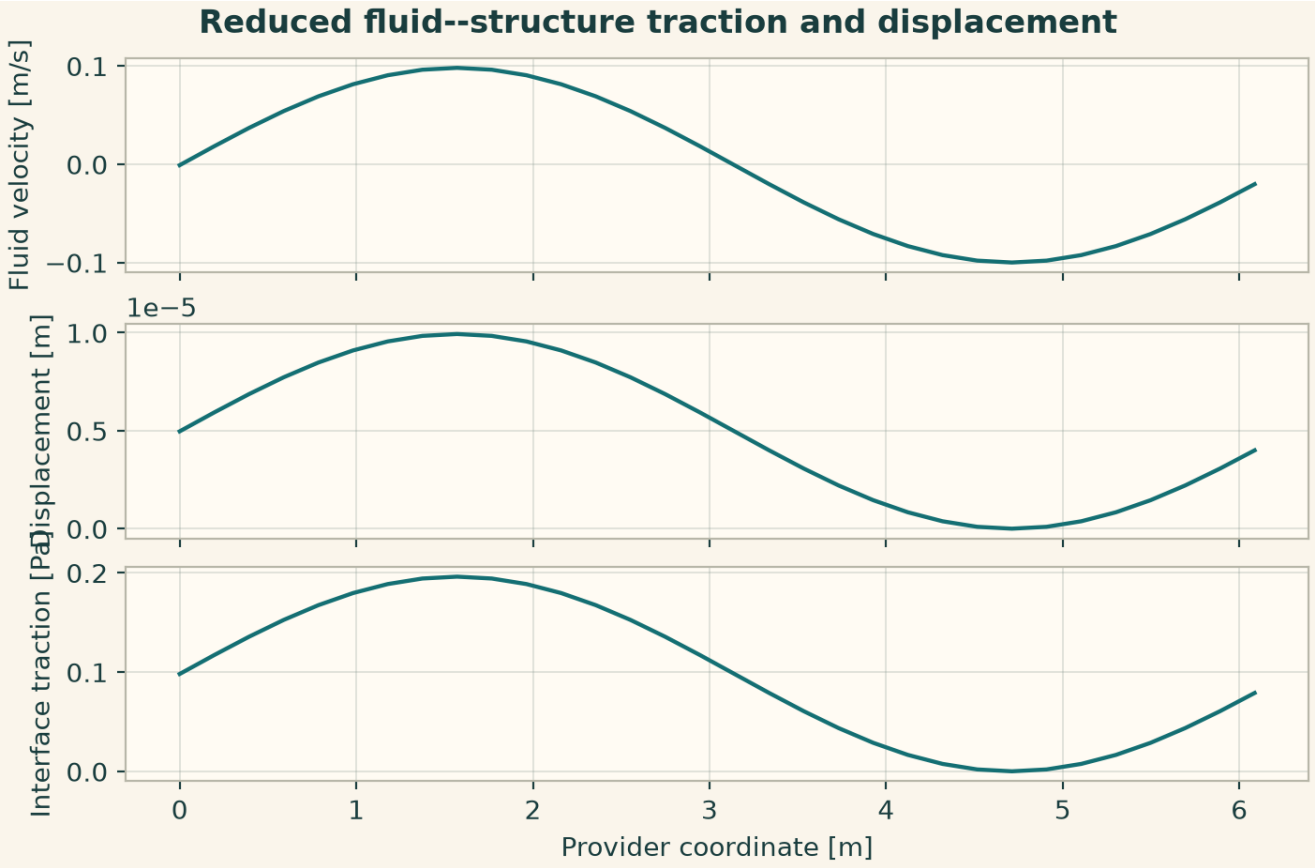


6.15.7 Reduced fluid–structure traction and displacement

Apply interface pressure and vary the initial fluid profile. The service returns traction and both sides of the interface response.

```
from onesim.physics.multiphysics import (
    FluidStructureConfiguration,
    FluidStructurePublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = FluidStructureConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_fluid_velocity=0.1 * wave,
        interface_pressure=0.1,
    )
)
result = run_service(FluidStructurePublicApi(), configuration)
fig = service_figure(
    "Reduced fluid--structure traction and displacement",
    result.coordinates,
    "Provider coordinate [m]",
    ("Fluid velocity [m/s]", result.fluid_velocity),
```

```
    ("Displacement [m]", result.structural_displacement),
    ("Interface traction [Pa]", result.interface_traction),
)
plt.show()
```

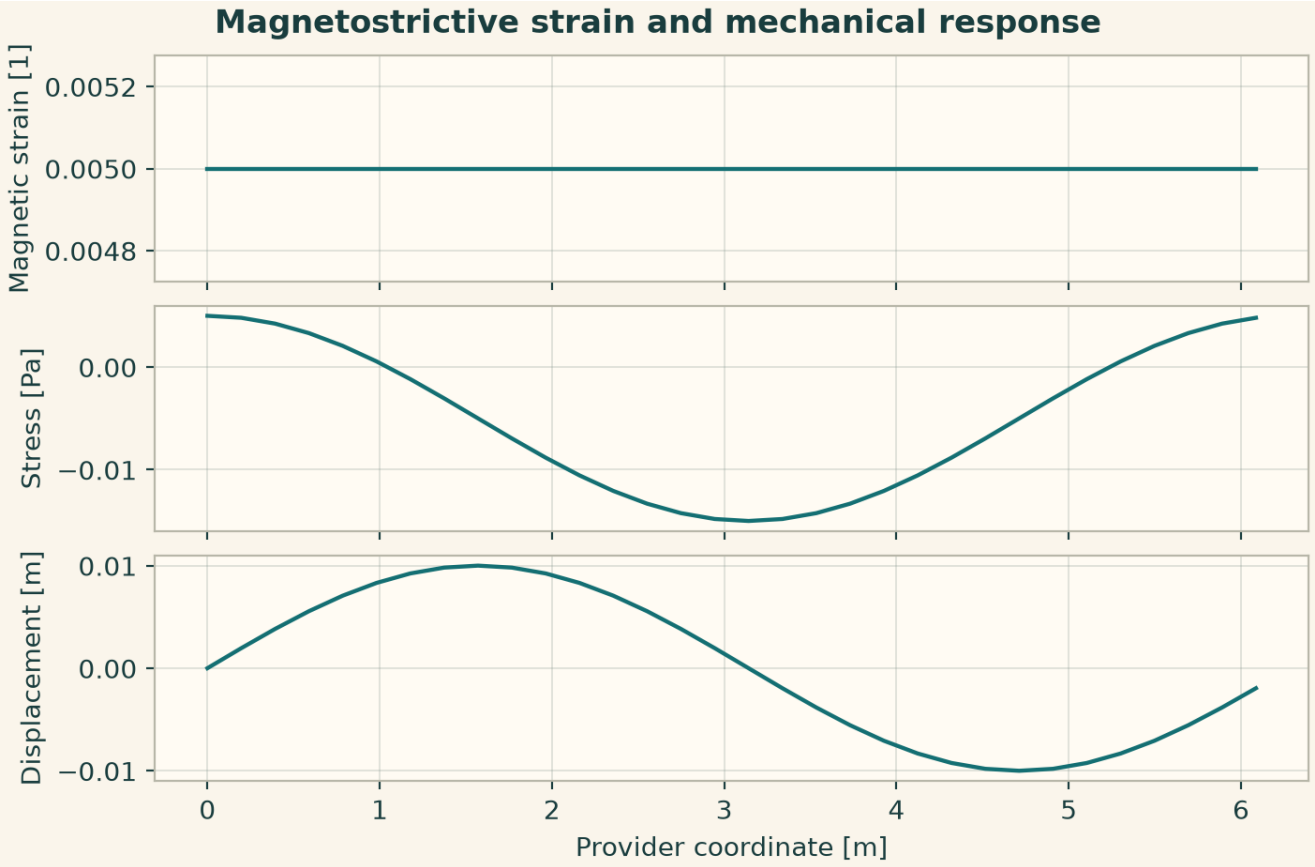


6.15.8 Magnetostrictive strain and mechanical response

Apply a magnetic field and set a magnetostriction coefficient. Read magnetic strain, resulting stress and displacement from the provider result.

```
from onesim.physics.multiphysics import (
    MagnetoMechanicsConfiguration,
    MagnetoMechanicsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = MagnetoMechanicsConfiguration(
    parameters=dict(
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_displacement=0.01 * wave,
        applied_magnetic_field=0.5,
        magnetostriction_coefficient=0.02,
    )
```

```
)
result = run_service(MagnetoMechanicsPublicApi(), configuration)
fig = service_figure(
    "Magnetostrictive strain and mechanical response",
    result.coordinates,
    "Provider coordinate [m]",
    ("Magnetic strain [1]", result.magnetic_strain),
    ("Stress [Pa]", result.stress),
    ("Displacement [m]", result.displacement),
)
plt.show()
```

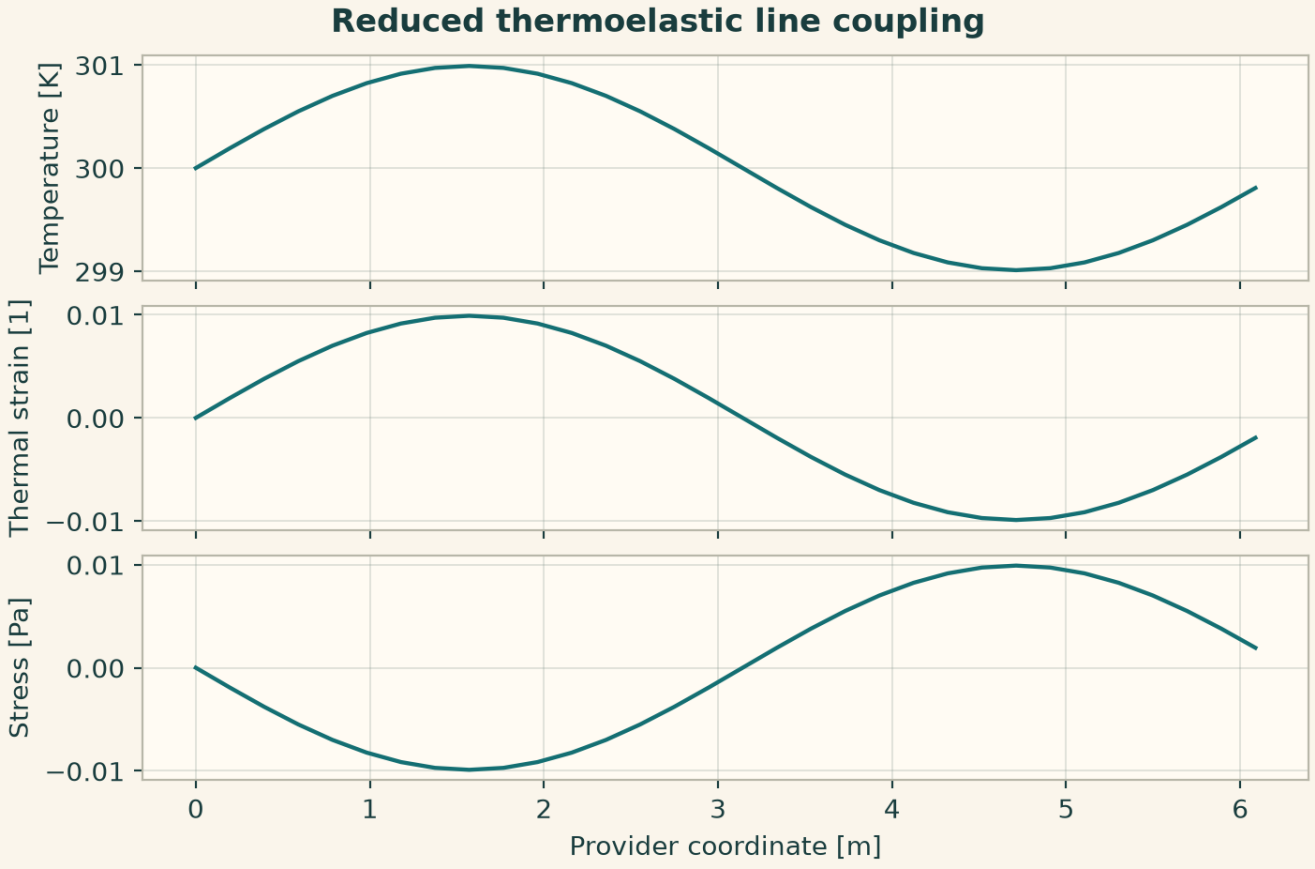


6.15.9 Reduced thermoelastic line coupling

A temperature perturbation produces thermal strain and mechanical stress through the material coefficient. Read the fields on the shared line carrier.

```
from onesim.physics.multiphysics import (
    ThermoMechanicsConfiguration,
    ThermoMechanicsPublicApi,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = ThermoMechanicsConfiguration(
    parameters=dict(
```

```
        cell_count=n,
        final_time=0.01,
        time_step=0.0001,
        initial_temperature=300 + wave,
        reference_temperature=300,
        thermal_expansion=0.01,
    )
)
result = run_service(ThermoMechanicsPublicApi(), configuration)
fig = service_figure(
    "Reduced thermoelastic line coupling",
    result.coordinates,
    "Provider coordinate [m]",
    ("Temperature [K]", result.temperature),
    ("Thermal strain [1]", result.thermal_strain),
    ("Stress [Pa]", result.stress),
)
plt.show()
```

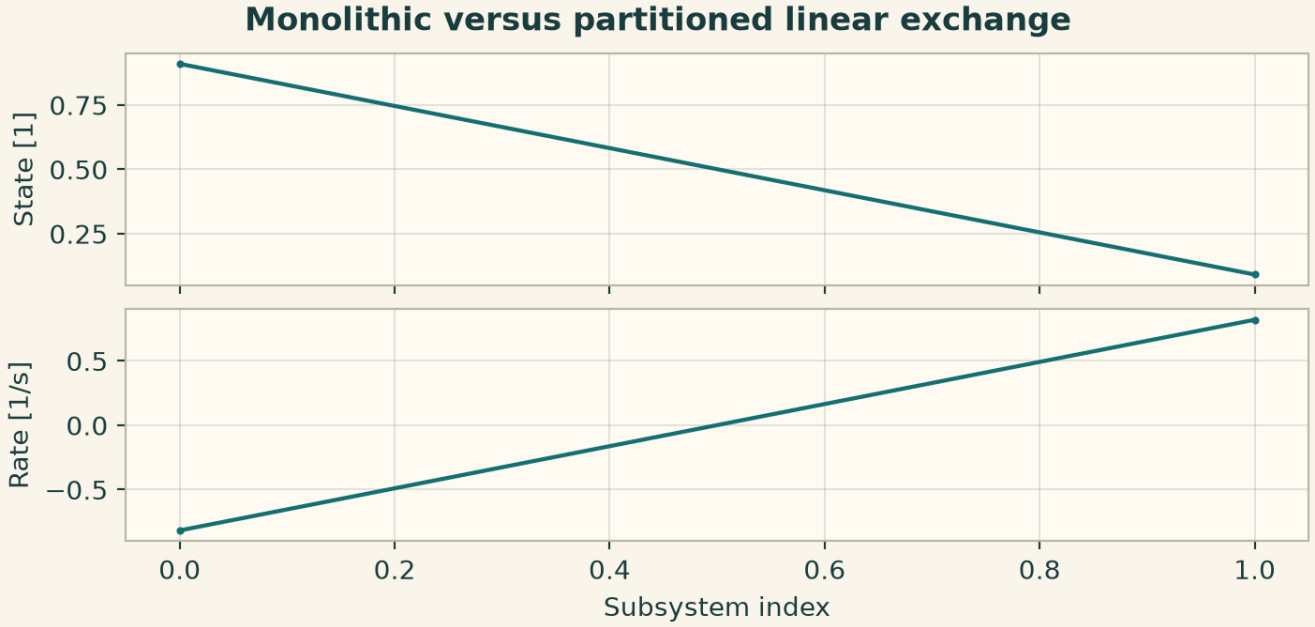


6.15.10 Monolithic versus partitioned linear exchange

Describe the block partition separately from its system matrix. The selected monolithic strategy integrates the two-state exchange and returns a derivative for each block.

```
from onesim.physics.multiphysics import (
    GeneralCoupledSubsystemsConfiguration,
    GeneralCoupledSubsystemsPublicApi,
```

```
)

settings = dict(
    subsystem_ids=("a", "b"),
    subsystem_sizes=(1, 1),
    system_matrix=((-1.0, 1.0), (1.0, -1.0)),
    initial_state=(1.0, 0.0),
    final_time=0.1,
    time_step=0.01,
)
result = run_service(
    GeneralCoupledSubsystemsPublicApi(),
    GeneralCoupledSubsystemsConfiguration(parameters=settings),
)
fig = service_figure(
    "Monolithic versus partitioned linear exchange",
    np.arange(2),
    "Subsystem index",
    ("State [1]", result.state.values),
    ("Rate [1/s]", result.derivative),
)
plt.show()
```

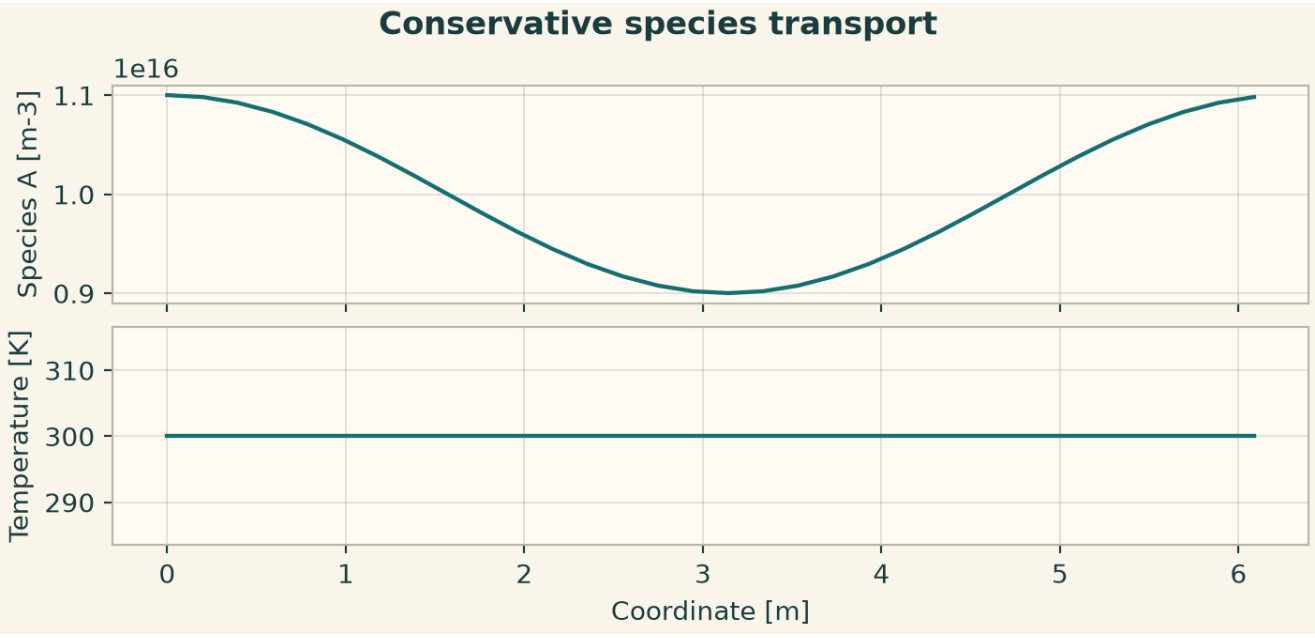


6.15.11 Reactive coupling facade with an explicit species registry

The facade requires a species set, reaction network, initial density arrays and initial temperature. Begin with zero reactions to isolate conservative diffusion. Add balanced reaction channels only after the transport check.

```
from onesim.physics.multiphysics import (
    ReactiveCouplingConfiguration,
    ReactiveCouplingPublicApi,
)
```

```
from onesim.physics.plasma.collisions.models.reactions import ReactionNetwork
from onesim.physics.plasma.fluid.models.declarations.species.species import (
    SpeciesDefinition,
    SpeciesSet,
)

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
species = SpeciesSet(
    (SpeciesDefinition("A", "Tracer", charge_number=0, mass=1.66e-27),)
)
initial = 1e16 * (1 + 0.1 * np.cos(2 * np.pi * x0))
configuration = ReactiveCouplingConfiguration(
    parameters=dict(
        species=species,
        reaction_network=ReactionNetwork(),
        initial_densities={"A": initial},
        initial_temperature=np.full(n, 300.0),
        default_species_diffusivity=0.01,
        thermal_diffusivity=0.02,
        final_time=0.01,
        time_step=0.0001,
    )
)
result = run_service(ReactiveCouplingPublicApi(), configuration)
fig = service_figure(
    "Conservative species transport",
    result.coordinates,
    "Coordinate [m]",
    ("Species A [m-3]", result.densities["A"]),
    ("Temperature [K]", result.temperature),
)
plt.show()
```



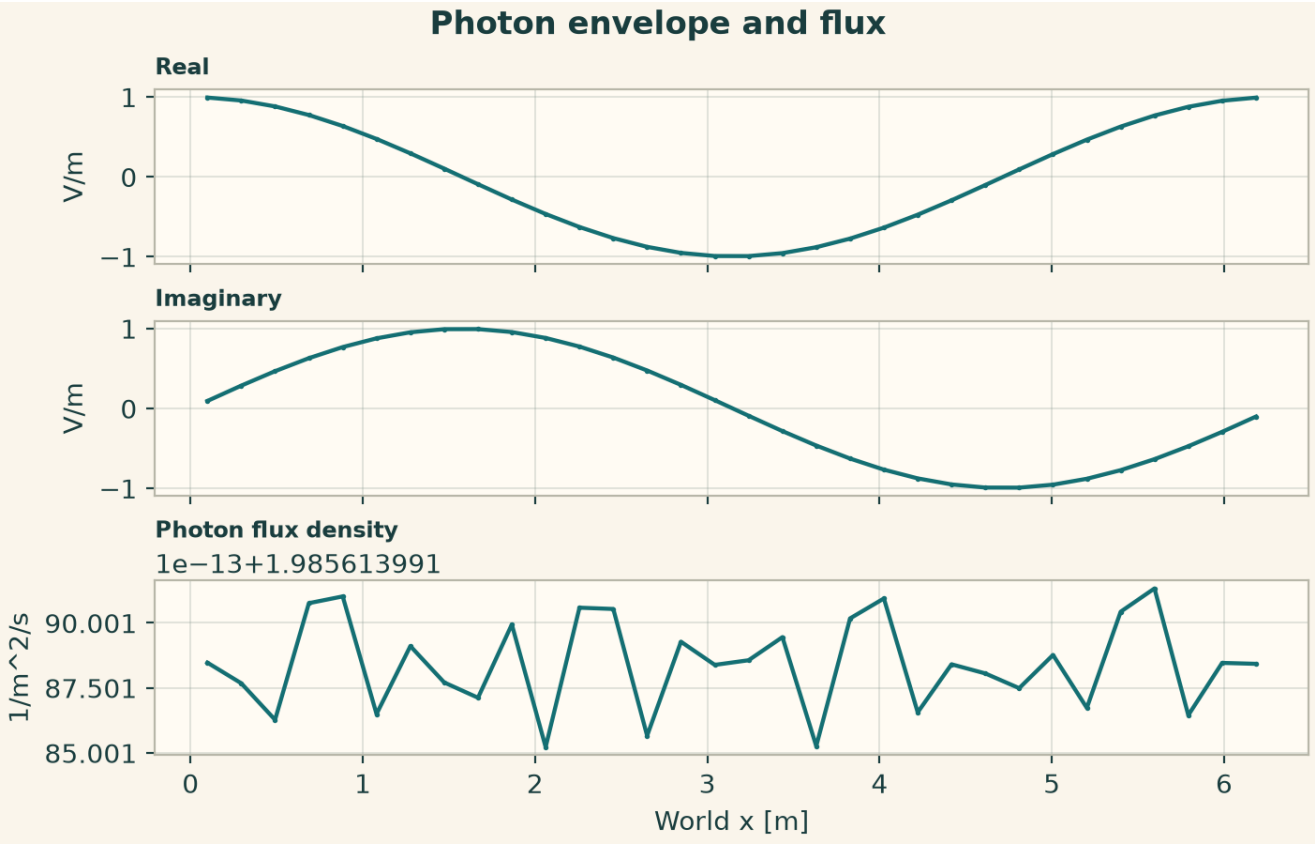
6.16 Optics and photonics recipes

This section includes the field-based declarations and the independently callable optics and photonics service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.16.1 Optical amplitude and photon observables

Initialize both real components of the envelope. The photon energy converts optical intensity into photon flux, while group speed converts flux into density.

```
from onesim.physics.optics import PhotonEnvelope

line = Line(start=(0, 0, 0), end=(2 * np.pi, 0, 0))
model = PhotonEnvelope(
    domain=line.line,
    wavenumber=2.0,
    group_speed=1.0,
    permittivity=2.0,
    photon_energy=0.5,
    absorption=0.1,
)
result = line_experiment(
    line,
    model,
    dict(real=lambda x: np.cos(x), imaginary=lambda x: np.sin(x)),
    FiniteVolume(diffusion=TwoPointDiffusion()),
    cells=32,
    end=0.02,
    dt=0.005,
    implicit=True,
)
flux = result.field(model.field("photon_flux_density")).values
density = result.field(model.field("photon_density")).values
np.testing.assert_allclose(flux, model.group_speed * density)
fig = profile_figure(
    "Photon envelope and flux",
    result,
    model,
    ("real", "imaginary", "photon_flux_density"),
)
plt.show()
```

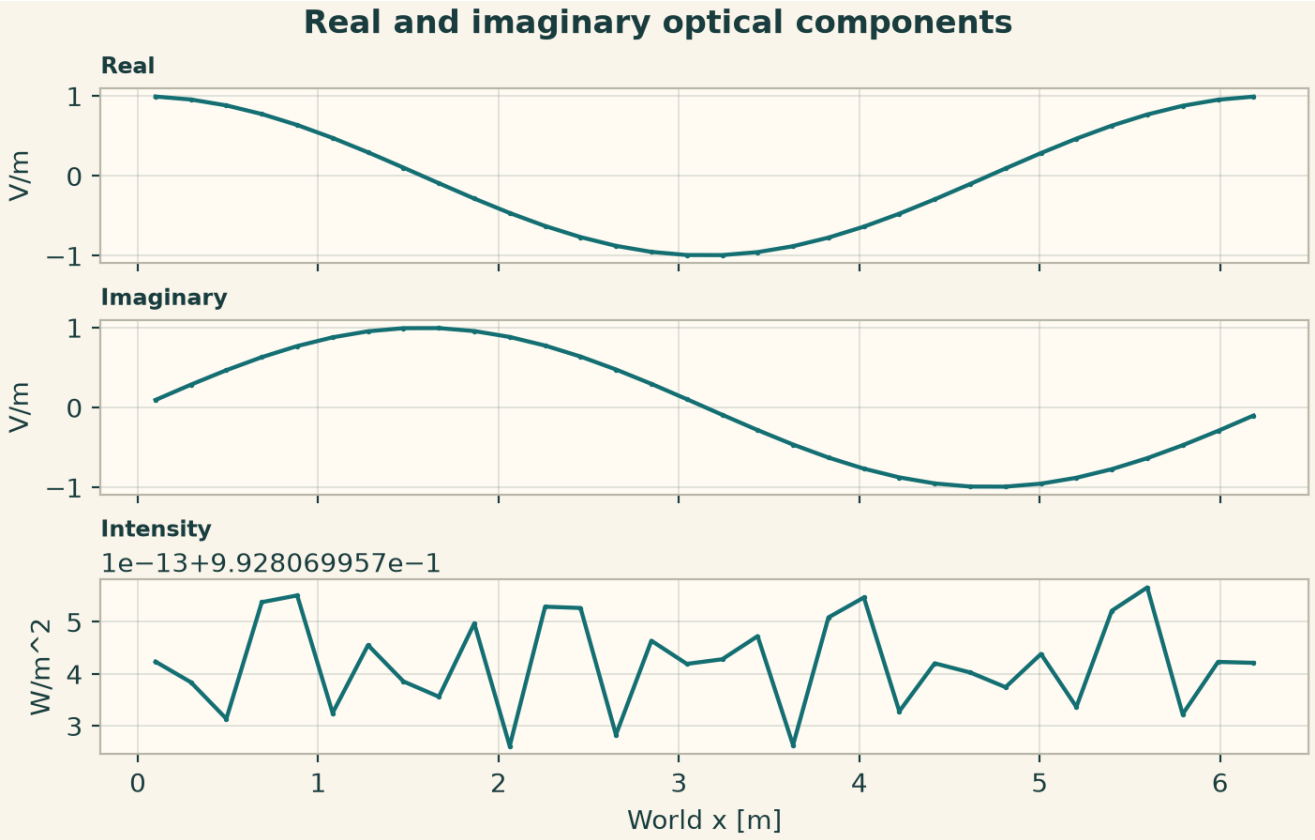


6.16.2 Real and imaginary components of a paraxial envelope

The two-real-field form stores phase through a pair of components. Recombine them only for plotting or comparison. Intensity is available directly from the model.

```
from onesim.physics.optics import ParaxialEnvelope

line = Line(start=(0, 0, 0), end=(2 * np.pi, 0, 0))
model = ParaxialEnvelope(
    domain=line.line,
    wavenumber=2.0,
    group_speed=1.0,
    permittivity=2.0,
    absorption=0.1,
)
result = line_experiment(
    line,
    model,
    dict(real=lambda x: np.cos(x), imaginary=lambda x: np.sin(x)),
    FiniteVolume(diffusion=TwoPointDiffusion()),
    cells=32,
    end=0.02,
    dt=0.005,
    implicit=True,
)
a = result.field(model.field("real")).values
b = result.field(model.field("imaginary")).values
np.testing.assert_allclose(
    result.field(model.field("intensity")).values, a * a + b * b
```

```
)
fig = profile_figure(
    "Real and imaginary optical components",
    result,
    model,
    ("real", "imaginary", "intensity"),
)
plt.show()
```

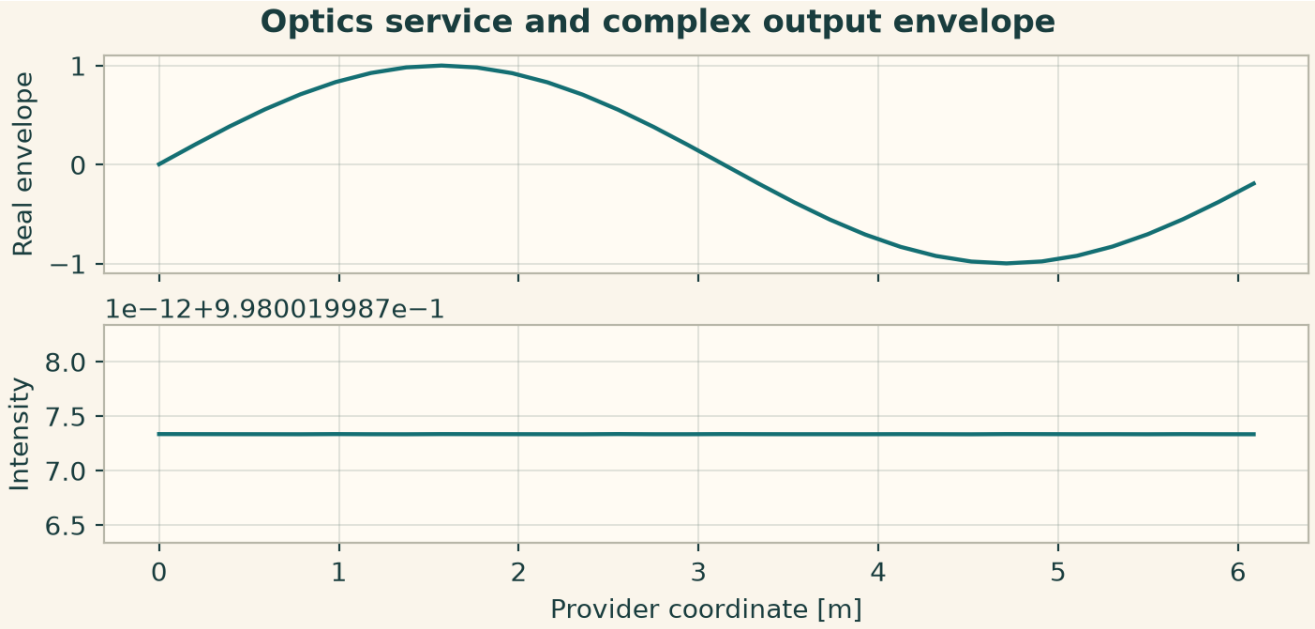


6.16.3 Optics service and complex output envelope

Supply the real and imaginary initial arrays, a wavenumber and a propagation distance. Plot the real part, magnitude and intensity from the returned complex envelope.

```
from onesim.physics.optics import OpticsConfiguration, OpticsPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = OpticsConfiguration(
    parameters=dict(
        cell_count=n,
        initial_real=wave,
        initial_imaginary=np.cos(2 * np.pi * x0),
        absorption=0.1,
        wavenumber=2,
        propagation_distance=0.01,
```

```
        step_size=0.0001,
    )
)
result = run_service(OpticsPublicApi(), configuration)
fig = service_figure(
    "Optics service and complex output envelope",
    result.coordinates,
    "Provider coordinate [m]",
    ("Real envelope", result.envelope.real),
    ("Intensity", result.intensity),
)
plt.show()
```

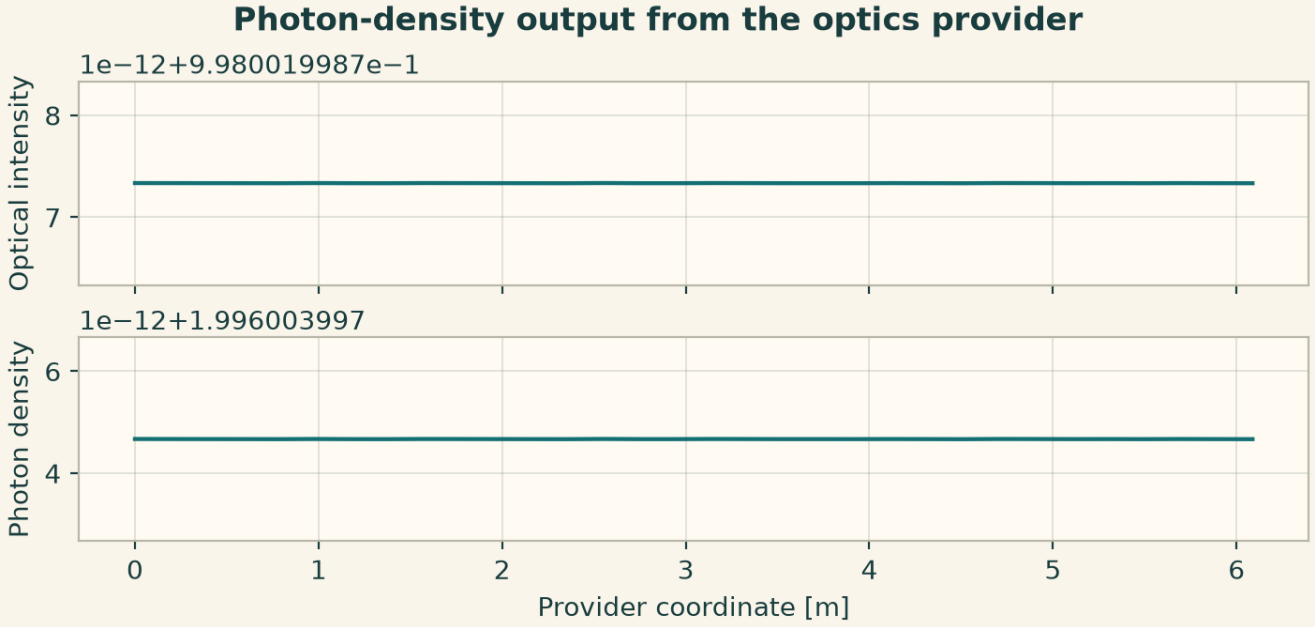


6.16.4 Photon-density output from the optics provider

Supply the same optical inputs plus photon energy. The optical result and photon density share a coordinate array.

```
from onesim.physics.optics import PhotonicsConfiguration, PhotonicsPublicApi

n = 32
x0 = np.linspace(0, 1, n, endpoint=False)
wave = np.sin(2 * np.pi * x0)
configuration = PhotonicsConfiguration(
    parameters=dict(
        cell_count=n,
        initial_real=wave,
        initial_imaginary=np.cos(2 * np.pi * x0),
        absorption=0.1,
        wavenumber=2,
        propagation_distance=0.01,
        step_size=0.0001,
        photon_energy=0.5,
```

```
    )
)
result = run_service(PhotonicsPublicApi(), configuration)
fig = service_figure(
    "Photon-density output from the optics provider",
    result.coordinates,
    "Provider coordinate [m]",
    ("Optical intensity", result.optics_result.intensity),
    ("Photon density", result.photon_density),
)
plt.show()
```



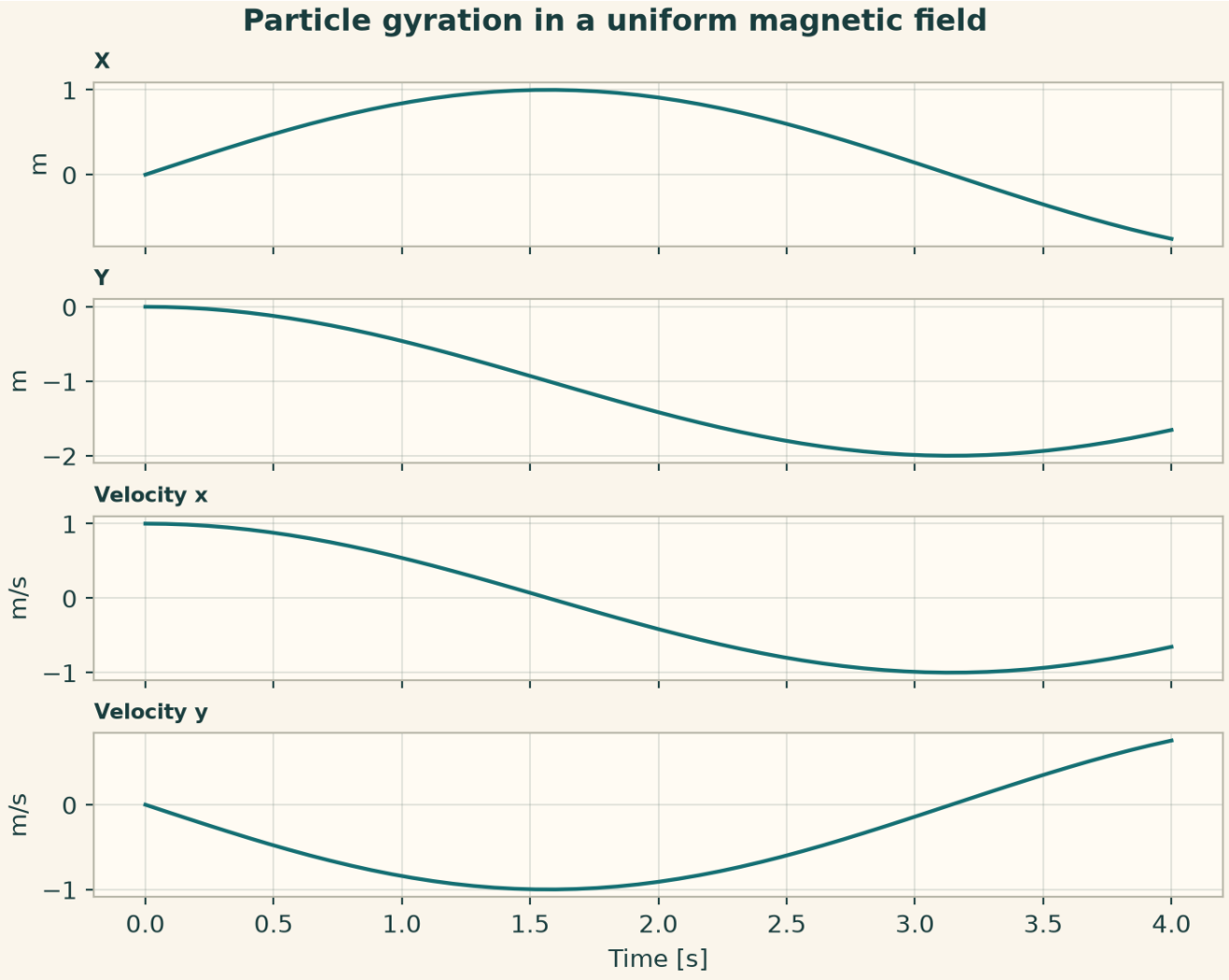
6.17 Plasma physics recipes

6.17.1 Particle gyration in a uniform magnetic field

A magnetic field bends the orbit while preserving speed in the absence of an electric field. Use the Cartesian field components as explicit inputs.

```
from onesim.physics.plasma import ChargedParticle

domain = LumpedDomain()
model = ChargedParticle(
    domain=domain.system,
    mass=1,
    charge=1,
    magnetic=(0.0, 0.0, 1.0),
)
t, data = history_case(
    domain,
    model,
    dict(x=0.0, y=0.0, z=0.0, velocity_x=1.0, velocity_y=0.0, velocity_z=0.0),
    ("x", "y", "velocity_x", "velocity_y"),
```

```
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(
    data["velocity_x"] ** 2 + data["velocity_y"] ** 2, 1.0, atol=1e-8
)
fig = history_plot("Particle gyration in a uniform magnetic field", model, t, data)
plt.show()
```

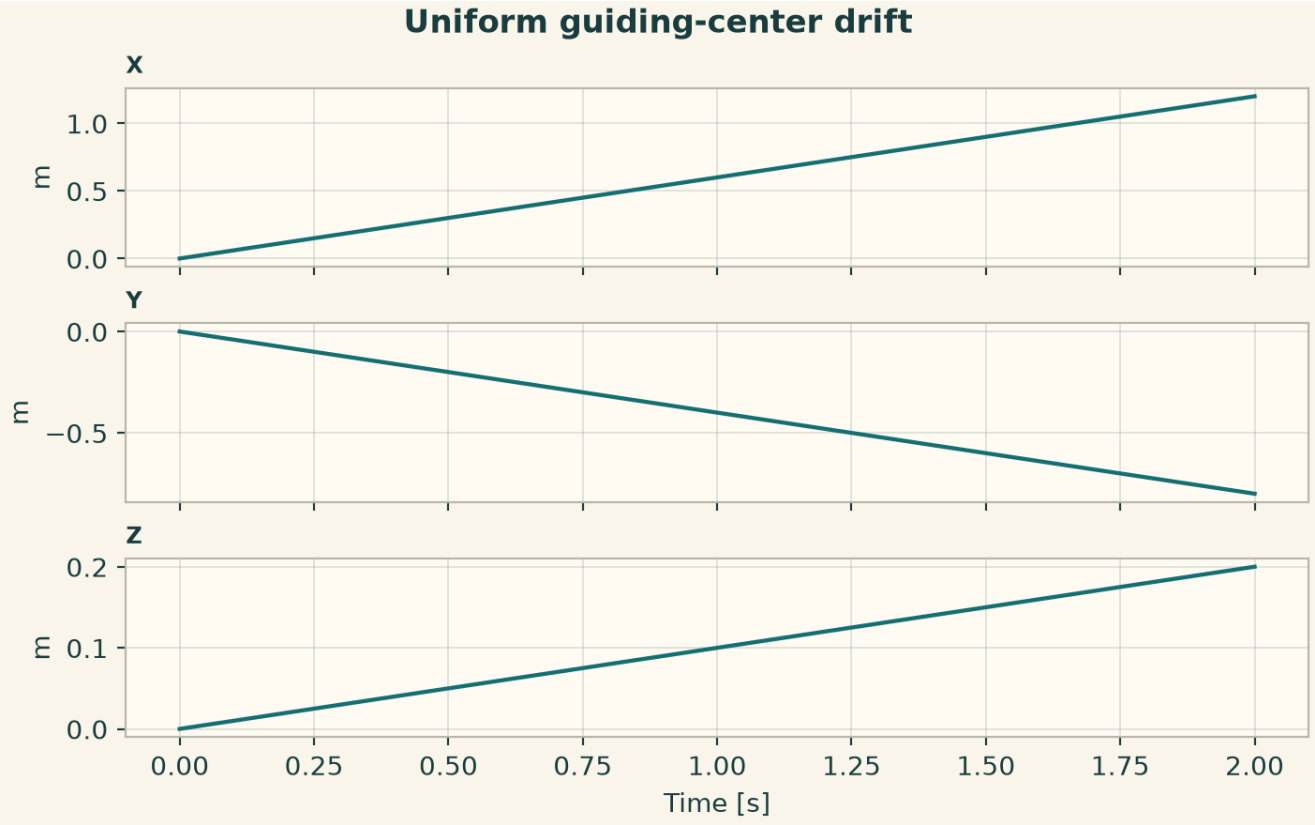


6.17.2 Uniform guiding-center drift

A constant transverse electric field and axial magnetic field produce guiding-center drift. The parallel velocity remains constant in the uniform model.

```
from onesim.physics.plasma import UniformGuidingCenter

domain = LumpedDomain()
model = UniformGuidingCenter(
    domain=domain.system,
    mass=2,
    magnetic_field=0.5,
    magnetic_moment=0.1,
```

```
    electric_x=0.2,
    electric_y=0.3,
)
t, data = history_case(
    domain,
    model,
    dict(x=0.0, y=0.0, z=0.0, parallel_velocity=0.1),
    ("x", "y", "z"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["z"], 0.1 * t, atol=1e-9)
fig = history_plot("Uniform guiding-center drift", model, t, data)
plt.show()
```

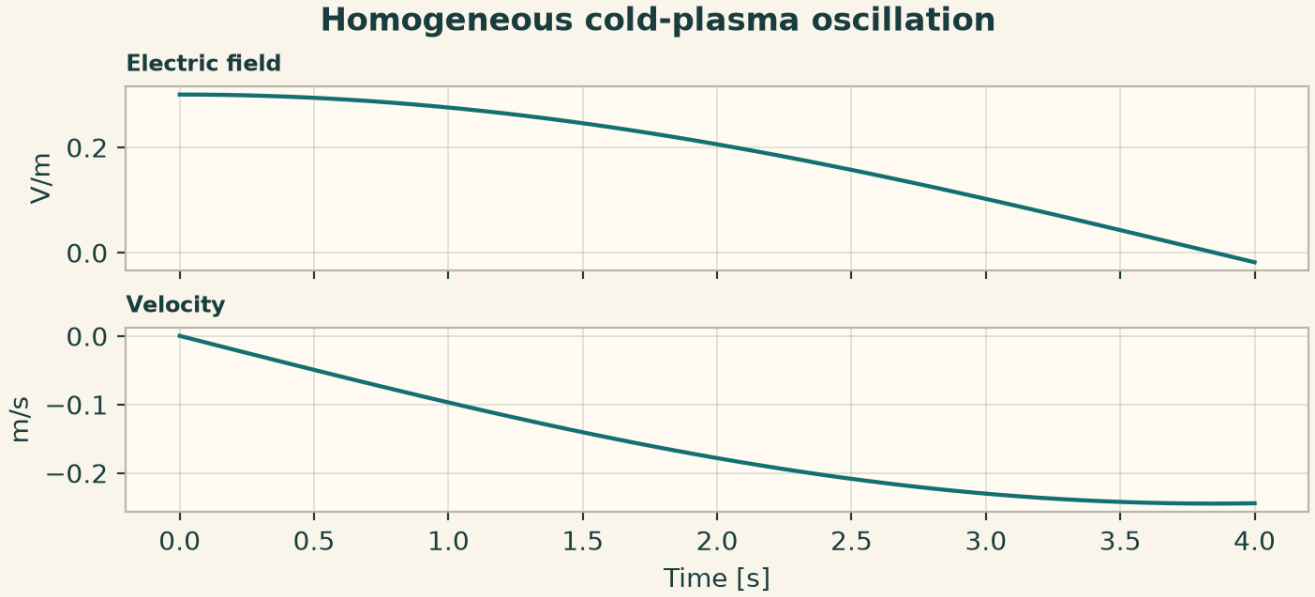


6.17.3 Homogeneous cold-plasma oscillation

The initial electric field and electron velocity exchange energy through the homogeneous plasma balance. The collision-free reference gives the plasma frequency.

```
from onesim.physics.plasma import ColdPlasmaOscillation

domain = LumpedDomain()
model = ColdPlasmaOscillation(
    domain=domain.system,
    density=2,
    electron_mass=3,
    elementary_charge=1,
```

```
    permittivity=4,
    collision_frequency=0,
)
t, data = history_case(
    domain,
    model,
    dict(velocity=0.0, electric_field=0.3, thermal_energy=0.0),
    ("electric_field", "velocity"),
    end=4,
    dt=0.02,
)
np.testing.assert_allclose(
    data["electric_field"], 0.3 * np.cos(np.sqrt(1 / 6) * t), atol=1e-9
)
fig = history_plot("Homogeneous cold-plasma oscillation", model, t, data)
plt.show()
```

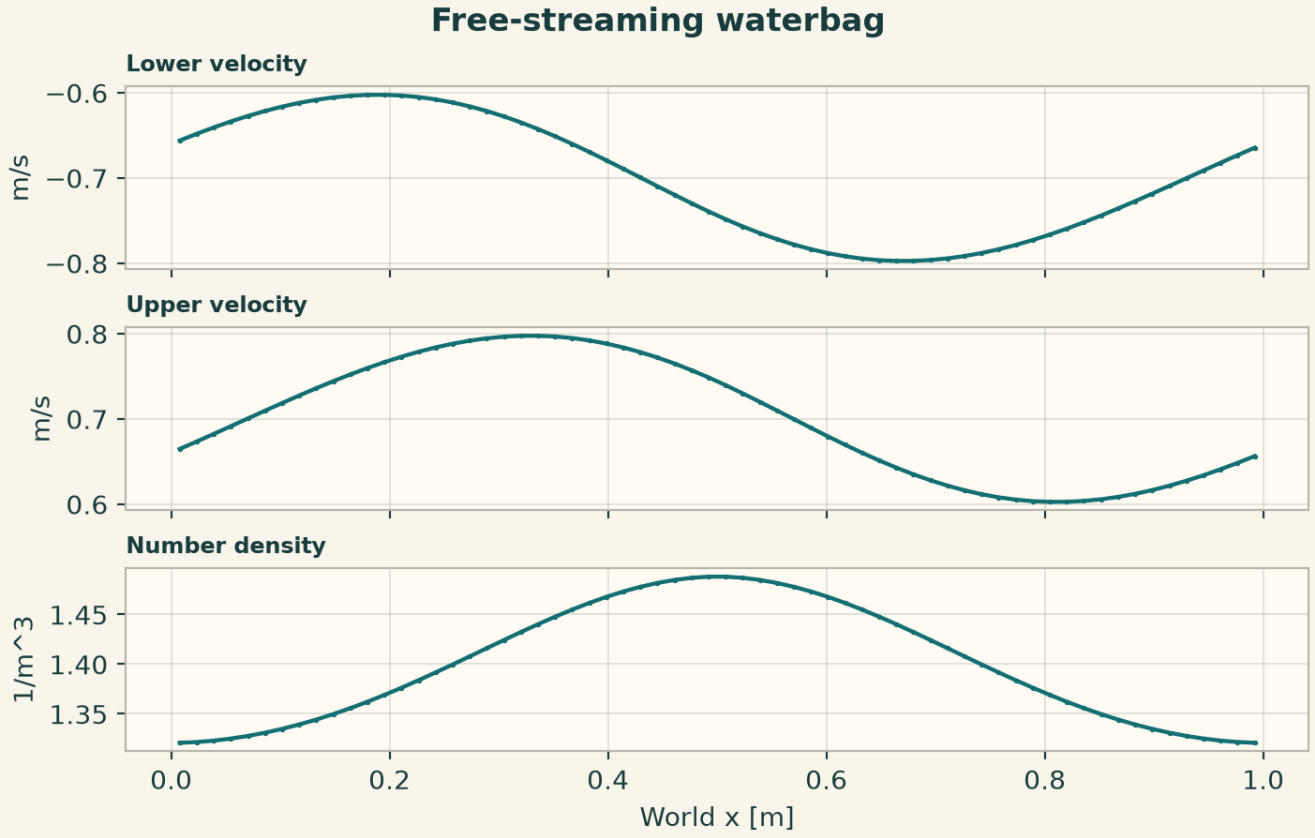


6.17.4 Free-streaming velocity contours

The phase density is constant between the two evolving contour speeds. Check that their ordering is preserved and inspect number density and bulk velocity.

```
from onesim.physics.plasma import FreeStreamingWaterbag

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = FreeStreamingWaterbag(domain=line.line, phase_density=1.0, particle_mass=1.0)
result = line_experiment(
    line,
    model,
    dict(
        lower_velocity=lambda x: -0.7 + 0.1 * np.sin(2 * np.pi * x),
        upper_velocity=lambda x: 0.7 + 0.1 * np.sin(2 * np.pi * x),
    ),
    FiniteVolume(convection=Rusanov()),
```

```
    cells=64,
    end=0.1,
    dt=0.001,
)
lo = result.field(model.field("lower_velocity")).values
hi = result.field(model.field("upper_velocity")).values
assert np.all(lo < hi)
np.testing.assert_allclose(
    result.field(model.field("number_density")).values, hi - lo
)
fig = profile_figure(
    "Free-streaming waterbag",
    result,
    model,
    ("lower_velocity", "upper_velocity", "number_density"),
)
plt.show()
```

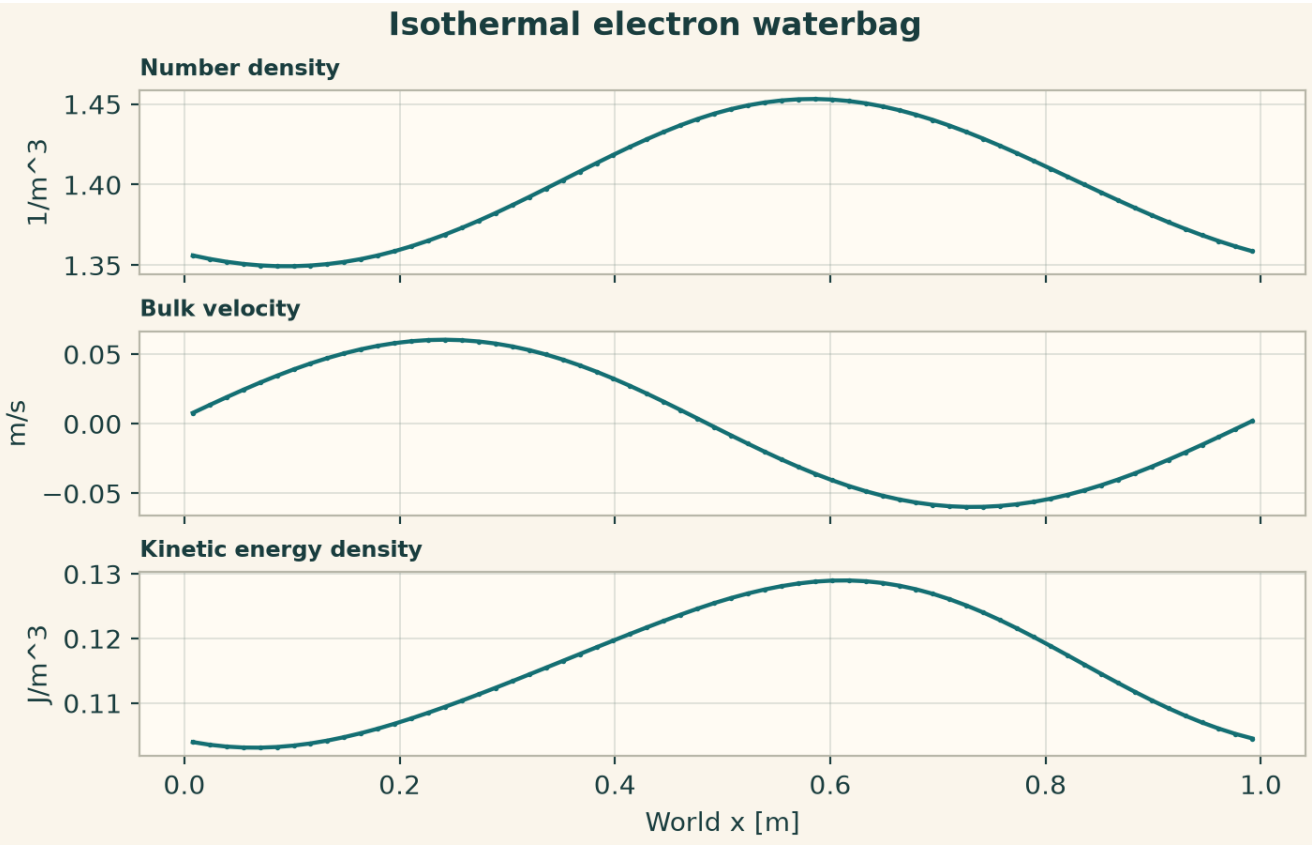


6.17.5 Isothermal electron waterbag

Add an electron-temperature closure to the two contour equations. The density is still the contour width times phase density. Resolve the fastest characteristic of the coupled contour balance.

```
from onesim.physics.plasma import IsothermalElectronWaterbag

line = Line(start=(0, 0, 0), end=(1, 0, 0))
```

```
model = IsothermalElectronWaterbag(
    domain=line.line, phase_density=1.0, particle_mass=1.0, electron_temperature=0.05
)
result = line_experiment(
    line,
    model,
    dict(
        lower_velocity=lambda x: -0.7 + 0.08 * np.sin(2 * np.pi * x),
        upper_velocity=lambda x: 0.7 + 0.05 * np.sin(2 * np.pi * x),
    ),
    FiniteVolume(convection=Rusanov()),
    cells=64,
    end=0.08,
    dt=0.0005,
)
density = result.field(model.field("number_density"))
assert density.values.min() > 0
np.testing.assert_allclose(density.integrate().value, 1.4, atol=1e-9)
fig = profile_figure(
    "Isothermal electron waterbag",
    result,
    model,
    ("number_density", "bulk_velocity", "kinetic_energy_density"),
)
plt.show()
```



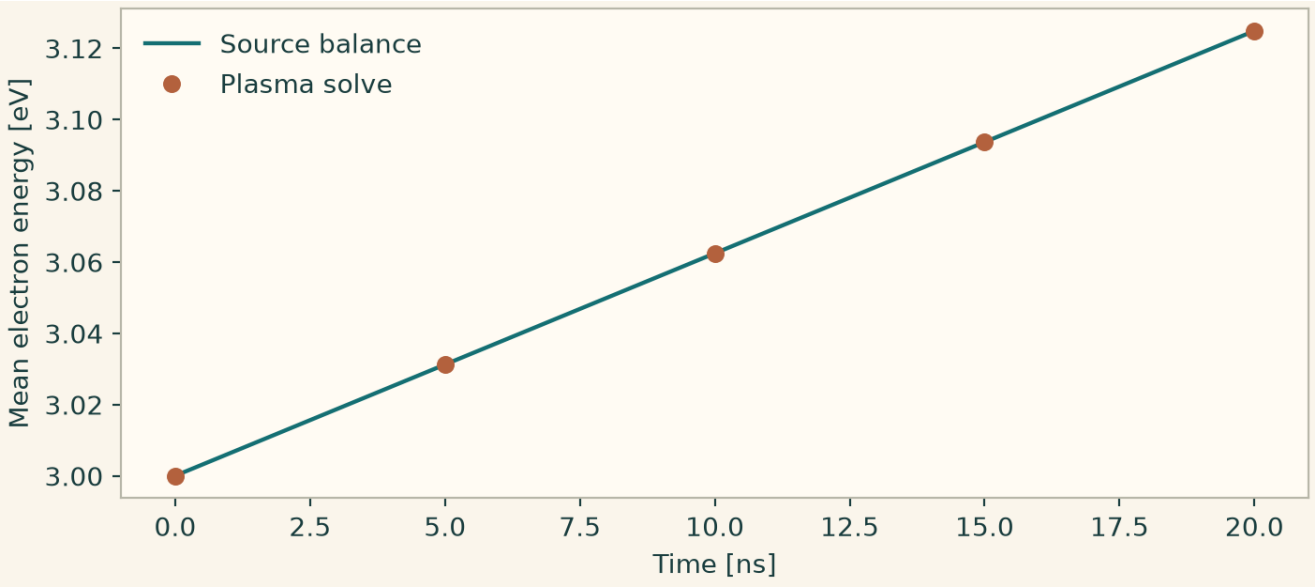
6.17.6 Insulated plasma heating with declared transport

Specify electron and ion mobility data, neutral gas and an energy input. The closed boundary preserves electron density. Compare the mean energy to the uniform source balance and convert joules to electronvolts for plotting.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Cylinder
from onesim.materials import NeutralGas
from onesim.materials.properties.units.constants import (
    ELECTRON_MASS,
    ELEMENTARY_CHARGE,
)
from onesim.meshing import StructuredCylindricalMesher
from onesim.numerics import Numerics
from onesim.numerics.fvm import FiniteVolume, ImplicitLocalSources, ScharfetterGummel
from onesim.numerics.linear import GMRES, ILU
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.chemistry import (
    Chemistry,
    ChemistryEvaluation,
    ReactionNetworkIR,
    SpeciesIR,
)
from onesim.physics.plasma import (
    DriftDiffusionPlasma,
    ElectronCollisionModel,
    IonMobilityModel,
    Maxwellian,
    MagnetizedElectronTransport,
    MeanElectronEnergyEquation,
    NoParticleFlux,
)
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue


class EmptyKinetics:
    state_independent = True

    def evaluate(self, **context):
        return ChemistryEvaluation(reaction_rates={})


cylinder = Cylinder(radius=0.01, length=0.02)
network = ReactionNetworkIR(
    "closed argon balance",
    "1",
    (
```

```
        SpeciesIR("electron", -1, ELECTRON_MASS, role="electron"),
        SpeciesIR(
            "Ar+",
            1,
            6.63e-26,
            transport={"mobility": {"value": 0.2, "unit": "m^2/V/s"}},
        ),
        SpeciesIR("Ar", 0, 6.63e-26),
    ),
    (),
    metadata={
        "electron_momentum_transfer": {
            "kind": "constant_frequency",
            "value": 1e9,
            "unit": "1/s",
        }
    },
)
chemistry = Chemistry(network=network, kinetics=EmptyKinetics())
gas = NeutralGas(species="Ar", pressure=0.5, temperature=300.0)
collisions = ElectronCollisionModel(
    gas=gas, chemistry=chemistry, energy_distribution=Maxwellian()
)
model = DriftDiffusionPlasma(
    domain=cylinder.volume,
    gas=gas,
    chemistry=chemistry,
    electron_energy=MeanElectronEnergyEquation(),
    electron_transport=MagnetizedElectronTransport(collisions=collisions),
    ion_transport=IonMobilityModel(gas=gas, chemistry=chemistry),
    prescribed_inputs={"electron_heating": 100.0},
)
mesh = StructuredCylindricalMesher(
    radial_cells=1, azimuthal_cells=4, axial_cells=1
).generate(cylinder)
problem = Problem(
    name="Closed plasma energy balance",
    geometry=cylinder,
    physics=(model,),
    boundary_conditions=(
        NoParticleFlux(boundary=cylinder.boundaries.all, physics=model),
    ),
    initial_conditions=(
        InitialValue(model.electron_density, 1e14),
        InitialValue(model.density("Ar+"), 1e14),
        InitialValue(model.mean_electron_energy, 3.0, unit="eV"),
    ),
)
sim = Simulation(
    problem=problem,
    mesh=mesh,
```

```
    numerics=Numerics(
        spatial=FiniteVolume(
            particle_flux=ScharfetterGummel(),
            energy_flux=ScharfetterGummel(),
            reaction_sources=ImplicitLocalSources(),
        ),
        nonlinear_solver=Newton(
            linear_solver=GMRES(
                preconditioner=ILU(), relative_tolerance=1e-11, absolute_tolerance=0.0
            )
        ),
        time_integrator=BackwardEuler(),
    ),
)
sim.validate().require_valid()
result = sim.run(start_time=0, end_time=2e-8, time_step=5e-9)
times = np.asarray(result.times)
energy = result.field(model.mean_electron_energy)
values = np.array([energy.evaluate(mesh.cell_centers, time=t).mean() for t in times])
exact = 3 + 100 * times / (1e14 * ELEMENTARY_CHARGE)
np.testing.assert_allclose(values / ELEMENTARY_CHARGE, exact, rtol=2e-10)
theme.use_theme()
fig, ax = plt.subplots(figsize=(7.2, 3.2), layout="constrained")
ax.plot(1e9 * times, exact, label="Source balance")
ax.plot(1e9 * times, values / ELEMENTARY_CHARGE, "o", label="Plasma solve")
ax.set(xlabel="Time [ns]", ylabel="Mean electron energy [eV]")
ax.legend(frameon=False)
plt.show()
```

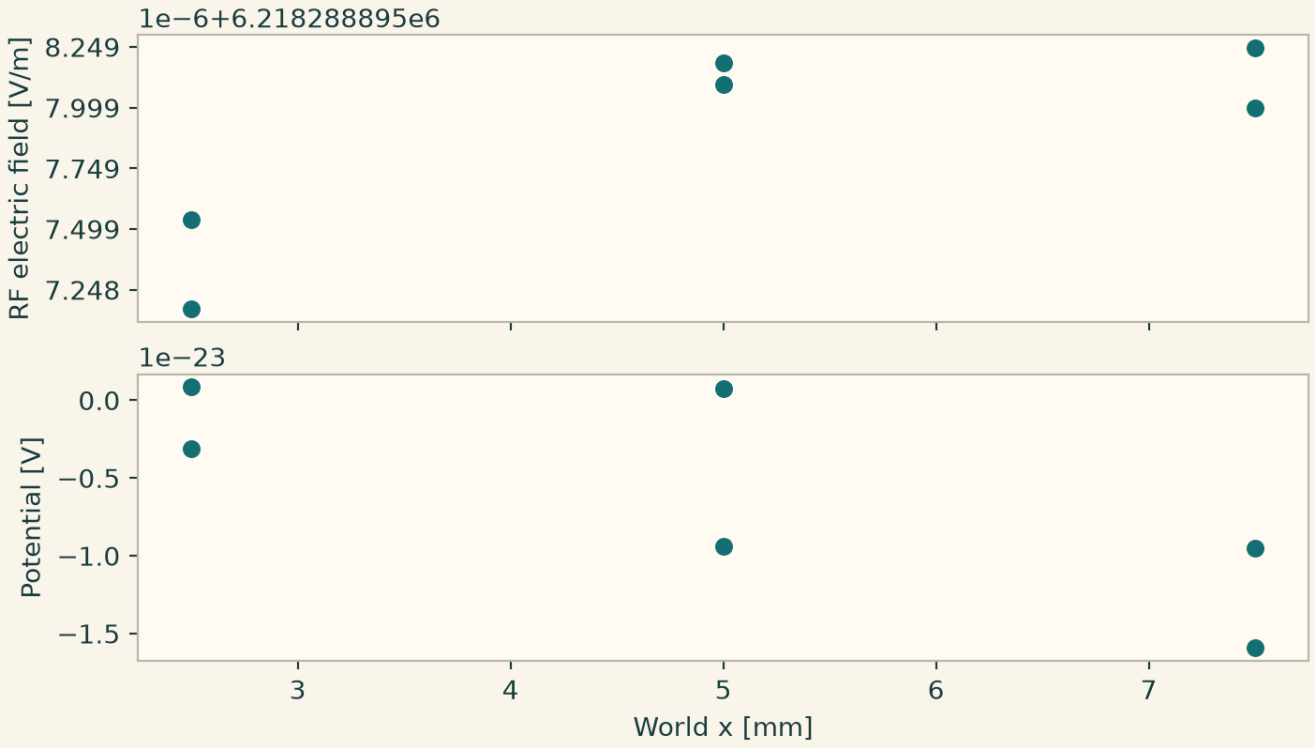


6.17.7 Native ECR with complex Maxwell and conserved plasma state

The example uses the source checkout's hydrogen chemistry and an explicit reflecting wall with grounded potential. The short run checks native coupling and field reconstruction. The catalogue rates and idealized box define a numerical example rather than an ECR device calibration.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Nedelec
from onesim.numerics.fvm import FiniteVolume
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.plasma import (
    EcrExcitation,
    NativeEcrDiscretization,
    NativeEcrPlasma,
)
from onesim.physics.plasma.data.chemistry_library import load_chemistry
from onesim.physics.plasma.fluid.models.declarations.conditions import (
    PotentialBoundaryCondition,
    SpeciesBoundaryCondition,
)
from onesim.physics.plasma.fluid.models.declarations.species.multispecies import (
    MultispeciesTransportModel,
)
from onesim.physics.plasma.walls.binding.boundary_binding import PlasmaBoundaryBinding
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

box = Box(length=0.01, width=0.01, height=0.01)
names = ("e", "H2", "H2+")
chemistry = load_chemistry("hydrogen", species=names)
wall = PlasmaBoundaryBinding(
    "wall",
    marker_id="wall",
    conditions=tuple(
        SpeciesBoundaryCondition(
            "reflect:" + n,
            "wall",
            n,
            "density",
            "reflecting",
            value={"gas_temperature": 350.0},
        )
        for n in names
    ),
    potential=PotentialBoundaryCondition("ground", "ground"),
)
model = NativeEcrPlasma(
    domain=box.volume,
    chemistry=chemistry,
    transport_model=MultispeciesTransportModel(neutral_diffusivity=(("H2", 0.01),)),
```

```
    excitation=EcrExcitation(
        frequency=2.45e9, power=1e-4, impressed_current_density=(1.0, 0.2, 0.3)
    ),
    walls=((box.boundaries.all, wall),),
    plasma_parameters={"magnetic_flux_density": (0, 0, 0.05)},
)
initial = tuple(
    InitialValue(model.density(n), v) for n, v in zip(names, (1e14, 1e20, 1e14))
)
mesh = StructuredTetrahedralMesher(cells=(1, 1, 1)).generate(box)
sim = Simulation(
    problem=Problem(
        name="Native ECR",
        geometry=box,
        physics=(model,),
        initial_conditions=initial
        + (InitialValue(model.electron_energy_density, 4.5e14),),
    ),
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteVolume(
            provider=NativeEcrDiscretization(
                maxwell=FiniteElement(element=Nedelec(order=2))
            )
        ),
        time_integrator=BackwardEuler(),
        nonlinear_solver=Newton(),
        linear_solver=SparseDirect(),
    ),
)
sim.validate().require_valid()
result = sim.run(start_time=0, end_time=2e-12, time_step=1e-12)
assert len(result.native_states) == 3
x = np.asarray(mesh.coordinates)[np.asarray(mesh.cells)].mean(axis=1)[:, 0]
electric = np.asarray(result.field(model.electric_field).values)
field = np.asarray(result.field(model.potential).values).ravel()
assert electric.shape == (x.size, 3) and field.shape == x.shape
assert np.isfinite(electric).all() and np.isfinite(field).all()
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
a.scatter(1e3 * x, np.linalg.norm(electric, axis=-1))
a.set_ylabel("RF electric field [V/m]")
b.scatter(1e3 * x, field)
b.set(xlabel="World x [mm]", ylabel="Potential [V]")
plt.show()
```

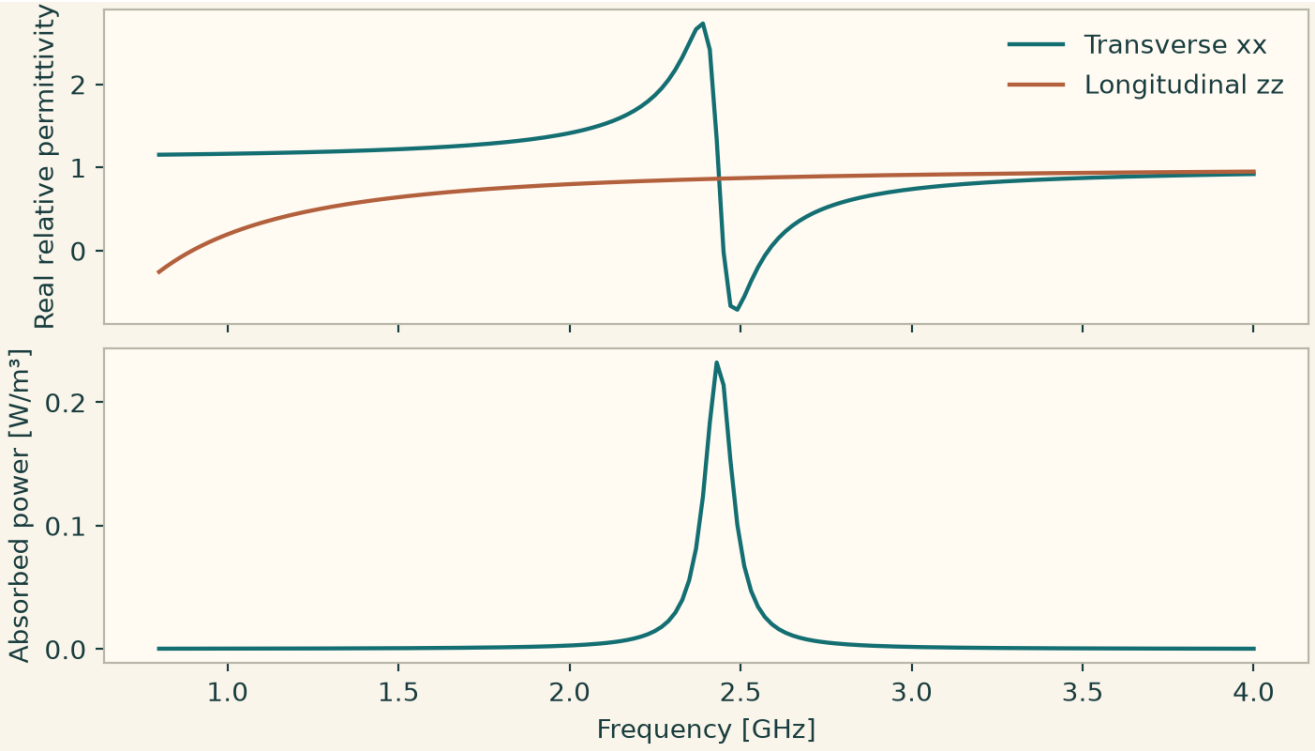


6.17.8 Frequency sweep of the cold-plasma tensor

A collision collaborator supplies a frequency from the electron energy. Sweep the microwave frequency at fixed magnetic field and compare transverse and longitudinal permittivity. Absorbed power uses the same constitutive response and the selected peak phasor convention.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.materials.properties.units.constants import (
    ELEMENTARY_CHARGE,
    VACUUM_PERMITTIVITY,
)
from onesim.physics.plasma import MagnetizedColdPlasma


class ConstantCollisions:
    def collision_frequency(self, *, mean_electron_energy, points, time):
        return 3e8


plasma = MagnetizedColdPlasma(collision_model=ConstantCollisions())
frequencies = np.linspace(0.8e9, 4.0e9, 160)
response = [
    plasma.evaluate(
        electron_density=np.array([1e16]),
        mean_electron_energy=np.array([3 * ELEMENTARY_CHARGE]),
        magnetic_field=np.array([[0.0, 0.0, 0.087]]),
        frequency=f,
        phasor_convention="exp(-i*omega*t)",
        amplitude_convention="peak",
```

```
    )
    for f in frequencies
]
relative = np.array([r.permittivity[0] / VACUUM_PERMITTIVITY for r in response])
power = np.array(
    [r.absorbed_power_density(np.array([[1.0, 0.0, 0.0]]))[0] for r in response]
)
assert np.isfinite(relative).all() and power.min() >= 0
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
a.plot(frequencies / 1e9, relative[:, 0, 0].real, label="Transverse xx")
a.plot(frequencies / 1e9, relative[:, 2, 2].real, label="Longitudinal zz")
a.set_ylabel("Real relative permittivity")
a.legend(frameon=False)
b.plot(frequencies / 1e9, power)
b.set(xlabel="Frequency [GHz]", ylabel="Absorbed power [W/m³]")
plt.show()
```



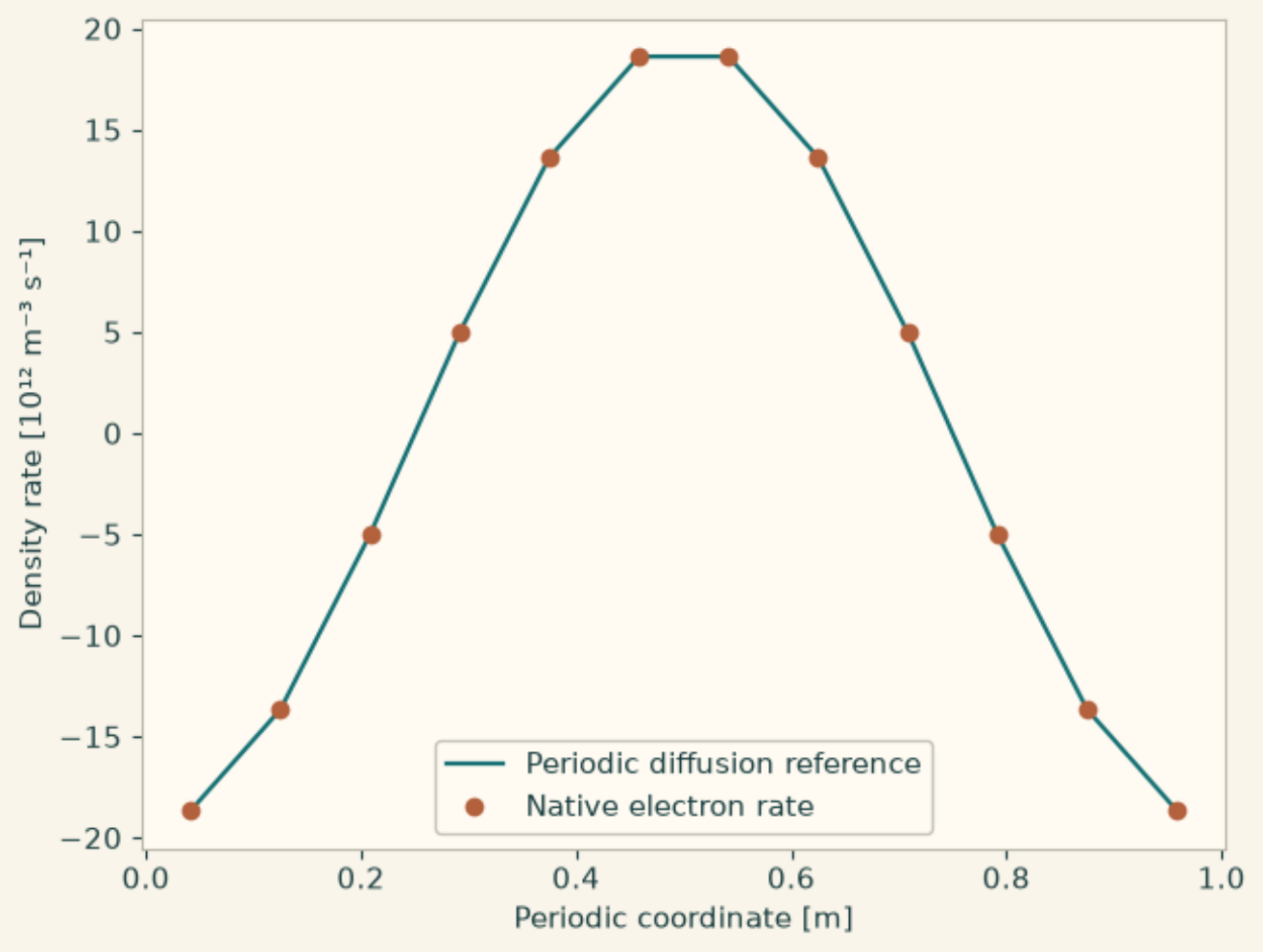
6.17.9 Prepare periodic plasma transport with an explicit Poisson gauge

Construct the wall-free periodic face topology and wrap its real cell operator with a global mean-potential constraint. The manufactured electron and ion populations start locally neutral and diffuse with equal coefficients. Verify the Gauss residual, the periodic diffusion rate and conservation of particle number.

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

```
theme.use_theme()

from onesim.materials.properties.units.constants import ELEMENTARY_CHARGE
from onesim.numerics.discretization.equations.contracts.operators.operator_contracts
    import SpatialExecutionContext
from onesim.numerics.fvm.contracts.coupled_cell_transport import CellCoefficients,
    CoupledNumericalConfiguration
from onesim.numerics.fvm.spatial.transport.coupled_cell_transport import
    CoupledCellTransportService
from onesim.physics.plasma import operators as plasma_ops
from onesim.simulation.state.history.state_snapshot import StateSnapshot

shape = (12, 3, 3)
extents = (1.0, 1.0, 1.0)
count = int(np.prod(shape))
dx = extents[0] / shape[0]
x_cells = (np.arange(shape[0]) + 0.5)*dx
density = np.broadcast_to((1e14*(1+0.1*np.cos(2*np.pi*x_cells)))[:, None, None], shape)
diffusivity = 0.05  # m²/s
epsilon = 8.8541878128e-12

def coefficients(state, time):
    return CellCoefficients(
        mobility=np.zeros_like(state),
        diffusivity=np.full_like(state, diffusivity),
        charge_number=(-1.0, 1.0, 0.0),
    )

def volume_rates(state, time):
    rates = np.zeros_like(state)
    rates[:, 2] = ELEMENTARY_CHARGE * (state[:, 1] - state[:, 0])
    return rates

request = plasma_ops.periodic_cell_transport_request(
    shape, extents, initial_fields=(density, density, np.zeros(shape)),
        potential_field_index=2,
    permittivity=epsilon, coefficients=coefficients, volume_rates=volume_rates,
)
assert not request.boundary_faces and len(request.internal_faces) == 3*count
field_ids = ("electron_density", "ion_density", "potential")
provider = CoupledCellTransportService().operator(
    request, CoupledNumericalConfiguration(),
    field_ids=field_ids, field_units=("1/m^3", "1/m^3", "V"),
)
gauss = plasma_ops.periodic_cell_poisson_dae_constraint(
    shape, extents, permittivity=epsilon,
    charge_fields={"electron_density": -ELEMENTARY_CHARGE, "ion_density":
        ELEMENTARY_CHARGE},
)
adapter = plasma_ops.periodic_cell_transport_dae(provider, gauss)
values = dict(zip(field_ids, request.initial_fields, strict=True))
```

```
values[gauss.gauge_field] = 0.0
snapshot = StateSnapshot("periodic-plasma", 0.0, values)
context = SpatialExecutionContext(gauss.to_carrier().carrier_id, snapshot.state_type,
    0.0)
balance = adapter.residual(snapshot, context).values
rates = np.asarray(balance.differential_rate(snapshot, 0.0)).reshape(2, *shape)
np.testing.assert_allclose(balance.algebraic_residual(snapshot, 0.0), 0.0, atol=1e-20)
np.testing.assert_allclose(rates[0], rates[1], atol=1e-2, rtol=1e-12)
expected = diffusivity * (np.roll(density, 1, axis=0) - 2*density + np.roll(density,
    -1, axis=0)) / dx**2
np.testing.assert_allclose(rates[0], expected, atol=5.0, rtol=1e-12)
np.testing.assert_allclose(np.sum(rates[0]) / np.sum(np.abs(rates[0])), 0.0,
    atol=1e-13)
print("Differential fields", balance.differential_field_ids)
print("Algebraic fields", balance.algebraic_field_ids)
fig, ax = plt.subplots(layout="constrained")
ax.plot(x_cells, expected[:, 0, 0] / 1e12, "-", label="Periodic diffusion reference")
ax.plot(x_cells, rates[0, :, 0, 0] / 1e12, "o", label="Native electron rate")
ax.set(xlabel="Periodic coordinate [m]", ylabel="Density rate [10¹² m⁻³ s⁻¹]")
ax.legend()
plt.show()
```

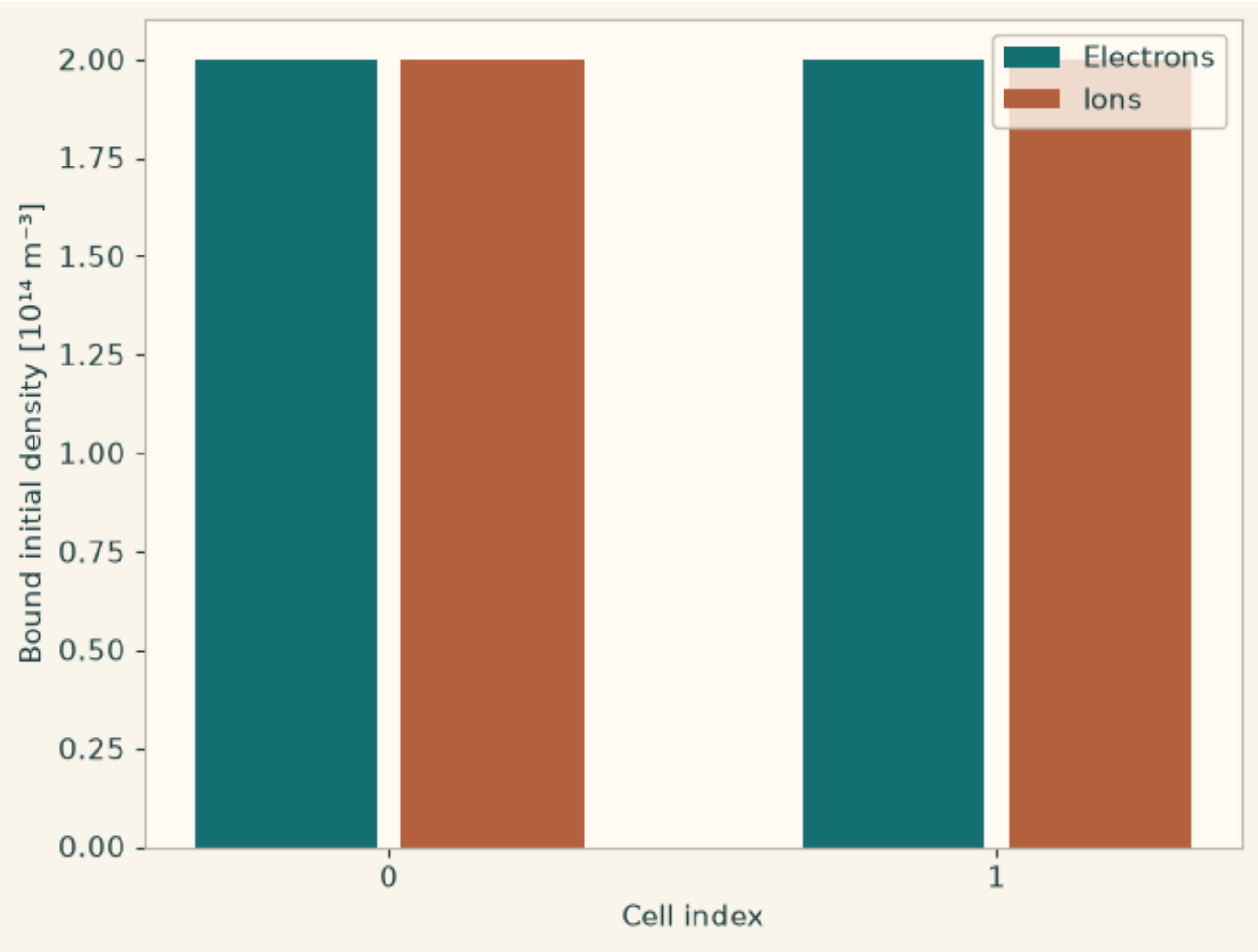


6.17.10 Bind a native plasma operator and translate its initial conditions

Build a bounded hydrogen plasma provider, then use the condition binder to replace the initial densities with explicit semantic declarations. Evaluate the differential rates and the grounded Poisson constraint through the cell DAE adapter. The local hydrogen catalogue includes estimated coefficients. This case checks the operator handoff and state ownership.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
theme.use_theme()

from types import SimpleNamespace
from onesim.numerics.discretization.equations.contracts.equations.problem_ir import
    InitialConditionIR
from onesim.numerics.discretization.equations.contracts.operators.operator_contracts
    import SpatialExecutionContext
from onesim.numerics.discretization.geometry.carriers.kernels.simplex.tetrahedral
    import (
    tetrahedral_dual_faces, tetrahedral_geometry,
)
from onesim.numerics.fvm.contracts.coupled_cell_transport import BoundaryFace
from onesim.physics.plasma import operators as plasma_ops
from onesim.physics.plasma.data.chemistry_library import load_chemistry
from onesim.physics.plasma.fluid.models.declarations.conditions import (
    PotentialBoundaryCondition, SpeciesBoundaryCondition,
)
from onesim.physics.plasma.fluid.models.declarations.species.multispecies import (
    MultispeciesDriftDiffusionProblem, MultispeciesTransportModel,
)
from onesim.physics.plasma.fluid.numerics.backends.transport.multispecies_drift_diffusi
    on import (
    MultispeciesDriftDiffusionProvider,
)
from onesim.physics.plasma.walls.binding.boundary_binding import PlasmaBoundaryBinding
from onesim.simulation.state.history.state_snapshot import StateSnapshot

points = 0.01 * np.array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1], [0, 0, -1]],
    dtype=float)
cells = np.array([[0, 1, 2, 3], [0, 2, 1, 4]])
exterior = tetrahedral_dual_faces(tetrahedral_geometry(points, cells))[1]
faces = tuple(
    BoundaryFace(face.owner_cell, face.area, tuple(face.normal), tuple(face.centroid),
                 face.centre_distance, "wall", "")
    for face in exterior
)
species = ("e", "H2", "H2+")
wall = PlasmaBoundaryBinding(
    "wall", marker_id="wall",
    conditions=tuple(
        SpeciesBoundaryCondition(
            "reflect:"+name, "wall", name, "density", "reflecting",
```

```
            value={"gas_temperature": 350.0},
        )
        for name in species
    ),
    potential=PotentialBoundaryCondition("ground", "ground"),
)
provider = MultispeciesDriftDiffusionProvider(
    chemistry=load_chemistry("hydrogen", species=species), boundaries=(wall,),
    transport_model=MultispeciesTransportModel(neutral_diffusivity=(("H2", 0.01),)),
    require_full_boundary_coverage=False,
)
problem = MultispeciesDriftDiffusionProblem(
    points, cells, faces, tuple(zip(species, (1e14, 1e20, 1e14), strict=True)),
    3.0, magnetic_flux_density=(0, 0, 0.05),
)
native = plasma_ops.plasma_operator(provider, problem, final_time=1e-11)
binding = plasma_ops.plasma_condition_binding(provider, problem, final_time=1e-11)
conditions = tuple(
    InitialConditionIR(
        "seed-"+name, name, ("discharge",), value, "1/m^3", value_kind="constant",
    )
    for name, value in (("e", "2e14"), ("H2", "1e20"), ("H2+", "2e14"))
)
bound = binding.bind(
    spatial_operator=native, problem=SimpleNamespace(initial_conditions=conditions),
    initial_state=None,
)
assert bound.applied_initial_ids == tuple(condition.initial_id for condition in
    conditions)
adapter = plasma_ops.cell_transport_dae(bound.spatial_operator)
fields = dict(zip(adapter.state_field_ids, np.asarray(bound.initial_state).T,
    strict=True))
snapshot = StateSnapshot("bounded-plasma", 0.0, fields)
context = SpatialExecutionContext("discharge", snapshot.state_type, 0.0)
balance = adapter.residual(snapshot, context).values
np.testing.assert_allclose(snapshot.field("e"), 2e14)
np.testing.assert_allclose(snapshot.field("H2+"), 2e14)
np.testing.assert_allclose(balance.algebraic_residual(snapshot, 0.0), 0.0, atol=1e-20)
assert np.isfinite(balance.differential_rate(snapshot, 0.0)).all()
print("Applied initial IDs", bound.applied_initial_ids)
print("Differential fields", balance.differential_field_ids)
print("Algebraic fields", balance.algebraic_field_ids)
fig, ax = plt.subplots(layout="constrained")
cell_index = np.arange(len(cells))
ax.bar(cell_index - 0.17, fields["e"]/1e14, width=0.3, label="Electrons")
ax.bar(cell_index + 0.17, fields["H2+"]/1e14, width=0.3, label="Ions")
ax.set(xlabel="Cell index", xticks=cell_index, ylabel="Bound initial density [10¹⁴
    m⁻³]")
ax.legend()
plt.show()
```



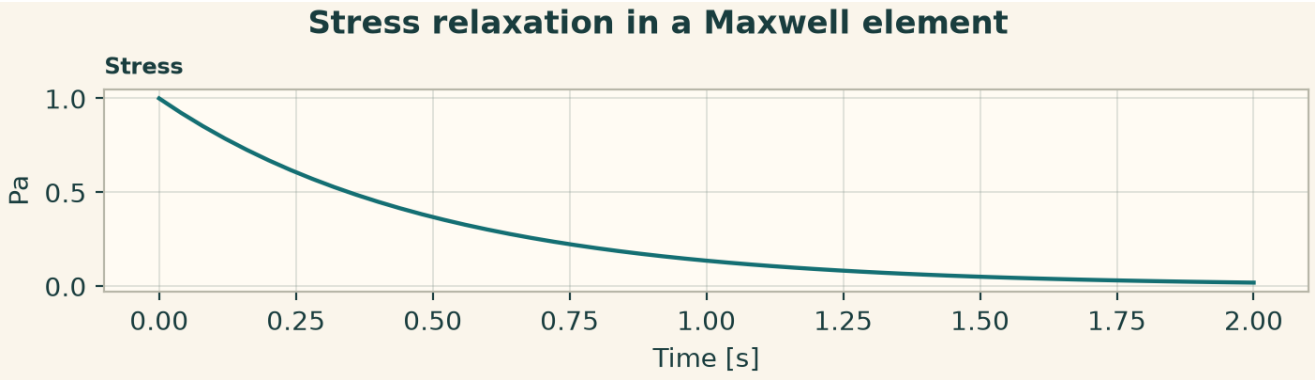
6.18 Solid mechanics recipes

6.18.1 Stress relaxation in a Maxwell element

Hold strain rate at zero and initialize stress. The relaxation time equals viscosity divided by Young's modulus.

```
from onesim.physics.solid_mechanics import MaxwellRelaxation

domain = LumpedDomain()
model = MaxwellRelaxation(
    domain=domain.system,
    youngs_modulus=4,
    viscosity=2,
    strain_rate=0,
)
t, data = history_case(
    domain,
    model,
    dict(stress=1.0),
    ("stress",),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["stress"], np.exp(-2 * t), atol=1e-7)
```

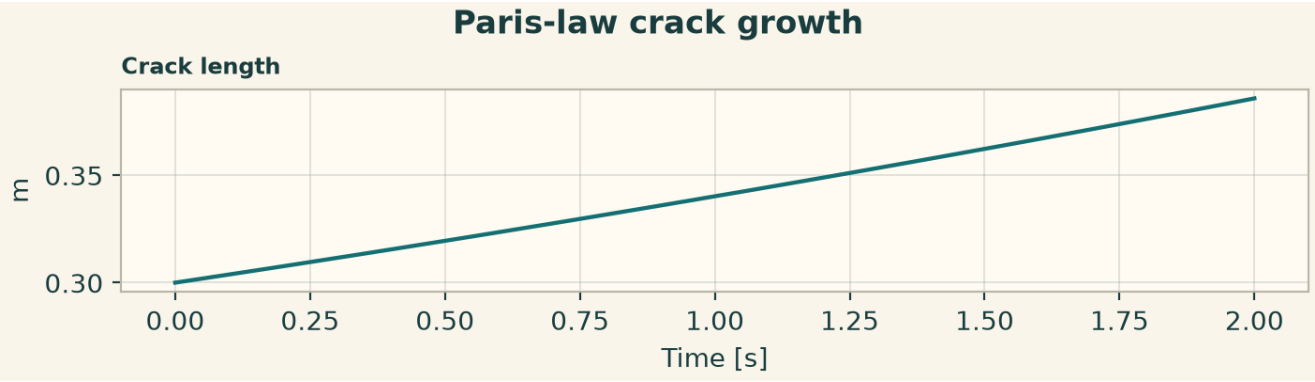
```
fig = history_plot("Stress relaxation in a Maxwell element", model, t, data)
plt.show()
```



6.18.2 Paris-law crack growth

Use a constant stress range and cycling frequency. At exponent two the stress-intensity law gives exponential crack growth for a fixed geometry factor.

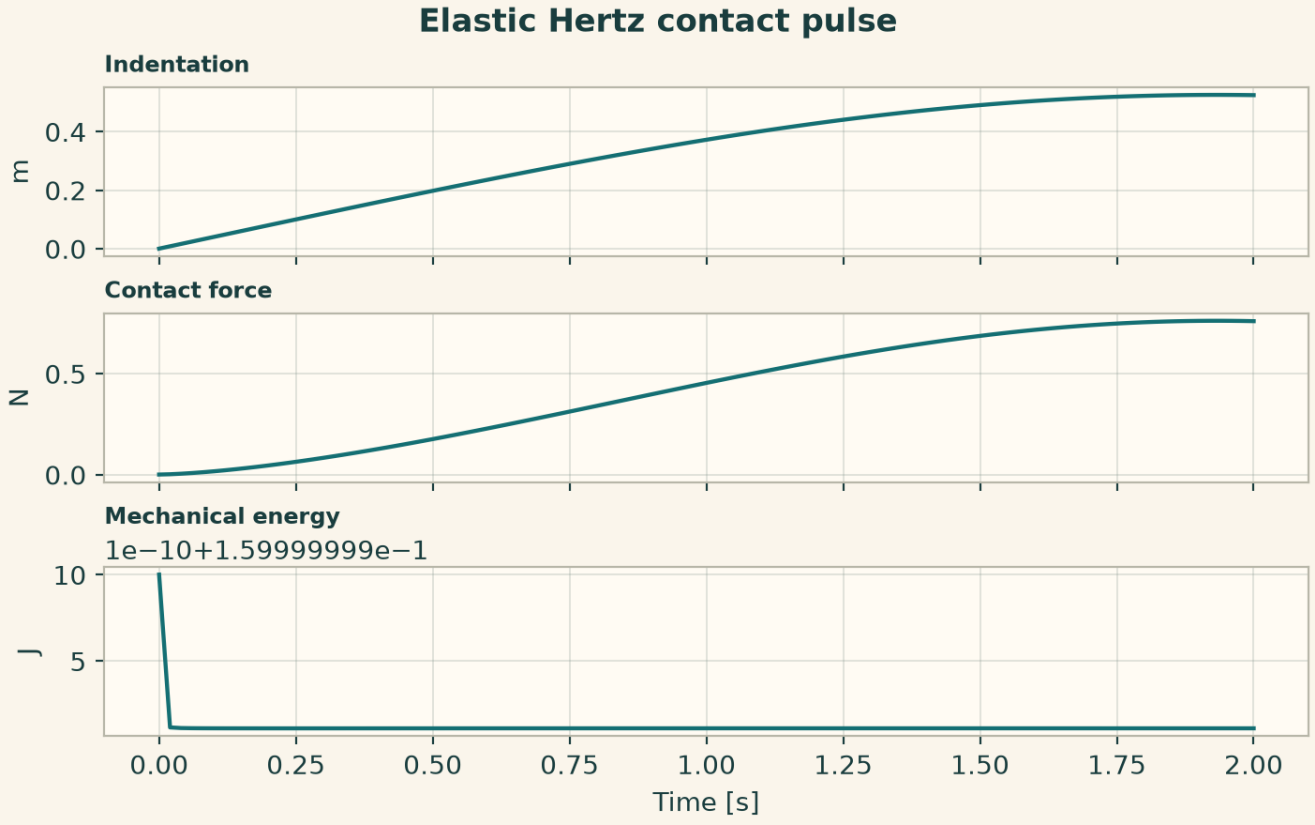
```
from onesim.physics.solid_mechanics import FatigueCrackGrowth, ParisLaw

domain = LumpedDomain()
model = FatigueCrackGrowth(
    domain=domain.system,
    stress_range=1,
    frequency=2,
    growth_law=ParisLaw(coefficient=0.02, exponent=2),
)
t, data = history_case(
    domain,
    model,
    dict(crack_length=0.3),
    ("crack_length",),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(
    data["crack_length"], 0.3 * np.exp(0.04 * np.pi * t), atol=1e-9
)
fig = history_plot("Paris-law crack growth", model, t, data)
plt.show()
```



6.18.3 Elastic Hertz contact pulse

Initialize positive approach velocity at first contact. The force follows positive indentation to the three-halves power and the isolated energy stays nearly constant.

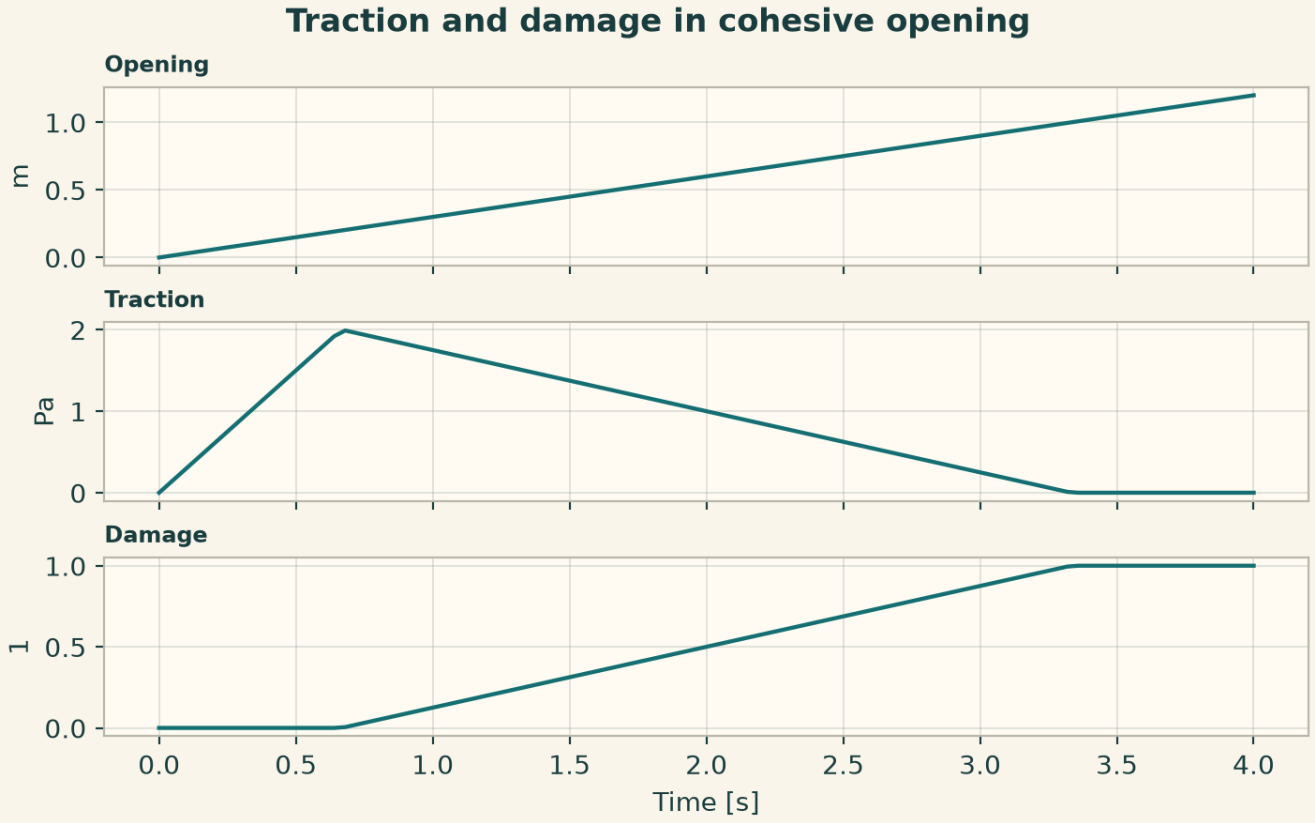
```
from onesim.physics.solid_mechanics import HertzImpact, HertzContactLaw

domain = LumpedDomain()
model = HertzImpact(
    domain=domain.system,
    reduced_mass=2,
    contact_law=HertzContactLaw(effective_modulus=3, effective_radius=0.25),
)
t, data = history_case(
    domain,
    model,
    dict(indentation=0.0, approach_velocity=0.4),
    ("indentation", "contact_force", "mechanical_energy"),
    end=2,
    dt=0.005,
)
np.testing.assert_allclose(
    data["contact_force"], 2 * np.maximum(data["indentation"], 0) ** 1.5, atol=1e-10
)
fig = history_plot("Elastic Hertz contact pulse", model, t, data)
plt.show()
```



6.18.4 Traction and damage in cohesive opening

Drive the opening monotonically through the elastic and softening regimes. The work integral accumulates from the traction and the prescribed opening rate.

```
from onesim.physics.solid_mechanics import CohesiveOpening, BilinearCohesiveLaw

domain = LumpedDomain()
model = CohesiveOpening(
    domain=domain.system,
    opening_rate=0.3,
    cohesive_law=BilinearCohesiveLaw(
        peak_traction=2, onset_opening=0.2, final_opening=1
    ),
)
t, data = history_case(
    domain,
    model,
    dict(opening=0.0, work_per_area=0.0),
    ("opening", "traction", "damage"),
    end=4,
    dt=0.01,
)
assert data["damage"][-1] > data["damage"][0] and data["traction"].max() > 1.9
fig = history_plot("Traction and damage in cohesive opening", model, t, data)
plt.show()
```



6.19 Systems and networks recipes

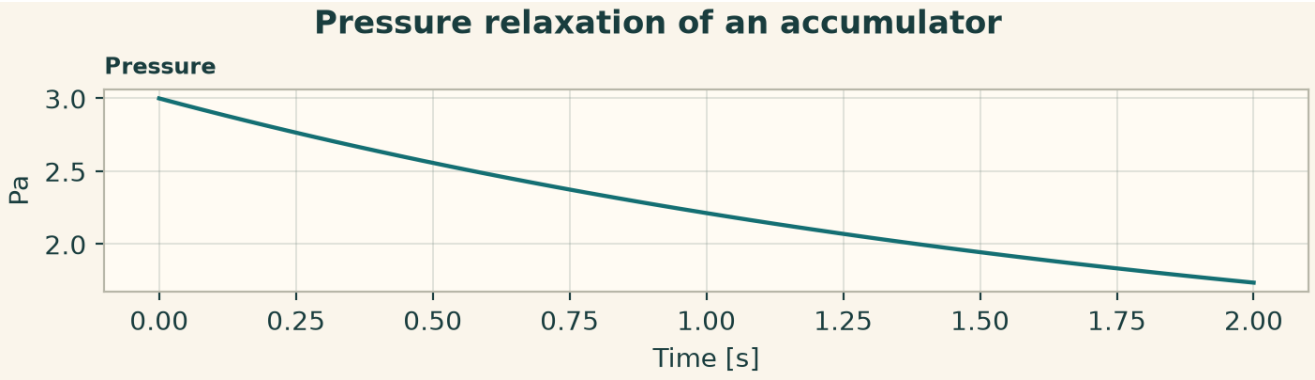
This section includes the field-based declarations and the independently callable systems and networks service configurations. The configuration keys in the service recipes belong to the indicated provider.

6.19.1 Pressure relaxation of an accumulator

A resistance connects a hydraulic compliance to a prescribed reference pressure. The pressure time constant is resistance times compliance.

```
from onesim.physics.systems import HydraulicAccumulator

domain = LumpedDomain()
model = HydraulicAccumulator(
    domain=domain.system,
    compliance=2,
    resistance=1,
    reference_pressure=1,
)
t, data = history_case(
    domain,
    model,
    dict(pressure=3.0),
    ("pressure",),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["pressure"], 1 + 2 * np.exp(-t / 2), atol=1e-9)
```

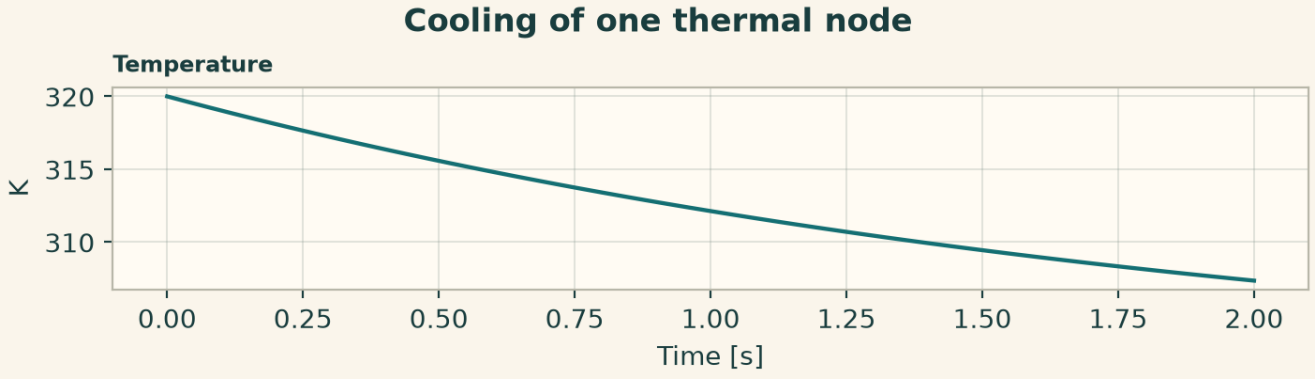
```
fig = history_plot("Pressure relaxation of an accumulator", model, t, data)
plt.show()
```



6.19.2 Cooling of one thermal node

Supply an ambient temperature and a linear conductance. The node temperature decays exponentially with capacity divided by conductance.

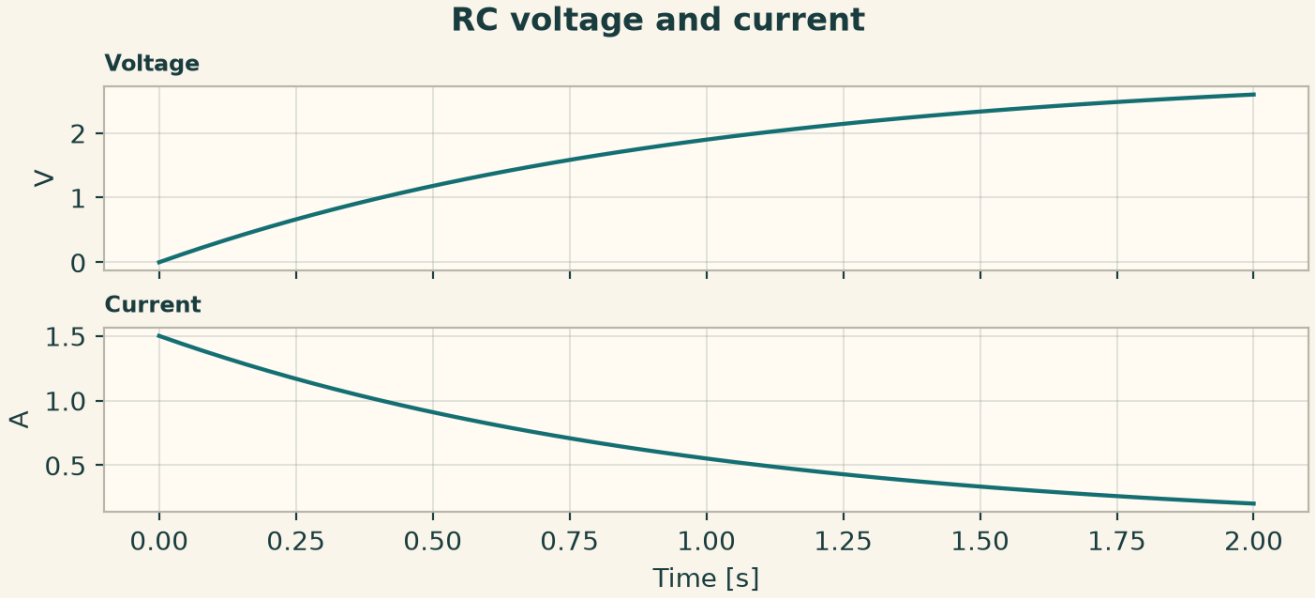
```
from onesim.physics.systems import ThermalCapacitance

domain = LumpedDomain()
model = ThermalCapacitance(
    domain=domain.system,
    heat_capacity=2,
    conductance=1,
    ambient_temperature=300,
)
t, data = history_case(
    domain,
    model,
    dict(temperature=320.0),
    ("temperature",),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["temperature"], 300 + 20 * np.exp(-t / 2), atol=1e-8)
fig = history_plot("Cooling of one thermal node", model, t, data)
plt.show()
```



6.19.3 RC voltage and current

Supply a fixed voltage across a series resistor and capacitor. The initial charging current falls as the capacitor voltage approaches the supply.

```
from onesim.physics.systems import ResistorCapacitor

domain = LumpedDomain()
model = ResistorCapacitor(
    domain=domain.system,
    resistance=2,
    capacitance=0.5,
    supply_voltage=3,
)
t, data = history_case(
    domain,
    model,
    dict(voltage=0.0),
    ("voltage", "current"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["voltage"], 3 * (1 - np.exp(-t)), atol=1e-8)
fig = history_plot("RC voltage and current", model, t, data)
plt.show()
```

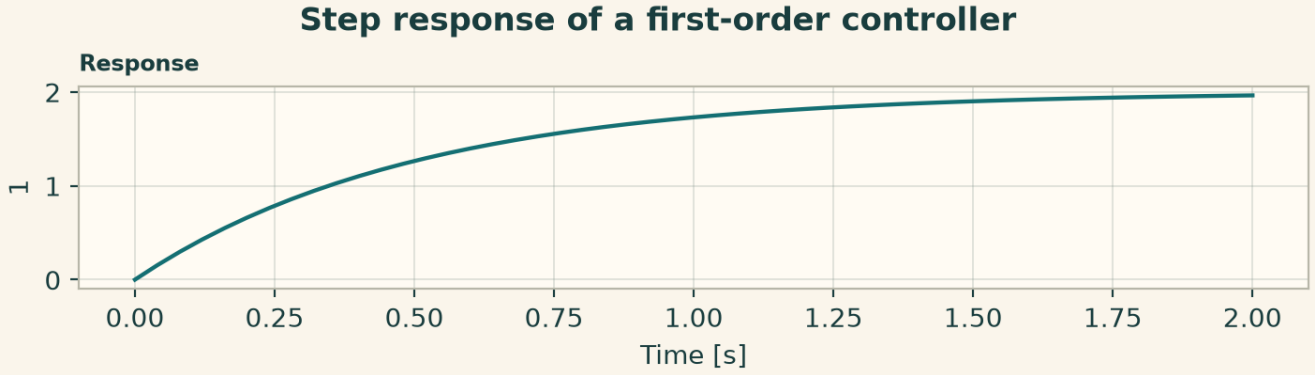


6.19.4 Step response of a first-order controller

Set the signal and output units when using this block with physical inputs. For a unit step the response approaches the configured gain.

```
from onesim.physics.systems import FirstOrderControl

domain = LumpedDomain()
model = FirstOrderControl(
```

```
    domain=domain.system,
    time_constant=0.5,
    gain=2,
    input_signal=1,
)
t, data = history_case(
    domain,
    model,
    dict(response=0.0),
    ("response",),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["response"], 2 * (1 - np.exp(-2 * t)), atol=1e-7)
fig = history_plot("Step response of a first-order controller", model, t, data)
plt.show()
```

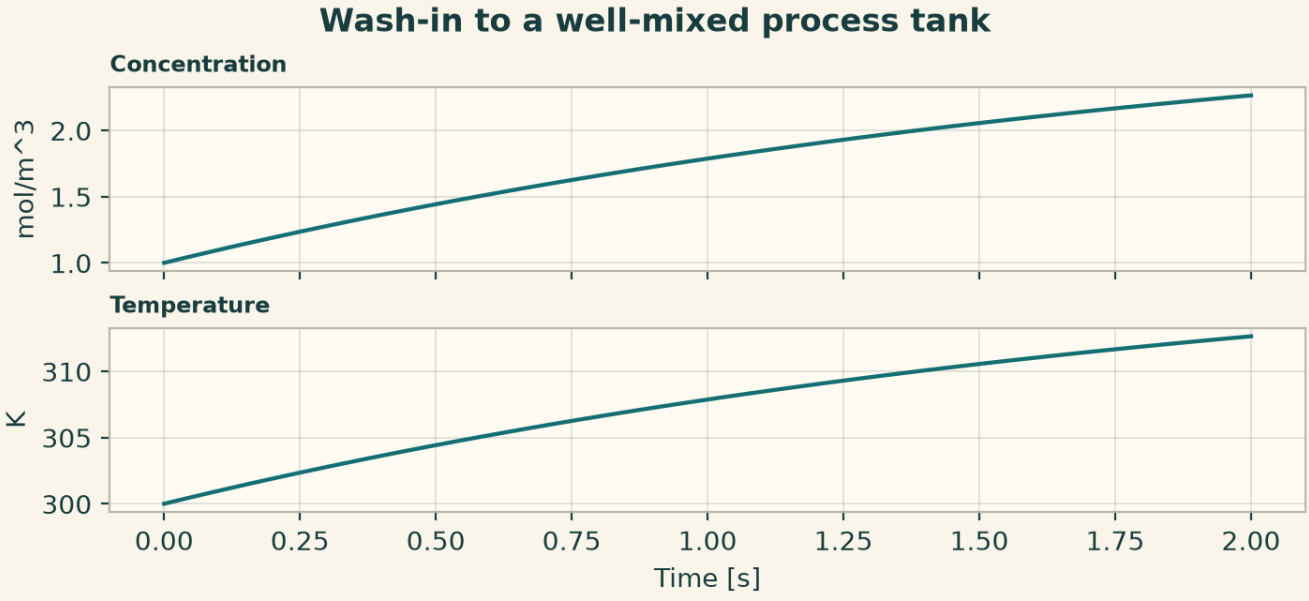


6.19.5 Wash-in to a well-mixed process tank

The volume-to-flow ratio determines the concentration and temperature relaxation time. The feed values and the tank initial state are separate inputs.

```
from onesim.physics.systems import WellMixedTank

domain = LumpedDomain()
model = WellMixedTank(
    domain=domain.system,
    volume=2,
    flow_rate=1,
    feed_concentration=3,
    feed_temperature=320,
)
t, data = history_case(
    domain,
    model,
    dict(concentration=1.0, temperature=300.0),
    ("concentration", "temperature"),
    end=2,
    dt=0.01,
)
```

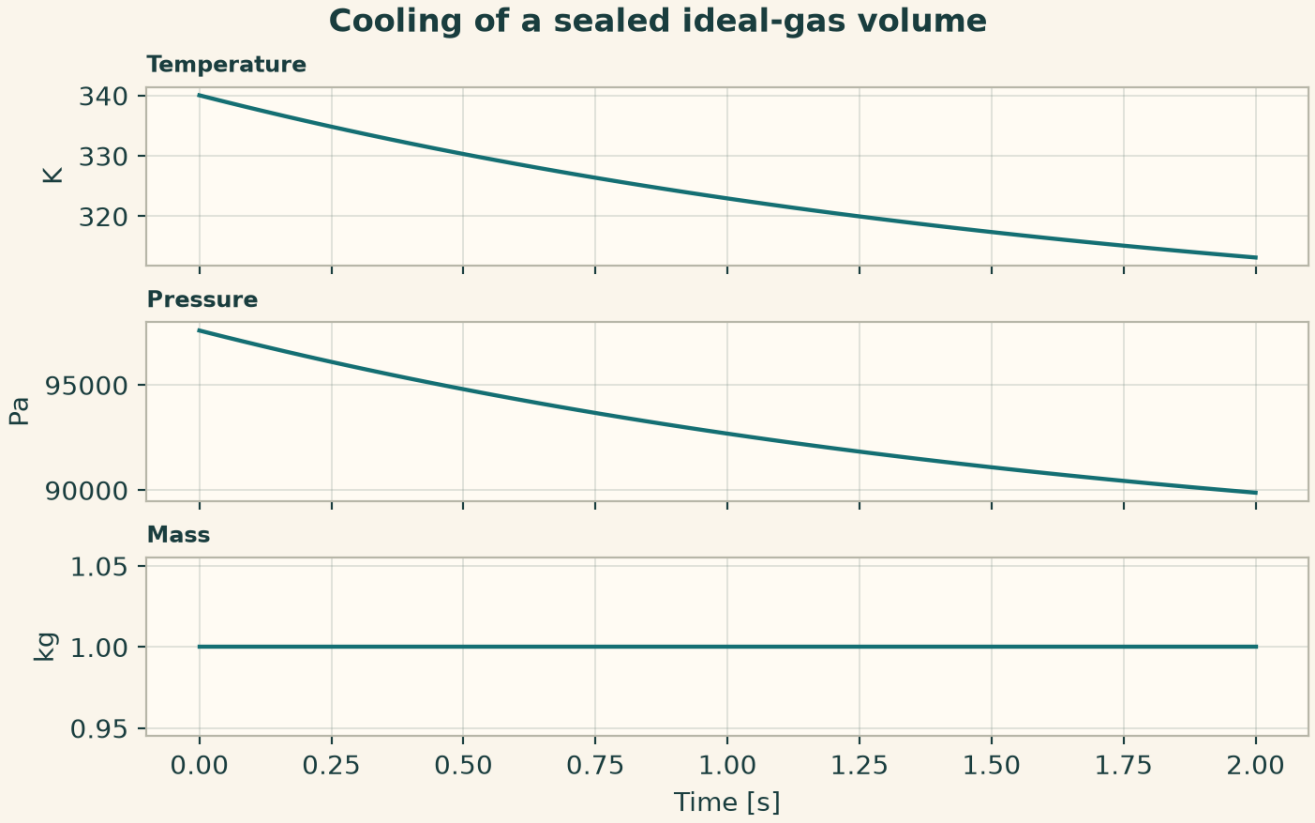
```
np.testing.assert_allclose(data["concentration"], 3 - 2 * np.exp(-t / 2), atol=1e-8)
fig = history_plot("Wash-in to a well-mixed process tank", model, t, data)
plt.show()
```



6.19.6 Cooling of a sealed ideal-gas volume

No mass leaves the chamber. The thermal state relaxes toward ambient, with pressure changing through the ideal-gas closure.

```
from onesim.physics.systems import SealedGasChamber

domain = LumpedDomain()
model = SealedGasChamber(
    domain=domain.system,
    volume=1,
    gas_constant=287,
    heat_capacity_cv=718,
    thermal_conductance=400,
    ambient_temperature=300,
)
t, data = history_case(
    domain,
    model,
    dict(mass=1.0, temperature=340.0),
    ("temperature", "pressure", "mass"),
    end=2,
    dt=0.01,
)
np.testing.assert_allclose(data["pressure"], 287 * data["temperature"], atol=1e-7)
fig = history_plot("Cooling of a sealed ideal-gas volume", model, t, data)
plt.show()
```

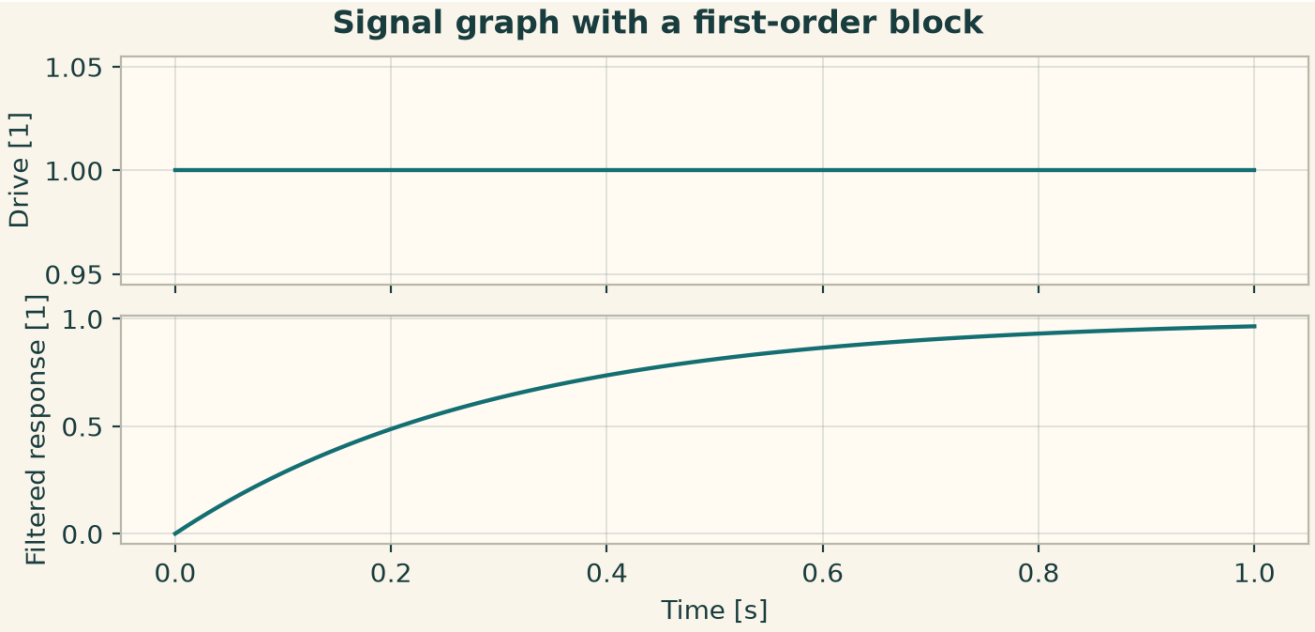


6.19.7 Signal graph with a first-order block

Wire a constant source into a first-order dynamic block. Named signals in the returned history follow the declared graph order.

```
from onesim.physics.systems import (
    ControlSignalFlowConfiguration,
    ControlSignalFlowPublicApi,
    SignalBlock,
)

settings = dict(
    blocks=(
        SignalBlock("drive", "constant", parameters={"value": 1.0}),
        SignalBlock(
            "filter",
            "first_order",
            inputs=("drive",),
            parameters={"time_constant": 0.3, "gain": 0.8},
        ),
    ),
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    ControlSignalFlowPublicApi(), ControlSignalFlowConfiguration(parameters=settings)
)
fig = service_figure(
    "Signal graph with a first-order block",
```

```
    result.times,
    "Time [s]",
    ("Drive [1]", result.signals["drive"]),
    ("Filtered response [1]", result.signals["filter"]),
)
plt.show()
```

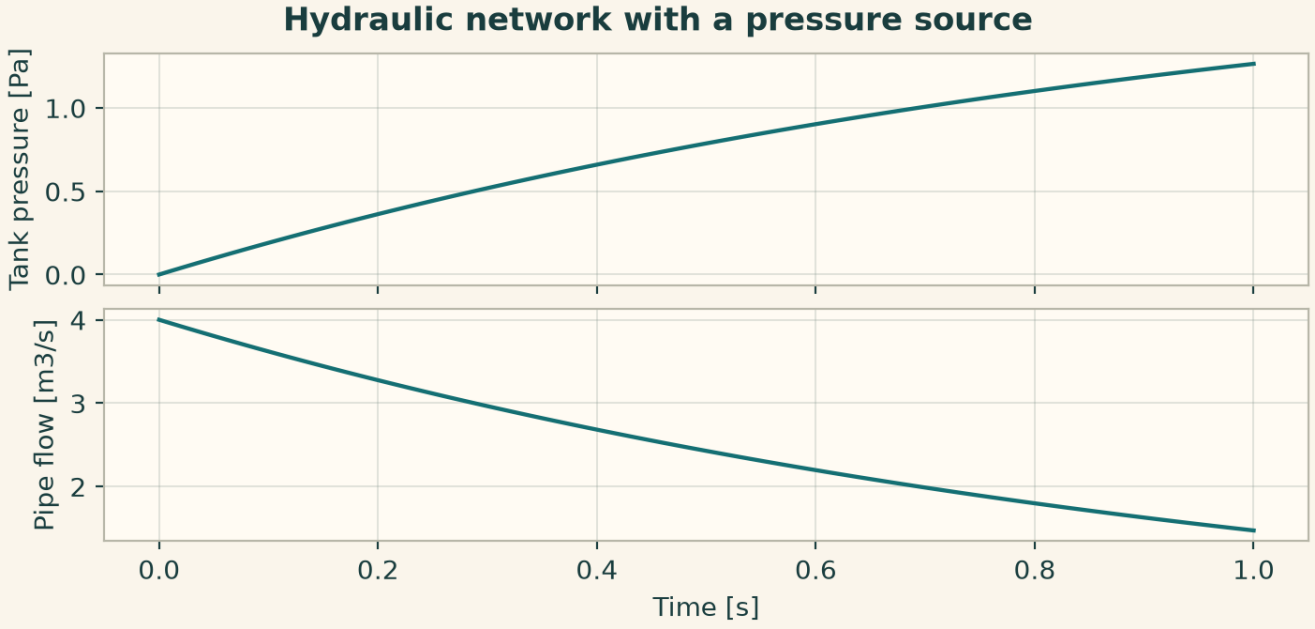


6.19.8 Hydraulic network with a pressure source

Connect an imposed-pressure node to a stored pressure through a resistance. Branch flow is oriented from the first node to the second.

```
from onesim.physics.systems import (
    HydraulicSystemsConfiguration,
    HydraulicSystemsPublicApi,
    HydraulicNode,
    HydraulicBranch,
)

settings = dict(
    nodes=(
        HydraulicNode("supply", imposed_pressure=2),
        HydraulicNode("tank", compliance=2),
    ),
    branches=(HydraulicBranch("pipe", "supply", "tank", resistance=0.5),),
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    HydraulicSystemsPublicApi(), HydraulicSystemsConfiguration(parameters=settings)
)
fig = service_figure(
    "Hydraulic network with a pressure source",
```

```
    result.times,
    "Time [s]",
    ("Tank pressure [Pa]", result.pressures["tank"]),
    ("Pipe flow [m3/s]", result.flows["pipe"]),
)
plt.show()
```

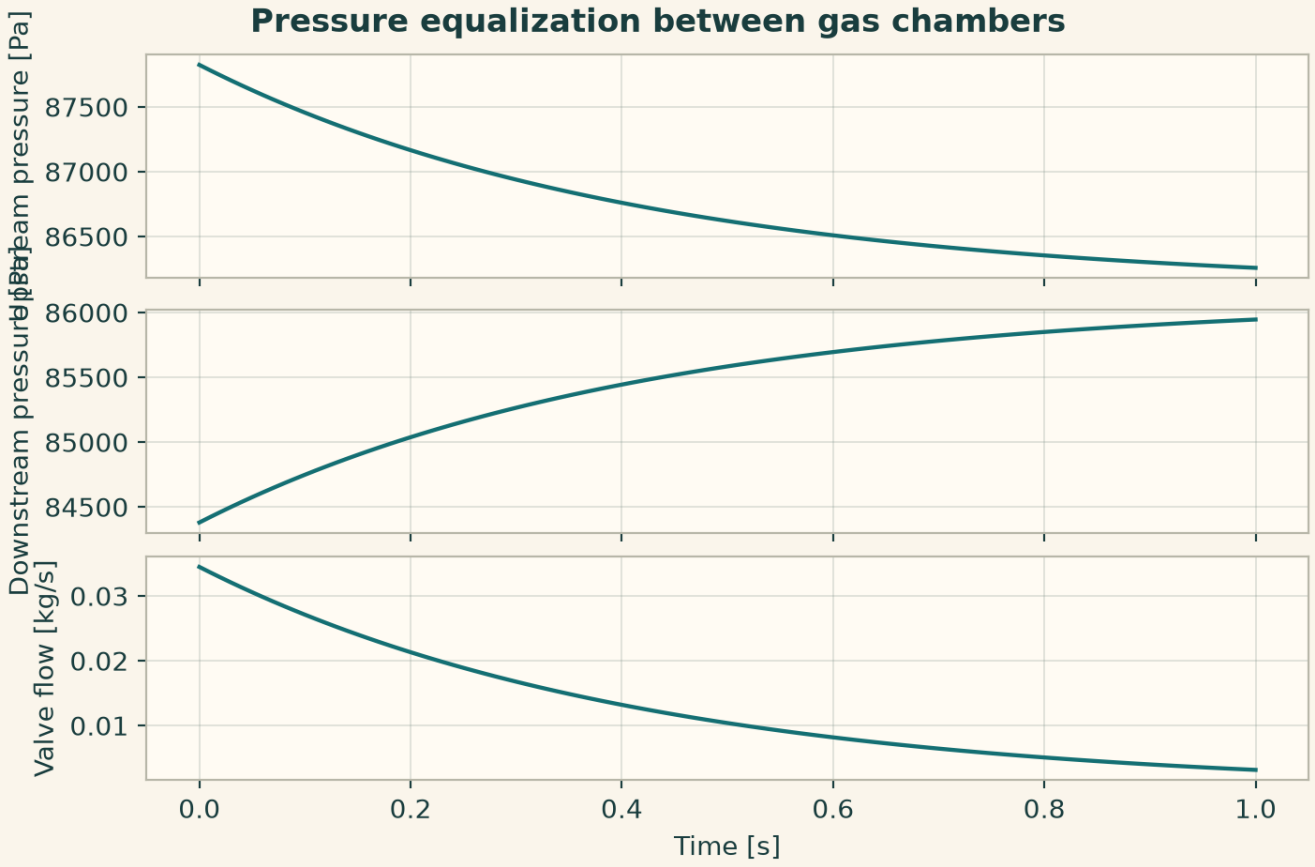


6.19.9 Pressure equalization between gas chambers

Use two gas volumes with distinct initial pressures and connect them with a pressure-driven mass-flow law. The result keeps each chamber and branch separately.

```
from onesim.physics.systems import (
    PneumaticSystemsConfiguration,
    PneumaticSystemsPublicApi,
    PneumaticChamber,
    PneumaticBranch,
)

settings = dict(
    chambers=(
        PneumaticChamber("up", initial_mass=1.02),
        PneumaticChamber("down", initial_mass=0.98),
    ),
    branches=(PneumaticBranch("valve", "up", "down", conductance=1e-5),),
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    PneumaticSystemsPublicApi(), PneumaticSystemsConfiguration(parameters=settings)
)
fig = service_figure(
    "Pressure equalization between gas chambers",
```

```
    result.times,
    "Time [s]",
    ("Upstream pressure [Pa]", result.pressures["up"]),
    ("Downstream pressure [Pa]", result.pressures["down"]),
    ("Valve flow [kg/s]", result.mass_flows["valve"]),
)
plt.show()
```

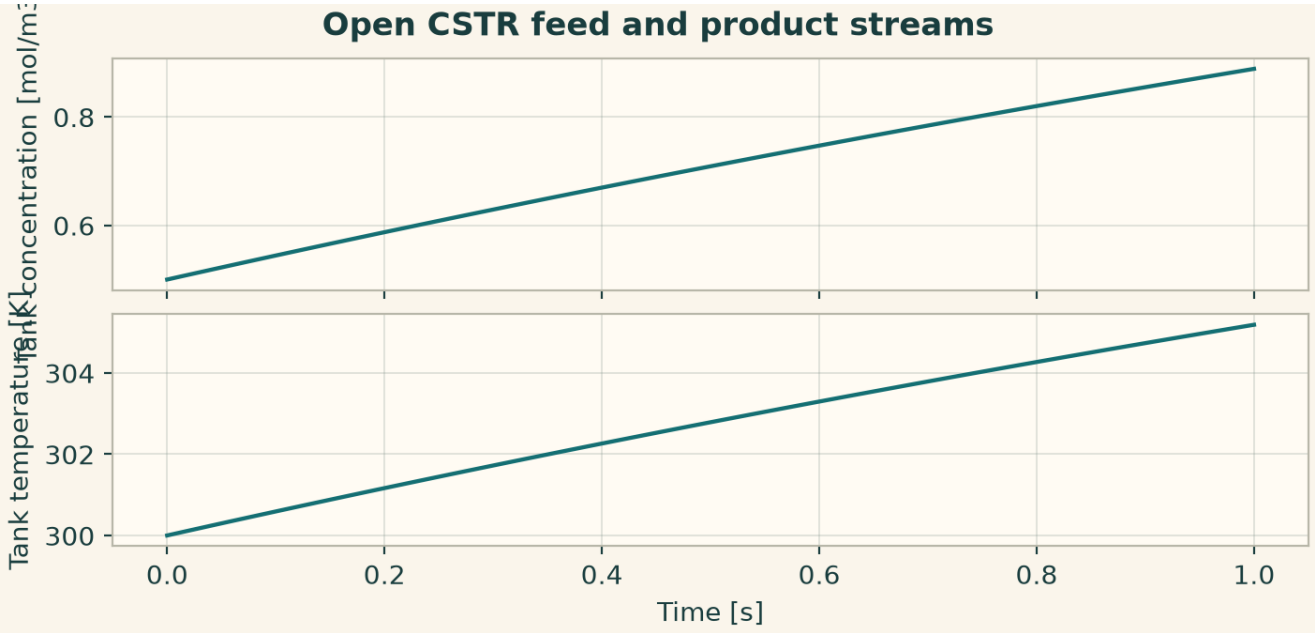


6.19.10 Open CSTR feed and product streams

Identify external feed with a missing source unit and external outlet with a missing target. Keep the two volumetric flows equal for this constant-volume tank.

```
from onesim.physics.systems import (
    ProcessModelsConfiguration,
    ProcessModelsPublicApi,
    ProcessUnit,
    ProcessStream,
)

settings = dict(
    component_ids=("A",),
    units=(ProcessUnit("tank", volume=2.0, initial_concentrations={"A": 0.5}),),
    streams=(
        ProcessStream("feed", None, "tank", 0.6, {"A": 2.0}, 320.0),
        ProcessStream("product", "tank", None, 0.6),
    ),
```

```
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    ProcessModelsPublicApi(), ProcessModelsConfiguration(parameters=settings)
)
fig = service_figure(
    "Open CSTR feed and product streams",
    result.times,
    "Time [s]",
    ("Tank concentration [mol/m3]", result.concentrations["tank"]["A"]),
    ("Tank temperature [K]", result.temperatures["tank"]),
)
plt.show()
```

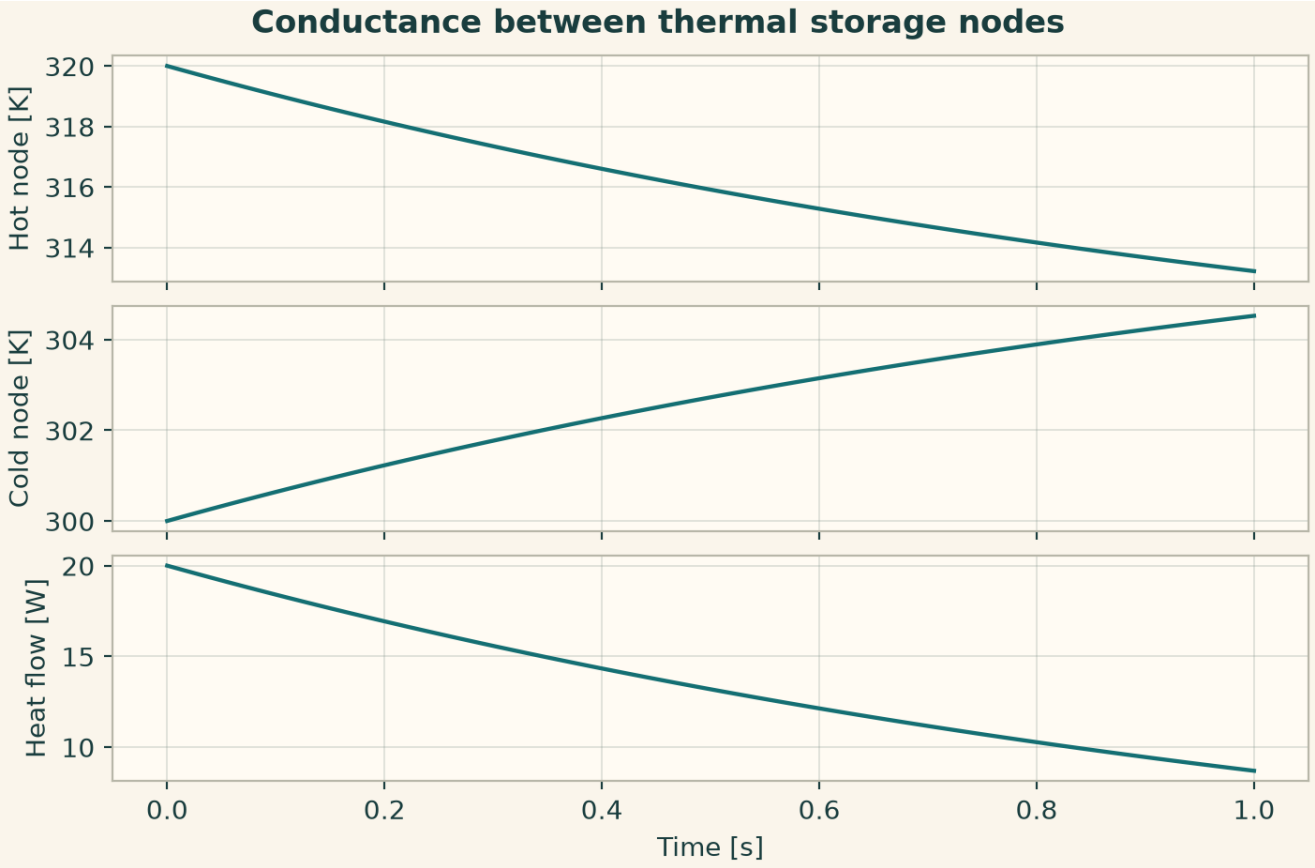


6.19.11 Conductance between thermal storage nodes

Connect two finite capacities. Node temperatures and link heat flow describe the closed redistribution of thermal energy.

```
from onesim.physics.systems import (
    ThermalNetworksConfiguration,
    ThermalNetworksPublicApi,
    ThermalNode,
    ThermalLink,
)

settings = dict(
    nodes=(
        ThermalNode("hot", thermal_capacity=2, initial_temperature=320),
        ThermalNode("cold", thermal_capacity=3, initial_temperature=300),
    ),
    links=(ThermalLink("bridge", "hot", "cold", conductance=1),),
```

```
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    ThermalNetworksPublicApi(), ThermalNetworksConfiguration(parameters=settings)
)
fig = service_figure(
    "Conductance between thermal storage nodes",
    result.times,
    "Time [s]",
    ("Hot node [K]", result.temperatures["hot"]),
    ("Cold node [K]", result.temperatures["cold"]),
    ("Heat flow [W]", result.heat_flows["bridge"]),
)
plt.show()
```

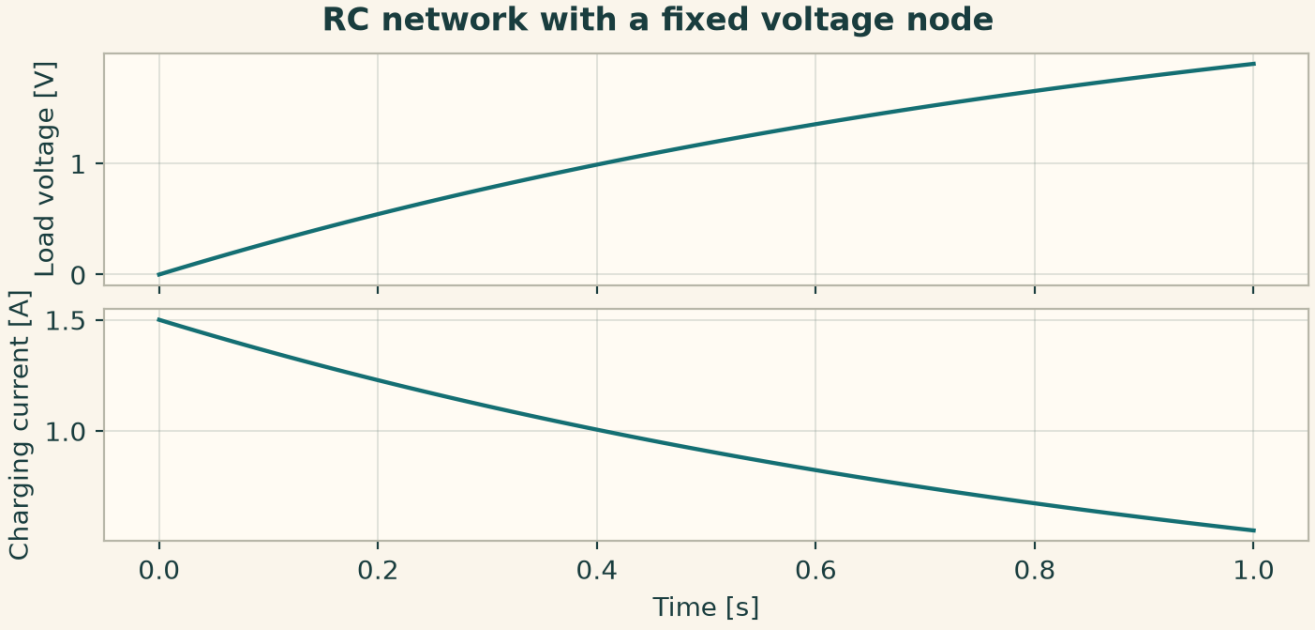


6.19.12 RC network with a fixed voltage node

A resistor links the supply node to a capacitive node. The branch orientation gives a positive charging current.

```
from onesim.physics.systems import (
    ElectricalNetworksConfiguration,
    ElectricalNetworksPublicApi,
    ElectricalNode,
    ElectricalBranch,
)
```

```
settings = dict(
    nodes=(
        ElectricalNode("supply", imposed_voltage=3),
        ElectricalNode("load", capacitance=0.5),
    ),
    branches=(ElectricalBranch("resistor", "supply", "load", resistance=2),),
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    ElectricalNetworksPublicApi(),
    ElectricalNetworksConfiguration(parameters=settings),
)
fig = service_figure(
    "RC network with a fixed voltage node",
    result.times,
    "Time [s]",
    ("Load voltage [V]", result.voltages["load"]),
    ("Charging current [A]", result.currents["resistor"]),
)
plt.show()
```

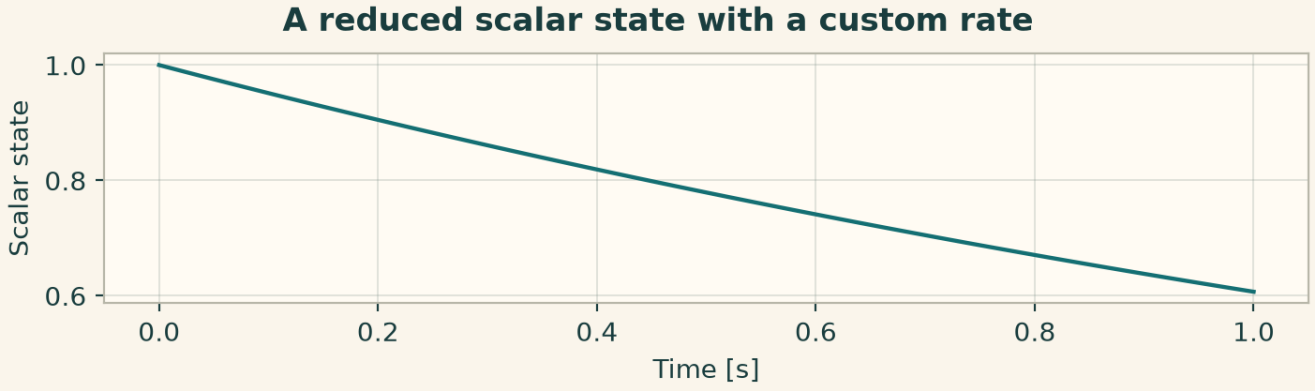


6.19.13 A reduced scalar state with a custom rate

Select a global state and provide its physical rate callback. The service returns the time grid and native state history.

```
from onesim.physics.systems import (
    ZeroDOneDModelsConfiguration,
    ZeroDOneDModelsPublicApi,
)
```

```
settings = dict(
    dimension=0,
    initial_state=1.0,
    rhs=lambda state, coordinates, time: -0.5 * state,
    final_time=1.0,
    time_step=0.01,
)
result = run_service(
    ZeroDOneDModelsPublicApi(), ZeroDOneDModelsConfiguration(parameters=settings)
)
fig = service_figure(
    "A reduced scalar state with a custom rate",
    result.times,
    "Time [s]",
    ("Scalar state", np.asarray(result.history).reshape(-1)),
)
plt.show()
```

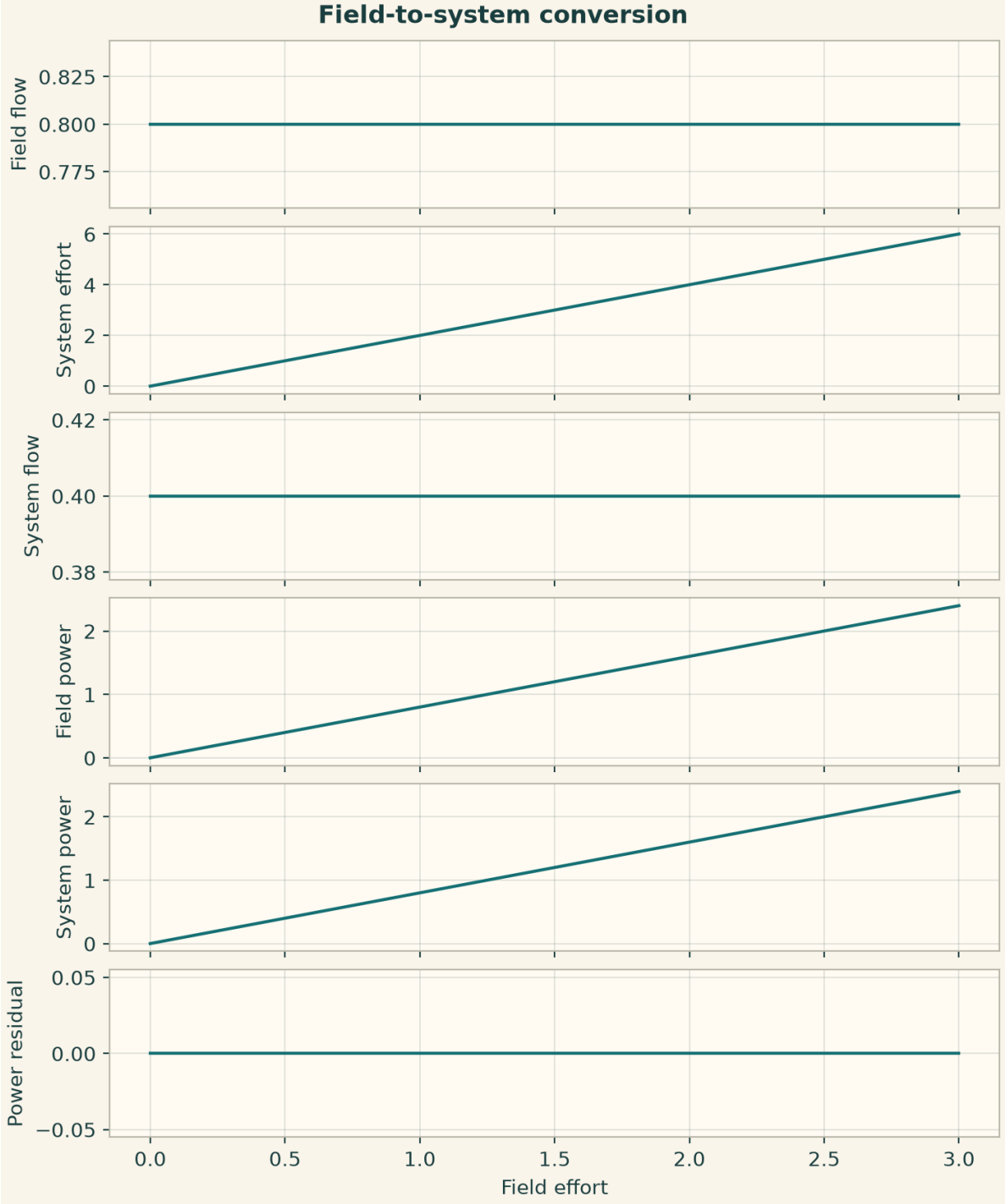


6.19.14 Power-consistent conversion between field effort and system flow

The effort ratio scales the field-to-system effort and the reverse reaction flow. Check the product of effort and flow on both sides. The plot sweeps a family of explicit adapter calls.

```
from onesim.physics.systems import FieldSystemCouplingProvider

provider = FieldSystemCouplingProvider(effort_ratio=2.0)
effort = np.linspace(0, 3, 41)
results = [
    provider.execute(field_effort=float(x), system_flow=0.4).to_mapping()
    for x in effort
]
print(results[-1])
keys = list(results[-1])
fig = service_figure(
    "Field-to-system conversion",
    effort,
    "Field effort",
    *(
        (key.replace("_", " ").capitalize(), [result[key] for result in results])
```

```
        for key in keys
        if key != "field_effort"
    )
)
plt.show()
```



PART III

Select and combine numerical methods

CHAPTER 7

Numerical configuration and spatial operators

7.1 Discretization and validation

`onesim.numerics.discretization`

Pass the selected spatial declaration as `Numerics(spatial=...)`. Attach a `linear_solver` for an assembled solve where required, `nonlinear_solver` for a residual iteration, `linearized_solver` for methods that solve a linearized stage, and `time_integrator` for transient studies. The Newton declaration can own its inner linear solver separately. `step_limits` provides method-independent restrictions. `coupling` holds numerical coupling within the selected binding, such as velocity–pressure correction. The multi-model `Simulation(coupling_solver=...)` selects a coupled study strategy. For heterogeneous models, pass `mesh={model: mesh}` and `numerics={model: numerics}` with all the participating models represented.

`LumpedParameters` represents global unknowns on `LumpedMesh`. Where a spatial declaration accepts `initialization=`, `NodalInterpolation` evaluates the initial data at nodes, `CellAverageProjection` computes physical cell means, and `L2Projection` performs a mass projection with selected quadrature. The representation influences an error comparison. For a finite-volume Fourier mode, compare the cell average of the reference, or compare a reconstructed solution at documented points. For higher-order FEM, compare via native evaluation or an integral norm.

`NumericalIR` states the problem requirements before execution. `NumericalExecutionPlan` records selected method and provider bindings. They are useful when checking how a variant was resolved. `CapabilitySet`, `RequirementSet`, `MethodProviderOffer`, and `evaluate_method_provider` are extension-level compatibility contracts. The evaluation function requires an actual supplied provider and context for confirmation. `count(value, name, minimum=..., maximum=...)` returns a validated built-in integer or raises `ValueError`. It rejects booleans and numeric types that are distinct from Python's `int`.

7.2 Finite differences

`onesim.numerics.fdm`

Use `FiniteDifference` for supported sample or conservative line carriers, and for specific volume bindings with a compatible operator. Set `gradient=` for first derivatives or flux differences and `diffusion=` for second derivatives. `CentralDifference` and `CentralLaplacian` are second-order uniform-grid choices. `PolynomialDifference` and `PolynomialLaplacian` expose accuracy order and nonuniform stencil support where implemented. `SummationByPartsDifference` supplies a derivative with a discrete integration-by-parts structure for supported boundary closures. If the transport law needs a boundary flux, `BoundaryFluxDifference` pairs the derivative with the boundary solver. `WENOFluxDifference` uses `GlobalLaxFriedrichsSplit` and `WENO5Flux` for a supported nonlinear flux route. `HydrostaticFluxDifference` adds the shallow-water equilibrium and admissibility treatment. Verify the stencil's domain, spacing and boundary compatibility before asking for higher order.

7.3 Finite volumes and finite integration

`onesim.numerics.fvm`

Use `FiniteVolume` for conservation laws on control volumes. Choose a face flux, a reconstruction and, when needed, a limiter, a positivity policy and source quadrature. `Rusanov` uses a local wave-speed estimate and is a starting flux for many systems. `MonotoneUpwind` needs a compatible scalar monotone flux. `LinearUpwind` uses gradient information for convection in the volume heat example. `CentralDifference` in this module is the FV diffusion choice in that example. `TwoPointDiffusion` uses a two-point face approximation where the carrier supports it. `LeastSquaresGradient` and `BoundaryLeastSquaresGradient` reconstruct gradients with different boundary information. `ScharfetterGummel` is an exponentially fitted drift–diffusion flux. Use `ImplicitLocalSources` where the physical and time-stage source contracts support it.

`MUSCL` reconstructs face states from a piecewise-linear cell field. Select slope limiting separately. Minmod is restrictive, monotonized central and Van Leer retain more slope in smooth cells, and Superbee is more compressive. Barth–Jespersen constrains reconstructed values by neighbor extrema and Venkatakrishnan regularizes the limiter. `TvbLimiter` wraps a base limiter to preserve smooth curvature under a specified threshold. The ENO and WENO reconstructor classes choose stencils or nonlinear weights for their supported scalar carrier. `ConvexInvariantDomain` retracts traces toward the unchanged mean. `HydrostaticReconstruction`, `WetDryActiveSet` and `WetDryTraceScaling` address balance and admissibility around shallow-water dry fronts. Use a constant solution, a shock or front, and a conservation check before judging the method by a smooth plot.

For collocated flow, `RhieChow` supplies face interpolation and couples pressure to flux. The separate `PISO` or `CoupledVelocityPressure` strategy determines how to solve the pressure–velocity block. `CentralElasticStress` is an FV elasticity contribution. Electromagnetic FV uses `MimeticCurl` or `FiniteIntegration` with an explicit Hodge map. `OrientedCellComplex` handles edge and face incidence; `WhitneyHodge` provides metric maps through Whitney reconstruction and `OrthogonalHodge` targets supported orthogonal carriers. Use complex scalars for phasors. A curl operator on a compatible edge and face complex is distinct from a componentwise gradient.

7.4 Finite elements

`onesim.numerics.fem`

Choose `FiniteElement(element=...)` by continuity and by the quantity the method represents. `Lagrange(order=p)` uses continuous nodal Pp on simplices or Qp on supported tensor-product hexahedra. `DiscontinuousLagrange` gives cell-local degrees of freedom. Supply DG flux and diffusion contributions as required by the equation. `Nedelec` gives tangentially conforming H(curl) fields. `RaviartThomas` and `BrezziDouglasMarini` give normal-conforming H(div) fluxes, paired with a discontinuous scalar in `MixedDiffusionElement`. The lowest public Raviart–Thomas order is zero and the lowest public Nedelec order is one.

Choose `TaylorHood` for a compatible mixed velocity–pressure problem. `EqualOrderVelocityPressure` requires an appropriate pressure stabilization such as `PSPG` in the supported formulation. `GradDiv` penalizes divergence. `MixedMomentumContribution` and `MixedContinuityContribution` allow explicit assembly contributions. For scalar or system transport, `GalerkinAdvection` and `GalerkinDiffusion` select Galerkin contributions. `InterpolatedFluxDivergence` uses an interpolated physical flux. SUPG, GLS, CIP and volume CIP are distinct stabilizations with coefficients and residual contracts. Check the units and scale of

the supplied τ, penalty or diffusivity.

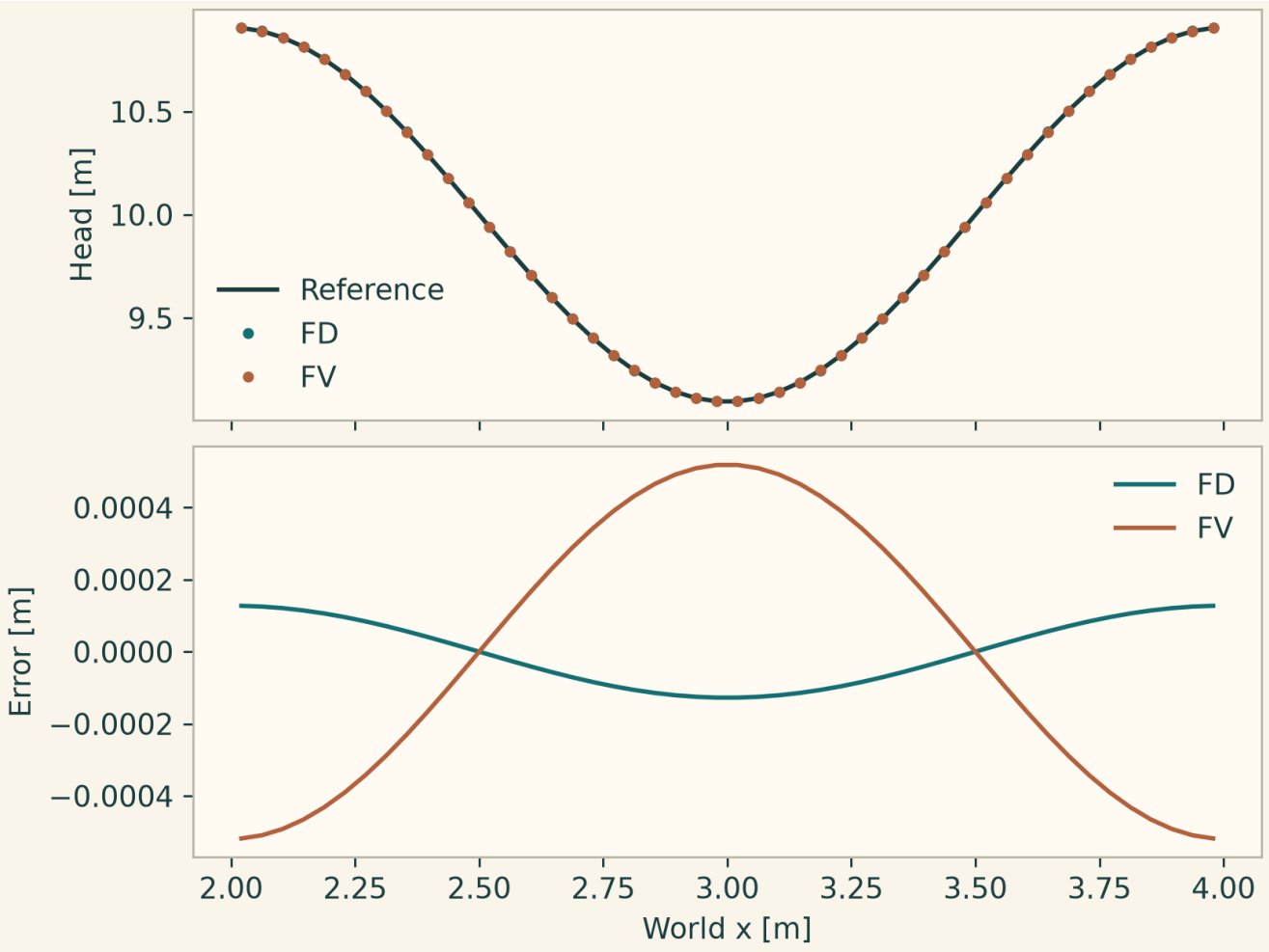
`Nitsche` and `HcurlNitsche` weakly impose the appropriate value or tangential trace. `SymmetricInteriorPenalty` is a DG diffusion contribution with an explicit penalty. `DGJumpPenalty` contributes interface dissipation. `DGPolynomialLimiter`, `ScalarInterval`, `OrderedFields`, `InvariantDomainGraphViscosity` and `HydrostaticGraphViscosity` enforce selected admissibility conditions in supported formulations. `RowSumLumping` is a mass approximation. `ConsistentMassStability` and `DissipativeMassStability` are stability-bound declarations and should be selected for their stated operator properties. `ConstraintAugmentation` augments a primal equation while retaining the constraint. Record which of these components the model actually binds and executes.

7.5 Change one numerical choice at a time

This example advances the same periodic groundwater problem with central FD and two-point FV diffusion. Both runs keep the geometry, source, initial field and time step fixed. The line uses a world coordinate along its axis and the initial condition is periodic over one full wavelength. The checks use native evaluation for visualization and cell centers for the reference. The FV error bound allows for the small difference between a cell mean and a point value on this mesh.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics import Numerics
from onesim.numerics.fdm import CentralDifference, CentralLaplacian, FiniteDifference
from onesim.numerics.fvm import FiniteVolume, TwoPointDiffusion
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.geotechnics import GroundwaterFlow
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

geometry = Line(start=(2, 0, 0), end=(4, 0, 0))
water = GroundwaterFlow(domain=geometry.line, conductivity=0.1, specific_storage=1)
problem = Problem(
    name="Periodic groundwater", geometry=geometry, physics=(water,),
    initial_conditions=(InitialValue(water.head, lambda x: 10 + np.cos(np.pi*x)),),
    boundary_conditions=(
        PeriodicBoundary(
            physics=water,
            pair=PeriodicPair(geometry.boundaries.start, geometry.boundaries.end),
        ),
    ),
)
mesh = StructuredLineMesher(cells=48).generate(geometry)
choices = {
    "FD": FiniteDifference(gradient=CentralDifference(), diffusion=CentralLaplacian()),
    "FV": FiniteVolume(diffusion=TwoPointDiffusion()),
```

```
}
time = 0.1
x = mesh.cell_centers[:, 0]
exact = 10 + np.cos(np.pi*x) * np.exp(-0.1*np.pi**2*time)
theme.use_theme()
fig, (ax, err) = plt.subplots(2, 1, sharex=True, layout="constrained")
ax.plot(x, exact, color=theme.INK, label="Reference")
for name, spatial in choices.items():
    sim = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(spatial=spatial, time_integrator=ClassicalRungeKutta4()),
    )
    sim.validate().require_valid()
    result = sim.run(start_time=0.0, end_time=time, time_step=0.0025)
    values = result.field(water.head).evaluate(mesh.cell_centers)
    assert np.max(np.abs(values - exact)) < 0.003
    ax.plot(x, values, ".", label=name)
    err.plot(x, values - exact, label=name)
ax.set(ylabel="Head [m]")
ax.legend(frameon=False)
err.set(xlabel="World x [m]", ylabel="Error [m]")
err.legend(frameon=False)
plt.show()
```



Use a second mesh and calculate the observed error ratio to assess spatial convergence. Reduce the time step until the ratio is insensitive to time discretization. The same procedure applies to FEM polynomial order, but compare on common physical points or use a consistent quadrature norm. Use `simulation.validate().issues` when binding fails. Check model dimension, carrier geometry, boundary and source support, scalar type, and the capabilities required by each requested component. Change the incompatible choice explicitly.

CHAPTER 8

Solvers, time integration and coupled fields

8.1 Linear solvers and preconditioners

`onesim.numerics.linear`

Start with `SparseDirect()` for a moderate assembled real system and `ComplexSparseDirect()` for a complex one. Direct factorization can have substantial fill. Use `ConjugateGradient` only for an appropriate symmetric positive definite real operator or the supported Hermitian positive definite complex case, with a compatible preconditioner. A gauge, a Lagrange multiplier or a mixed pressure system can change definiteness. Use `GMRES` for a compatible nonsymmetric system, `RightGMRES` for fixed right preconditioning, and `FlexibleGMRES` when the right preconditioner varies during iteration. Set the solver's scalar type explicitly for phasors. Compare the physical residual and at least one field observable against a reference direct solve on a small mesh.

`ILU` is a sparse incomplete factorization with fill and drop controls. `AlgebraicMultigrid` accepts near-nullspace information where supported. `BlockPreconditioner` requires solvers and an owned degree-of-freedom partition for block sweeps. `SchurPreconditioner` uses a leading-block solve and a Schur solve or approximation. Use field-aware block order, including constraints. A smaller relative tolerance can expose an incorrect scaling or preconditioner, so also inspect absolute tolerance and iteration count.

```
from onesim.numerics.linear import GMRES, ILU

linear = GMRES(
    preconditioner=ILU(fill_factor=2.0, drop_tolerance=1e-5),
    relative_tolerance=1e-9, absolute_tolerance=1e-12,
    maximum_iterations=500, restart=40,
)
# Pass linear as Numerics(linear_solver=linear, spatial=...),
# or to a nonlinear declaration that owns its inner solve.
```

8.2 Nonlinear solvers

`onesim.numerics.nonlinear`

`Newton` uses a residual and the chosen Jacobian representation. `DampedNewton` adds a line search, `TrustRegionNewton` restricts the correction in solver coordinates, and `QuasiNewton` updates an approximate inverse after an initial solve. `JFNK`, also exported as `JacobianFreeNewtonKrylov`, needs an appropriate Krylov inner solve. Its Jacobian option selects the derivative route permitted by the binding, which can include a physical analytic action. A matrix-free operator and a preconditioner that requires an assembled matrix must be connected through a supported preconditioning representation. `FixedPoint` requires a supported fixed-point map and may need relaxation. Compare both residual reduction and changes in physical outputs. Field-specific `Numerics(field_scales=...)` and a physically meaningful `initial_guess=` can improve the

interpretation of convergence where the binding supports them.

8.3 Time integration, events and continuation

`onesim.numerics.time`

Use `ForwardEuler` for a simple first-order reference and `ClassicalRungeKutta4` for a higher-order explicit reference with four stages. For stiff dynamics, `BackwardEuler` supplies a first-order implicit step. `BDF(order=2, startup=BackwardEuler())` adds multistep history. The generic BDF declaration accepts orders one through six, subject to the bound study and startup policy. Document the startup method and check the maximum step ratio for variable-step history. For conservation-law and diffusion problems use a step small enough for the advective, acoustic or diffusion restrictions on the chosen mesh.

`TimeMethod(method_id)` selects a catalog ID in compatible studies. The following table gives the generic object-state catalog in the inspected source. Array, DAE and partitioned execution can impose different restrictions. Attach the required stage solver only when consumed by the selected method.

Stage solver	Method IDs	Selection notes
Explicit	`forward_euler`, `ssprk2`, `classical_rk4`, `embedded_heun_euler`	Single-step methods. Nominal orders are 1, 2, 4, and 2.
Explicit	`adams_bashforth2`, `predictor_corrector_abam2`, `leapfrog`	Second-order methods with history. Check stability and startup.
Nonlinear	`backward_euler`, `theta_075`, `crank_nicolson`, `implicit_rk_midpoint`, `dirk_sdirk2`, `adams_moulton_2`	Single-step implicit methods. Backward Euler and the fixed $\theta = 0.75$ method are first order. The others are second order.
Nonlinear	`adams_moulton_3`, `adams_moulton_4`, `bdf2_gear`, `bdf3_gear`, `bdf4_gear`, `bdf5_gear`, `bdf6_gear`	Implicit multistep methods. The number gives the nominal order.
Linearized	`rosenbrock_euler`, `rosenbrock_wanner_euler`	First-order linearly implicit methods in this catalog.

The following is a method substitution for a compatible lumped implicit study. Keep its physical declarations and mesh fixed and pass `implicit` as the new simulation's `numerics`. Compare with the explicit result at common output times.

```
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BDF, BackwardEuler, TimeStepLimits

implicit = Numerics(
    spatial=LumpedParameters(),
```

```
    time_integrator=BDF(order=2, startup=BackwardEuler()),
    nonlinear_solver=Newton(linear_solver=SparseDirect(), jacobian="analytic"),
    step_limits=TimeStepLimits(maximum_step=0.02),
)
```

Apply the declaration to the resistor–capacitor problem and compare with its closed form. Use the actual accepted time for the reference and re-run with a smaller step to assess temporal error.

```
import numpy as np
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.physics.systems import ResistorCapacitor
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
rc = ResistorCapacitor(
    domain=domain.system, resistance=2.0, capacitance=0.5, supply_voltage=3.0,
)
problem = Problem(
    name="Implicit RC step", geometry=domain, physics=(rc,),
    initial_conditions=(InitialValue(rc.voltage, 0.0),),
)
sim = Simulation(problem=problem, mesh=LumpedMesh(domain), numerics=implicit)
sim.validate().require_valid()
result = sim.run(start_time=0.0, end_time=1.0, time_step=0.01)
t_final = float(result.times[-1])
value = np.asarray(result.field(rc.voltage).values).item()
assert np.isclose(t_final, 1.0)
assert abs(value - 3.0*(1 - np.exp(-t_final))) < 5e-4
```

Pass `AdaptiveTimeStep(initial=..., minimum=..., maximum=..., ...)` as `Simulation.run(time_step=...)` when the method supports the adaptive controller. This sets the error tolerances and accepted-step range. `TimeStepLimits` independently sets an absolute maximum, a stability safety factor and a CFL limit. Monitor accepted times and rejections and tighten tolerances for a comparison. Align `Output` scheduling with the retained samples you intend to plot. `TimeEventDefinition` defines a scalar event on semantic states with crossing direction, terminal behavior and an optional reset. Use the typed `FieldThreshold` route for a supported field event. See `test_declared_field_events.py` and `test_array_adaptive_time.py`. Check a known event time on a small case and retain the checkpoint required by the selected continuation method.

8.4 Transfer between fields

`onesim.numerics.field_transfer`

A `FieldCoupling(source=..., target=..., transfer=...)` connects typed physical values and their numerical representations. Start by checking component count, units, ownership, and geometric overlap. Choose `GlobalValueTransfer` between global fields, `GlobalBroadcastTransfer` to

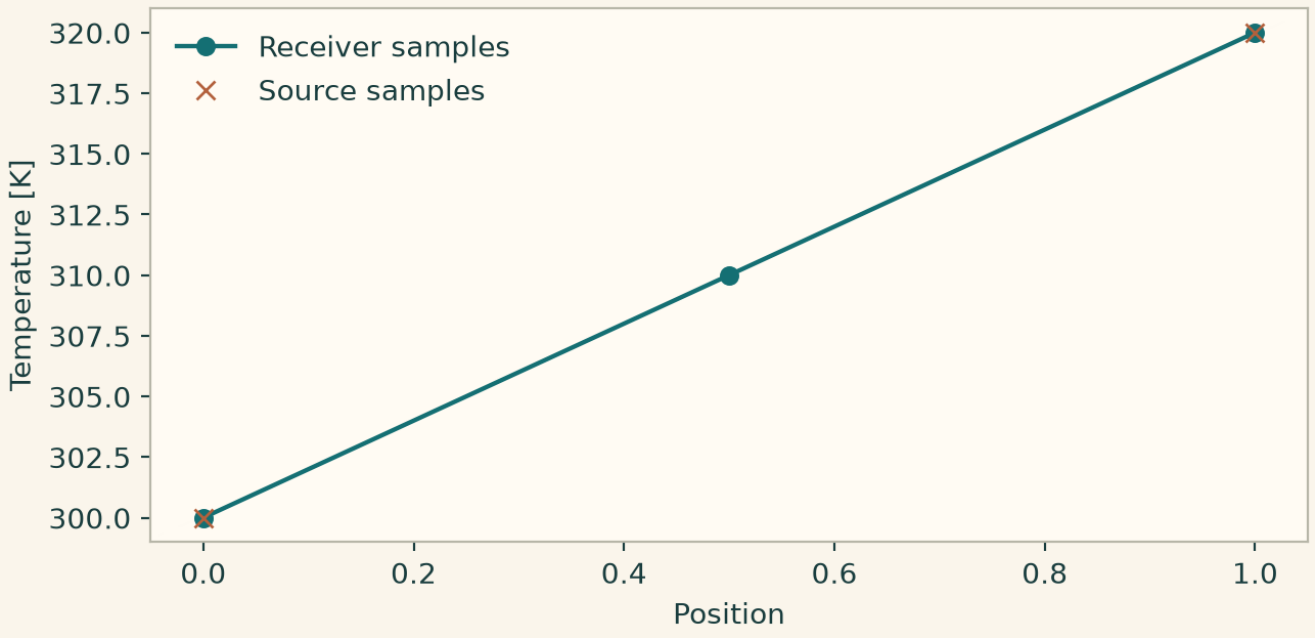
drive a spatial field uniformly, and `SpatialAverageTransfer` to reduce a spatial field to its physically weighted average. `FiniteElementEvaluation` evaluates the source at points requested by the target, often quadrature points. `CellToQuadratureProjection` samples native cell data on the receiver quadrature. `FieldInterpolation` represents point interpolation on compatible supports. `ElectricFieldToFaces` serves electric transport at faces, with `PeriodicPotentialTrace` declaring the signed potential jump when needed. `ConservativeVolumeTransfer` integrates native scalar FE density over physical FV overlap volumes. Check the integrated total and domain coverage. Interpolation and conservation are distinct requirements.

For a two-way coupling, first run both sides separately, check each transfer on a constant field, then compare the driven result with a known spatial or global response. Keep source and receiver mesh IDs with exported data when meshes differ. The exposed `operators` namespace also provides explicit request, validation and application functions for provider-level transfers.

8.4.1 Validate and apply an explicit interpolation

Use this low-level route when you already own the source and target coordinates and the samples. For example, resample a one-dimensional temperature profile onto an output grid. A one-to-one field coupling inside a simulation can use a declaration such as `FiniteElementEvaluation` instead.

```
import numpy as np
from onesim.numerics.field_transfer import operators

mapping = operators.interpolation(
    "temperature profile", "coarse", "display",
    source_coordinates=(0.0, 1.0), target_coordinates=(0.0, 0.5, 1.0),
    source_unit="K", target_unit="K",
)
values = np.array([300.0, 320.0])
report = operators.validate(mapping, source=values)
assert report.accepted, report.diagnostics
mapped = operators.apply(mapping, values)
np.testing.assert_allclose(mapped.value, [300.0, 310.0, 320.0])
print(mapped.report.mapping_kind)

import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.6,3.2),layout="constrained")
ax.plot((0,.5,1),mapped.value,"o-",label="Receiver samples")
ax.plot((0,1),values,"x",markersize=7,label="Source samples")
ax.set(xlabel="Position",ylabel="Temperature [K]")
ax.legend(frameon=False)
plt.show()
```



The returned value is an array on the target layout. The report records the provider, support, unit scaling and conservation evidence. For piecewise-constant cell values use the corresponding `conservative_remap` declaration with cell edges. Check the weighted total on both layouts. The tetrahedral versions take actual meshes, with coverage and conservation controls described in Appendix C.

8.5 Coupled systems

`onesim.numerics.coupled_systems`

Use `SequentialCoupling` for an explicitly ordered sweep with unique stationary and transient roles. Use `PartitionedCoupling` for peer blocks that iterate while replaying the same trial time step. Specify an iteration declaration and check convergence of exchanged observables. `QuasiStationaryCoupling` iterates transient states against stationary blocks at a shared trial time and rejects a failed trial step. `MonolithicCoupling` advances transient blocks with one time algorithm, with compatible stage solvers supplied to the coupled strategy. `AlgebraicClosure` handles instantaneous feedback in the relevant stage and its initial values are iteration guesses in the source field's units.

`PISO` selects momentum and pressure solvers, pressure correctors, tolerances, and relaxation in a supported FV flow. `CoupledVelocityPressure` selects a joint velocity, face-flux and pressure solve. Check discrete continuity and pressure reference handling. For multiphysics, compare the coupled result with a one-way solution or a small independent reference. Report the transferred observable and the residual or iteration change on each side. `test_public_quasistationary.py`, `test_native_transient_partitioned_coupling.py`, and `test_coupled_fv_pressure.py` show specific execution paths.

CHAPTER 9

Worked numerical comparisons

Keep the physical declaration fixed when changing a method. The comparisons below expose the particular numerical observable being varied. Run the one-dimensional helper from Chapter 6 before the line experiments.

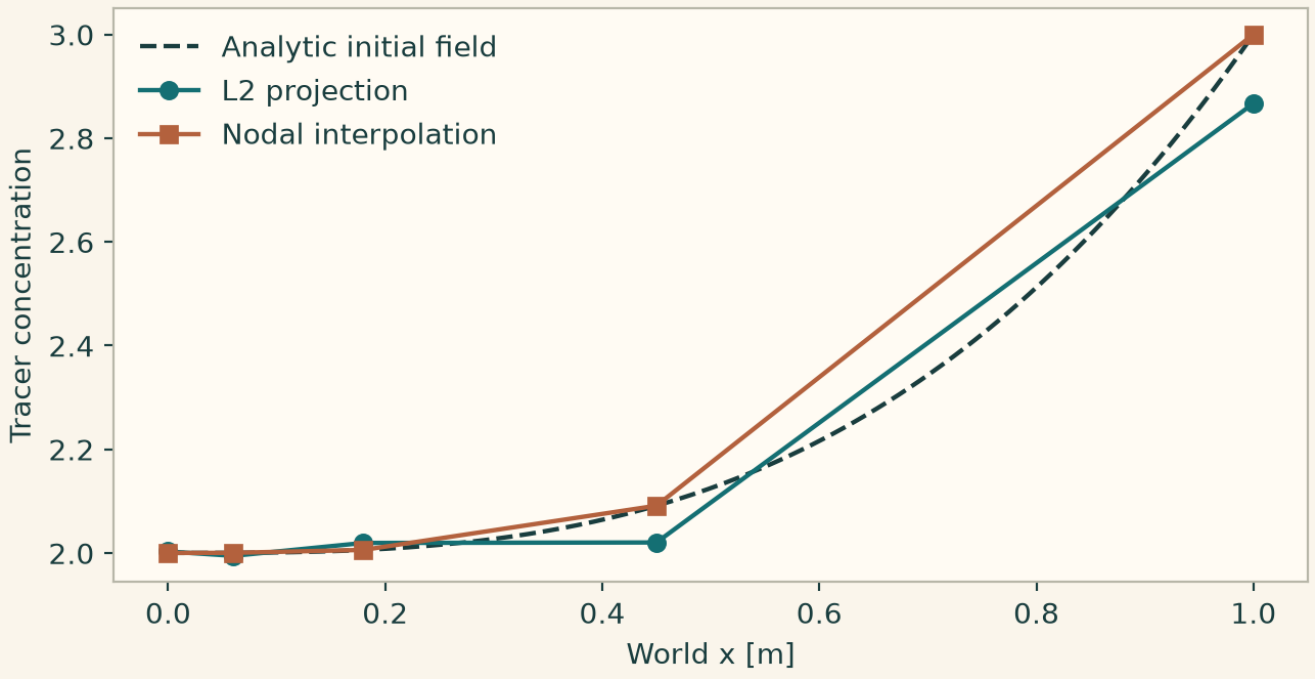
9.1 Discretization and initialization lab

9.1.1 L2 projection versus nodal interpolation

Project a cubic initial field onto a continuous piecewise-linear space on a nonuniform line. Nodal interpolation matches point values at nodes, while L2 projection preserves the integral of the analytic cubic when the constant test function is present. This listing uses the concrete line FE space as a provider-level diagnostic.

```
from onesim.meshing import NonuniformLineMesher
from onesim.numerics.discretization import L2Projection, NodalInterpolation
from onesim.numerics.fem.backends.spaces.line_lagrange import LineLagrangeSpace
from onesim.physics.chemistry import SpeciesTransport, TransportSpecies

line = Line(start=(0, 0, 0), end=(1, 0, 0))
mesh = NonuniformLineMesher(faces=(0.0, 0.06, 0.18, 0.45, 1.0)).generate(line)
space = LineLagrangeSpace(mesh, order=1)
model = SpeciesTransport(
    domain=line.line,
    velocity=0.0,
    species=(TransportSpecies("tracer", diffusivity=0.1),),
)
initial = InitialValue(model.concentration("tracer"), lambda x: 2 + x**3)
l2 = L2Projection(quadrature_order=4).project(
    initial=initial,
    space=space,
    time=0.0,
    linear_solver=SparseDirect().bind(),
)
nodal = NodalInterpolation().project(initial=initial, space=space, time=0.0)
integral = 0.0
for cell, dofs in enumerate(space.cells):
    _, weights, (basis, _, _) = space.element_data(cell)
    integral += weights @ (basis @ l2[dofs])
np.testing.assert_allclose(integral, 2.25, atol=1e-13)
x = space.dof_coordinates[:, 0]
dense = np.linspace(0, 1, 200)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.6, 3.4), layout="constrained")
ax.plot(dense, 2 + dense**3, "--", color=theme.INK, label="Analytic initial field")
ax.plot(x, l2, "o-", label="L2 projection")
ax.plot(x, nodal, "s-", label="Nodal interpolation")
```

```
ax.set(xlabel="World x [m]", ylabel="Tracer concentration")
ax.legend(frameon=False)
plt.show()
```



9.1.2 Inspect and check a method offer

Validate integer controls at a notebook boundary. Read the numerical IR from the bound model instance. The separate offer check shows the difference between a method identifier and a provider confirming a context-specific capability.

```
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import (
    CapabilitySet,
    LumpedParameters,
    MethodProviderOffer,
    RequirementSet,
    count,
    evaluate_method_provider,
)
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.systems import ResistorCapacitor
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

steps = count(20, "requested steps", minimum=2)
domain = LumpedDomain()
model = ResistorCapacitor(
    domain=domain.system, resistance=2, capacitance=0.5, supply_voltage=3
)
problem = Problem(
```

```
    name="Inspect a plan",
    geometry=domain,
    physics=(model,),
    initial_conditions=(InitialValue(model.voltage, 0.0),),
)
sim = Simulation(
    problem=problem,
    mesh=LumpedMesh(domain),
    numerics=Numerics(
        spatial=LumpedParameters(), time_integrator=ClassicalRungeKutta4()
    ),
)
sim.validate().require_valid()
ir = sim.compile().numerical_ir(model)
print(ir.to_dict())
sim.run(start_time=0, end_time=1, time_step=1 / steps)

method = "spatial-method:finite.elemente.methoden.fem"
features = frozenset({method, "spatial-method", "dimension:2"})
offer = MethodProviderOffer("example.p1", CapabilitySet(features), available=True)
accepted = evaluate_method_provider(
    method, RequirementSet(frozenset({"dimension:2"})), provider=offer
)
rejected = evaluate_method_provider(
    method, RequirementSet(frozenset({"dimension:3"})), provider=offer
)
assert accepted.accepted and not rejected.accepted
print(rejected.diagnostics)
```

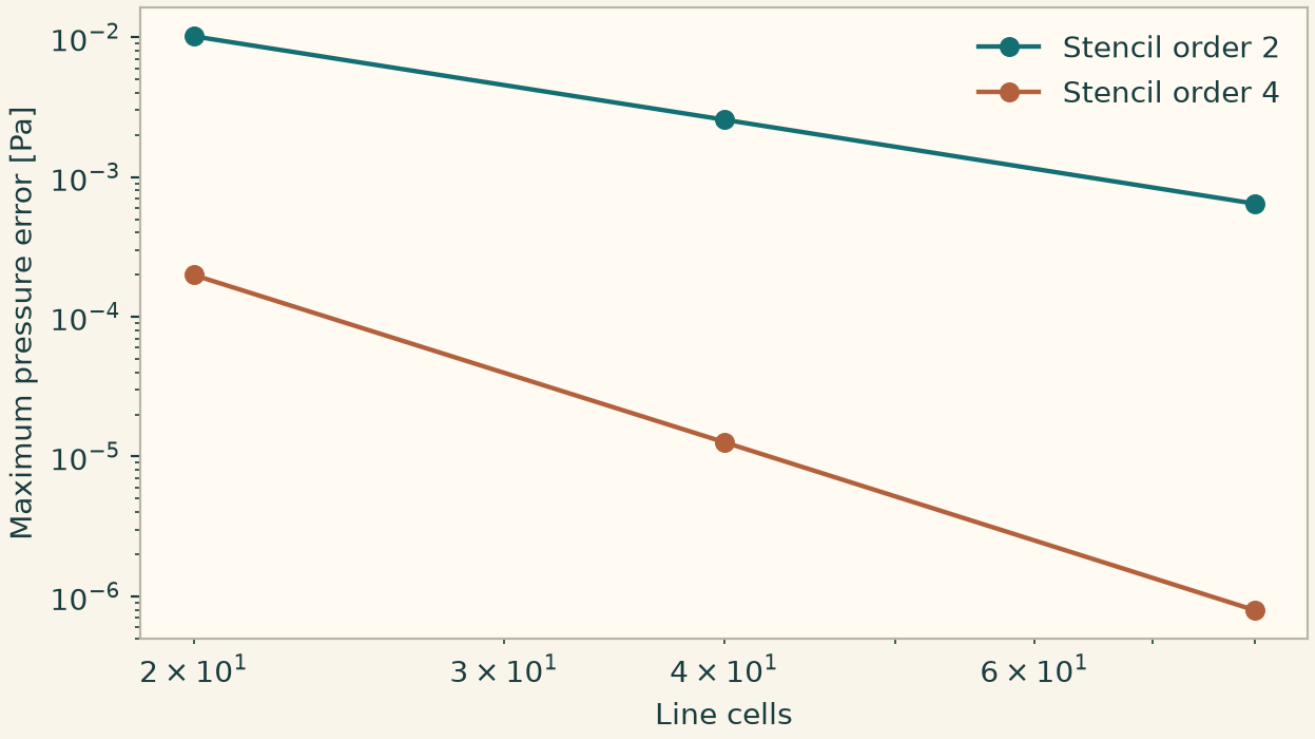
9.2 Finite-difference lab

9.2.1 Measure finite-difference order on a traveling wave

Reuse the linear acoustic characteristic and hold the time error small. Compare the maximum pressure error across meshes when changing the first-derivative declaration.

```
from onesim.numerics.fdm import PolynomialDifference
from onesim.physics.acoustics import LinearAcoustics

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = LinearAcoustics(domain=line.line, density=1, sound_speed=1)
wave = lambda x: np.sin(2 * np.pi * x)
counts = (20, 40, 80)
errors = {}
for order in (2, 4):
    errors[order] = []
    for n in counts:
        result = line_experiment(
            line,
            model,
            dict(pressure=wave, velocity=wave),
```

```
            FiniteDifference(gradient=PolynomialDifference(accuracy_order=order)),
            cells=n,
            end=0.1,
            dt=0.0005,
        )
        p = result.field(model.pressure)
        errors[order].append(
            np.max(np.abs(p.values - wave(p.space.dof_coordinates[:, 0] - 0.1)))
        )
assert errors[4][-1] < errors[2][-1] and errors[2][-1] < errors[2][0]
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.4, 3.6), layout="constrained")
for order, y in errors.items():
    ax.loglog(counts, y, "o-", label=f"Stencil order {order}")
ax.set(xlabel="Line cells", ylabel="Maximum pressure error [Pa]")
ax.legend(frameon=False)
plt.show()
```



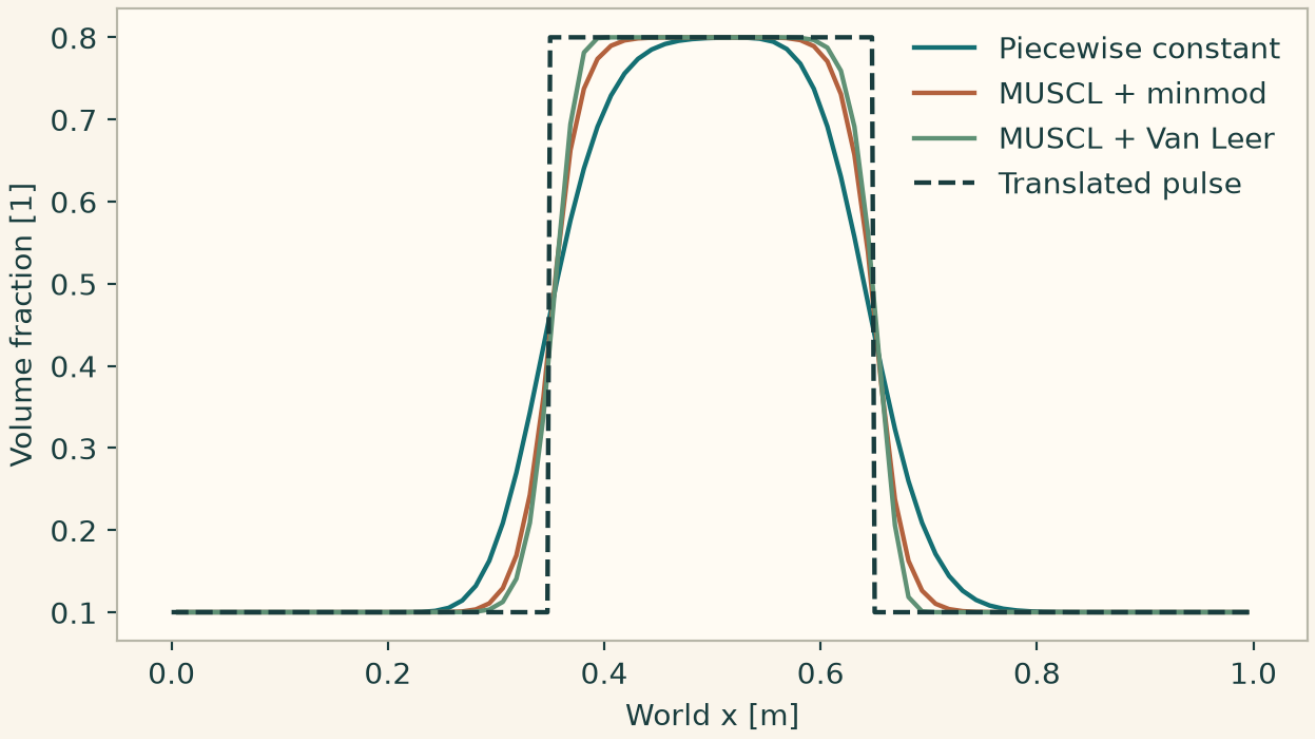
9.3 Finite-volume lab

9.3.1 Conservative reconstruction and slope limiting

Use an advected pulse with three FV choices on the same mesh. A limiter controls reconstructed traces and conservation is checked independently by integrating the cell field.

```
from onesim.numerics.fvm import MinmodLimiter, VanLeerLimiter
from onesim.physics.hydrodynamics import PhaseAdvection

line = Line(start=(0, 0, 0), end=(1, 0, 0))
model = PhaseAdvection(
```

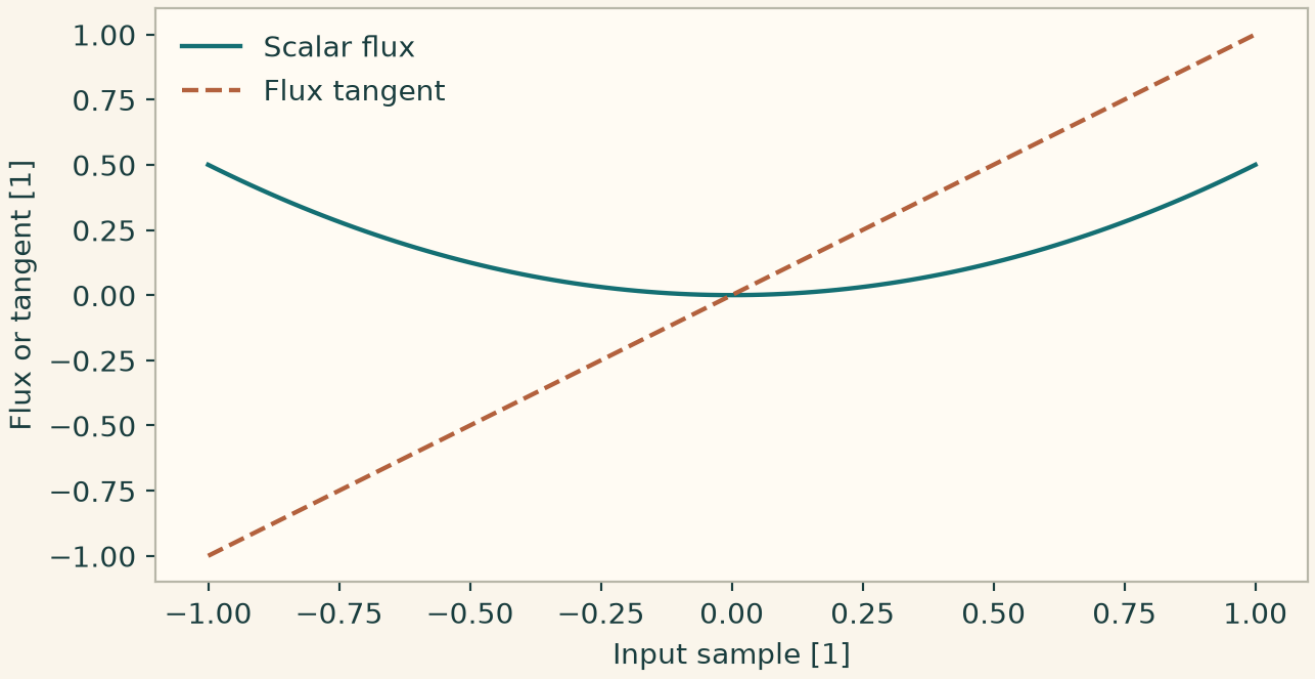
```
    domain=line.line, velocity=0.6, first_density=1000, second_density=1
)


def pulse(x):
    return np.where((x >= 0.2) & (x < 0.5), 0.8, 0.1)


choices = {
    "Piecewise constant": FiniteVolume(convection=Rusanov()),
    "MUSCL + minmod": FiniteVolume(
        convection=Rusanov(), reconstruction=MUSCL(), limiter=MinmodLimiter()
    ),
    "MUSCL + Van Leer": FiniteVolume(
        convection=Rusanov(), reconstruction=MUSCL(), limiter=VanLeerLimiter()
    ),
}
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.4, 3.6), layout="constrained")
for name, spatial in choices.items():
    result = line_experiment(
        line,
        model,
        dict(volume_fraction=pulse),
        spatial,
        cells=80,
        end=0.25,
        dt=0.001,
    )
    fraction = result.field(model.volume_fraction)
    np.testing.assert_allclose(fraction.integrate().value, 0.31, atol=1e-10)
    ax.plot(fraction.space.dof_coordinates[:, 0], fraction.values, label=name)
x = np.linspace(0, 1, 401)
ax.plot(x, pulse((x - 0.15) % 1), "--", color=theme.INK, label="Translated pulse")
ax.set(xlabel="World x [m]", ylabel="Volume fraction [1]")
ax.legend(frameon=False)
plt.show()
```



9.3.2 Differentiate a sampled nonlinear flux

For a scalar flux with independent input samples, initialize the identity tangent and propagate the derivative through a unary law. The derivative of each flux value is on the diagonal of the sparse Jacobian.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.numerics.fvm import DifferentiableSamples

x = np.linspace(-1, 1, 101)
samples = DifferentiableSamples.variables(x)
flux = samples.unary(0.5 * x * x, x)
derivative = flux.jacobian.diagonal()
np.testing.assert_allclose(flux.values, 0.5 * x * x)
np.testing.assert_allclose(derivative, x)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.6, 3.4), layout="constrained")
ax.plot(x, flux.values, label="Scalar flux")
ax.plot(x, derivative, "--", label="Flux tangent")
ax.set(xlabel="Input sample [1]", ylabel="Flux or tangent [1]")
ax.legend(frameon=False)
plt.show()
```



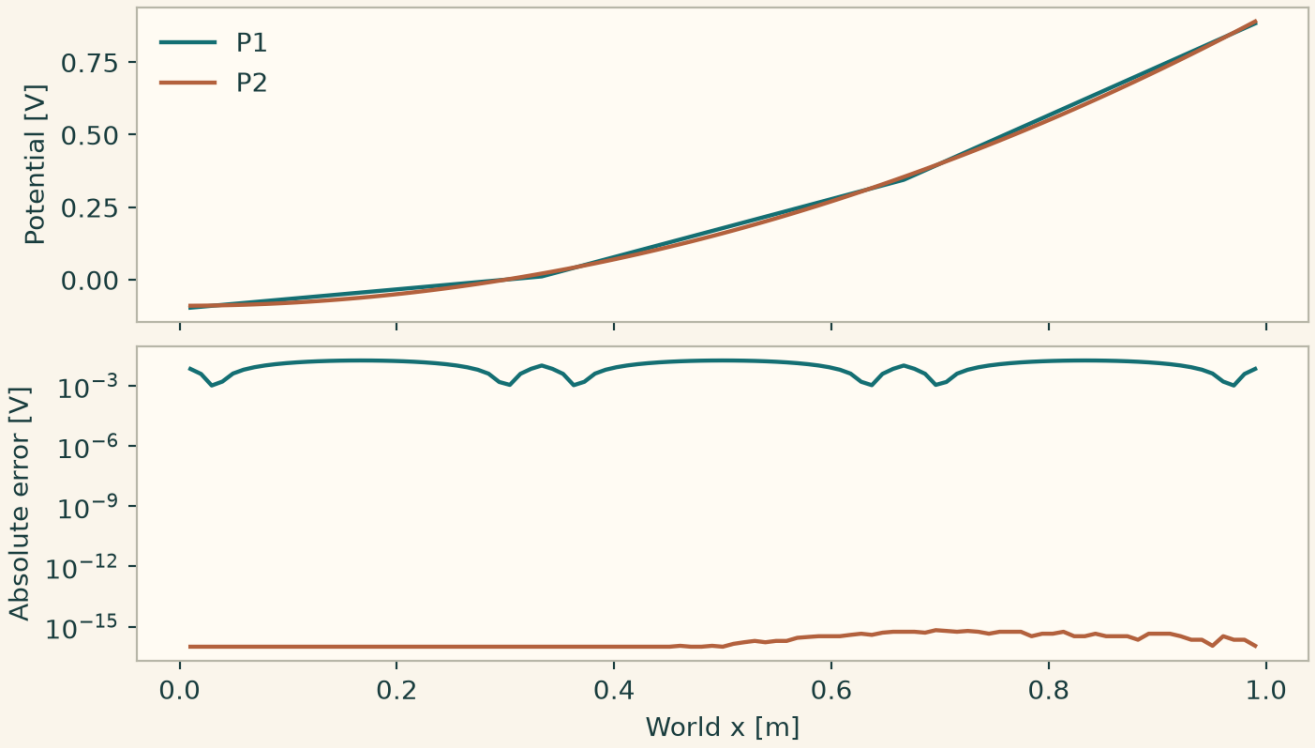
9.4 Finite-element lab

9.4.1 Compare finite-element orders on a manufactured harmonic field

Prescribe a harmonic quadratic potential on the box boundary. Its representation distinguishes first- and second-order spaces. Use physical evaluation and a logarithmic error panel for the comparison.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import VacuumPermittivity
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.linear import SparseDirect
from onesim.physics.electrostatics import FixedPotential, PoissonEquation
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
model = PoissonEquation(domain=box.volume, permittivity=VacuumPermittivity())


def exact(x, y):
    return x * x - y * y


problem = Problem(
    name="Harmonic potential",
    geometry=box,
    physics=(model,),
    boundary_conditions=(
```

```
        FixedPotential(physics=model, boundary=box.boundaries.all, value=exact),
    ),
)
mesh = StructuredTetrahedralMesher(cells=(3, 3, 3)).generate(box)
x = np.linspace(0.01, 0.99, 101)
points = np.column_stack((x, np.full_like(x, 0.3), np.full_like(x, 0.55)))
reference = exact(points[:, 0], points[:, 1])
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
errors = {}
for p in (1, 2):
    sim = Simulation(
        problem=problem,
        mesh=mesh,
        numerics=Numerics(
            spatial=FiniteElement(element=Lagrange(order=p)),
            linear_solver=SparseDirect(),
        ),
    )
    result = sim.solve_stationary()
    potential = result.field(model.potential).evaluate(points)
    errors[p] = np.max(np.abs(potential - reference))
    a.plot(x, potential, label=f"P{p}")
    b.semilogy(x, np.maximum(np.abs(potential - reference), 1e-16), label=f"P{p}")
assert errors[2] < 1e-10 < errors[1]
a.set_ylabel("Potential [V]")
a.legend(frameon=False)
b.set(xlabel="World x [m]", ylabel="Absolute error [V]")
plt.show()
```

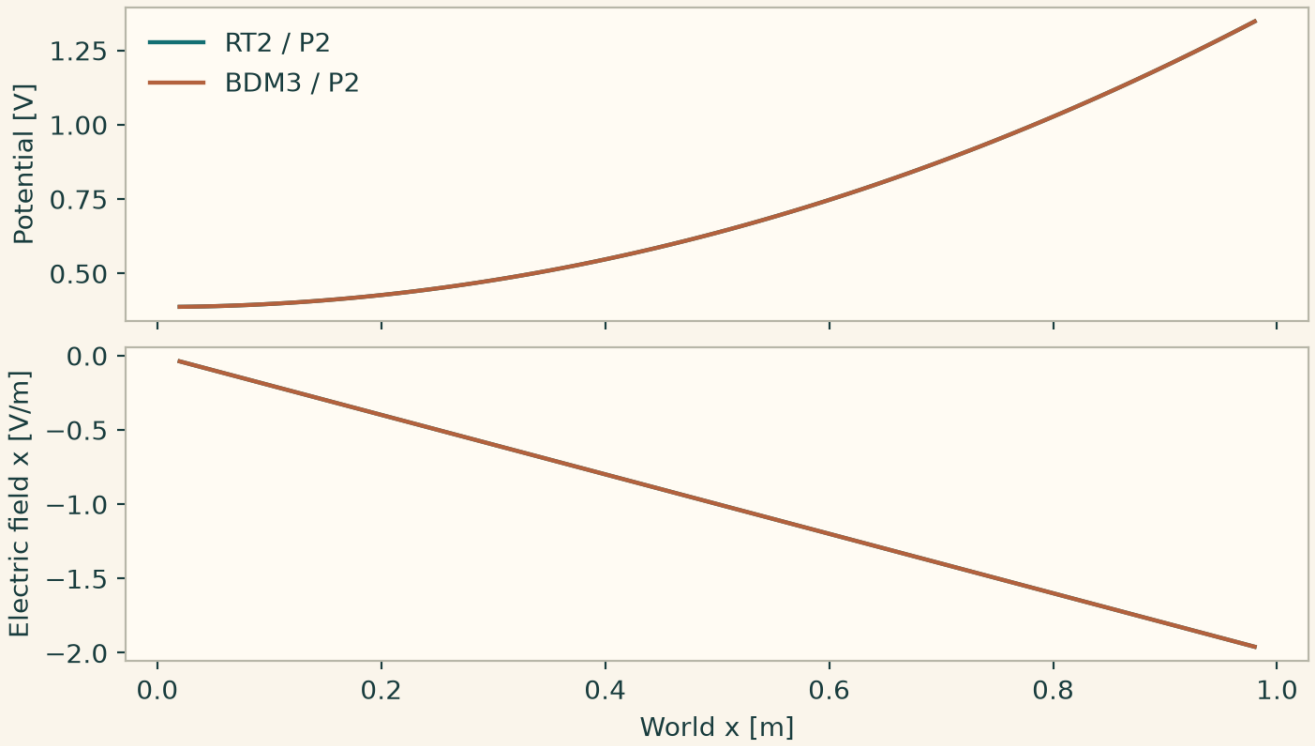


9.4.2 Mixed normal-flux elements for the Poisson balance

Pair each H(div) flux family with its compatible discontinuous scalar order. Solve a manufactured quadratic potential whose gradient is linear and compare the physical potential and electric field along the same line.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import (
    BrezziDouglasMarini,
    DiscontinuousLagrange,
    FiniteElement,
    MixedDiffusionElement,
    RaviartThomas,
)
from onesim.numerics.linear import SparseDirect
from onesim.physics.electrostatics import FixedPotential, PoissonEquation
from onesim.simulation import Problem, Simulation


class ConstantPermittivity:
    unit = "F/m"

    def evaluate(self, **state):
        return 2.0
```

```
box = Box(length=1, width=1, height=1)
model = PoissonEquation(
    domain=box.volume, permittivity=ConstantPermittivity(), charge_density=-12.0
)


def exact(x, y, z):
    return x * x + y * y + z * z


problem = Problem(
    name="Mixed potential and flux",
    geometry=box,
    physics=(model,),
    boundary_conditions=(
        FixedPotential(physics=model, boundary=box.boundaries.all, value=exact),
    ),
)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
x = np.linspace(0.02, 0.98, 91)
p = np.column_stack((x, np.full_like(x, 0.27), np.full_like(x, 0.56)))
theme.use_theme()
fig, (a, b) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
)
for flux, degree, label in (
    (RaviartThomas(2), 2, "RT2 / P2"),
    (BrezziDouglasMarini(3), 2, "BDM3 / P2"),
):
    spatial = FiniteElement(
        element=MixedDiffusionElement(flux=flux, scalar=DiscontinuousLagrange(degree))
    )
    result = Simulation(
        problem=problem,
        mesh=mesh,
        numerics=Numerics(spatial=spatial, linear_solver=SparseDirect()),
    ).solve_stationary()
    phi = result.field(model.potential).evaluate(p)
    ex = result.field(model.electric_field).evaluate(p)[:, 0]
    np.testing.assert_allclose(phi, exact(*p.T), atol=1e-8)
    np.testing.assert_allclose(ex, -2 * x, atol=1e-8)
    a.plot(x, phi, label=label)
    b.plot(x, ex, label=label)
a.set_ylabel("Potential [V]")
a.legend(frameon=False)
b.set(xlabel="World x [m]", ylabel="Electric field x [V/m]")
plt.show()
```



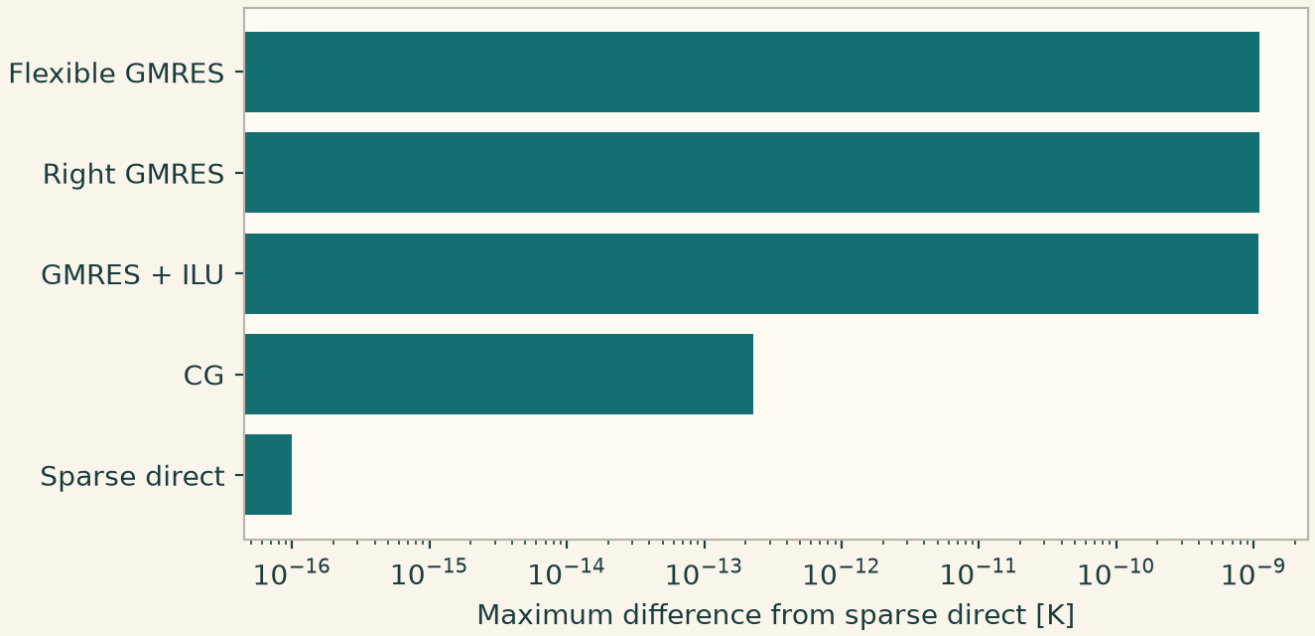
9.5 Linear algebra lab

9.5.1 Linear solvers against a common direct reference

Keep the physical conduction problem and FE space fixed. Measure the maximum difference from sparse direct solution and inspect the solver report for the executed provider. CG needs a suitable symmetric operator. Right and flexible GMRES own different preconditioning contracts.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import IsotropicSolid
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.linear import (
    SparseDirect,
    ConjugateGradient,
    GMRES,
    RightGMRES,
    FlexibleGMRES,
    ILU,
)
from onesim.physics.heat_transfer import FixedTemperature, HeatConduction
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
material = IsotropicSolid(
    density=1,
```

```
    specific_heat_capacity=1,
    thermal_conductivity=1,
    youngs_modulus=100,
    poisson_ratio=0.3,
    thermal_expansion_coefficient=0,
)
model = HeatConduction(domain=box.volume, material=material, heating=1.0)
problem = Problem(
    name="Heated box",
    geometry=box,
    physics=(model,),
    boundary_conditions=(
        FixedTemperature(physics=model, boundary=box.boundaries.all, value=300.0),
    ),
)
mesh = StructuredHexahedralMesher(cells=(5, 5, 5)).generate(box)
space = FiniteElement(element=Lagrange(1))
options = dict(
    relative_tolerance=1e-11, absolute_tolerance=1e-12, maximum_iterations=500
)
solvers = {
    "Sparse direct": SparseDirect(),
    "CG": ConjugateGradient(**options),
    "GMRES + ILU": GMRES(preconditioner=ILU(), **options),
    "Right GMRES": RightGMRES(preconditioner=ILU(), **options),
    "Flexible GMRES": FlexibleGMRES(preconditioner=ILU(), **options),
}
solutions = {}
for label, solver in solvers.items():
    sim = Simulation(
        problem=problem,
        mesh=mesh,
        numerics=Numerics(spatial=space, linear_solver=solver),
    )
    result = sim.solve_stationary()
    solutions[label] = np.asarray(result.field(model.temperature).values)
    print(label, result.result_set.solver_report)
reference = solutions["Sparse direct"]
errors = [np.max(np.abs(y - reference)) for y in solutions.values()]
assert max(errors) < 2e-8
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.3), layout="constrained")
ax.barh(list(solvers), np.maximum(errors, 1e-16))
ax.set(xscale="log", xlabel="Maximum difference from sparse direct [K]")
plt.show()
```



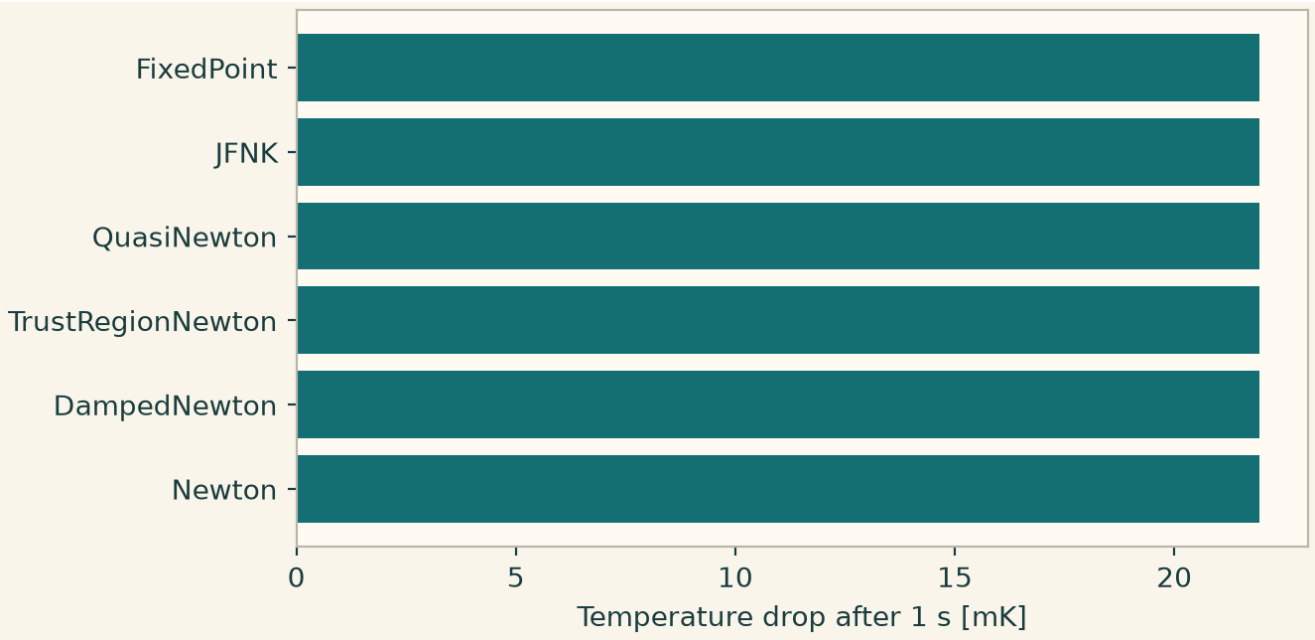
9.6 Nonlinear solver lab

9.6.1 Solve the same nonlinear implicit cooling stage

Select the nonlinear method separately from backward Euler. Compare the final temperature to the Newton route and keep the time grid fixed. The difference isolates stage convergence, while the bias relative to the analytic cooling curve also includes time error.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.linear import GMRES, SparseDirect
from onesim.numerics.nonlinear import (
    Newton,
    DampedNewton,
    TrustRegionNewton,
    QuasiNewton,
    JFNK,
    FixedPoint,
)
from onesim.numerics.time import BackwardEuler
from onesim.physics.heat_transfer import (
    RadiationCooling,
    ThermalGreyDiffuseRadiationBoundary,
)
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
model = RadiationCooling(
```

```
    domain=domain.system,
    heat_capacity=2,
    radiating_area=0.3,
    radiation=ThermalGreyDiffuseRadiationBoundary(
        condition_id="radiation",
        emissivity=0.5,
        ambient_temperature=1,
        stefan_boltzmann=0.02,
    ),
)
problem = Problem(
    name="Nonlinear cooling stage",
    geometry=domain,
    physics=(model,),
    initial_conditions=(InitialValue(model.temperature, 2.0),),
)
settings = dict(relative_tolerance=1e-12, absolute_tolerance=1e-13)
solvers = {
    s.__name__: s(linear_solver=SparseDirect(), jacobian="analytic", **settings)
    for s in (Newton, DampedNewton, TrustRegionNewton, QuasiNewton)
}
solvers["JFNK"] = JFNK(linear_solver=GMRES(), jacobian="analytic", **settings)
solvers["FixedPoint"] = FixedPoint(**settings)
values = {}
for label, solver in solvers.items():
    sim = Simulation(
        problem=problem,
        mesh=LumpedMesh(domain),
        numerics=Numerics(
            spatial=LumpedParameters(),
            time_integrator=BackwardEuler(),
            nonlinear_solver=solver,
        ),
    )
    result = sim.run(start_time=0, end_time=1, time_step=0.05)
    values[label] = np.asarray(result.field(model.temperature).values).item()
assert max(abs(y - values["Newton"]) for y in values.values()) < 1e-9
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.3), layout="constrained")
ax.barh(list(values), [(2 - y) * 1e3 for y in values.values()])
ax.set_xlabel("Temperature drop after 1 s [mK]")
plt.show()
```



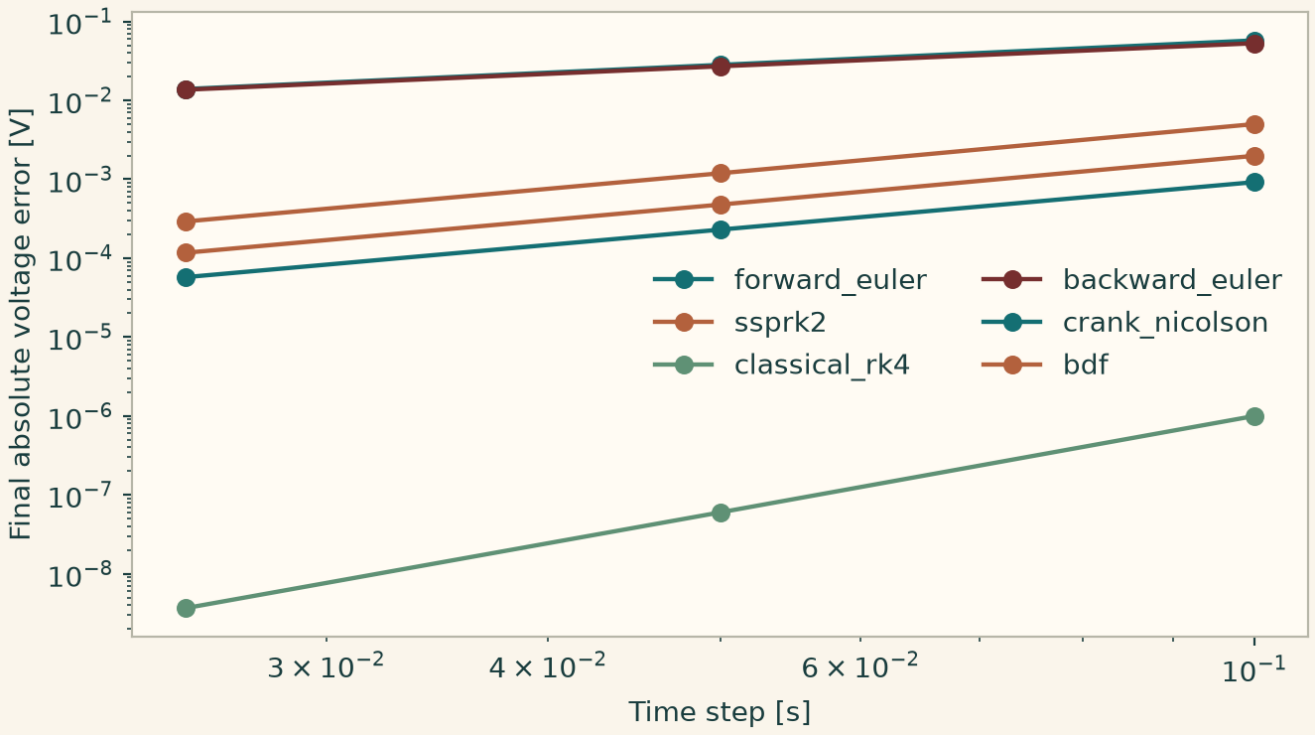
9.7 Time integration and continuation lab

9.7.1 Separate temporal error from the physical model

Repeat an RC step with successively smaller time steps. The plotted error is the final voltage error against an exact exponential. A steady mesh-free model isolates the time method.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import (
    TimeMethod,
    ForwardEuler,
    ClassicalRungeKutta4,
    BackwardEuler,
    BDF,
)
from onesim.physics.systems import ResistorCapacitor
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
rc = ResistorCapacitor(
    domain=domain.system, resistance=2, capacitance=0.5, supply_voltage=3
)
problem = Problem(
    name="RC time convergence",
    geometry=domain,
```

```
    physics=(rc,),
    initial_conditions=(InitialValue(rc.voltage, 0.0),),
)
methods = (
    ForwardEuler(),
    TimeMethod("ssprk2"),
    ClassicalRungeKutta4(),
    BackwardEuler(),
    TimeMethod("crank_nicolson"),
    BDF(order=2),
)
steps = (0.1, 0.05, 0.025)
exact = 3 * (1 - np.exp(-1))
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for method in methods:
    errors = []
    solver = (
        Newton(linear_solver=SparseDirect(), jacobian="analytic")
        if method.implicit
        else None
    )
    sim = Simulation(
        problem=problem,
        mesh=LumpedMesh(domain),
        numerics=Numerics(
            spatial=LumpedParameters(),
            time_integrator=method,
            nonlinear_solver=solver,
        ),
    )
    for dt in steps:
        result = sim.run(start_time=0, end_time=1, time_step=dt)
        y = np.asarray(result.field(rc.voltage).values).item()
        errors.append(abs(y - exact))
    assert errors[-1] < errors[0]
    ax.loglog(steps, errors, "o-", label=method.method_id)
ax.set(xlabel="Time step [s]", ylabel="Final absolute voltage error [V]")
ax.legend(frameon=False, ncol=2)
plt.show()
```



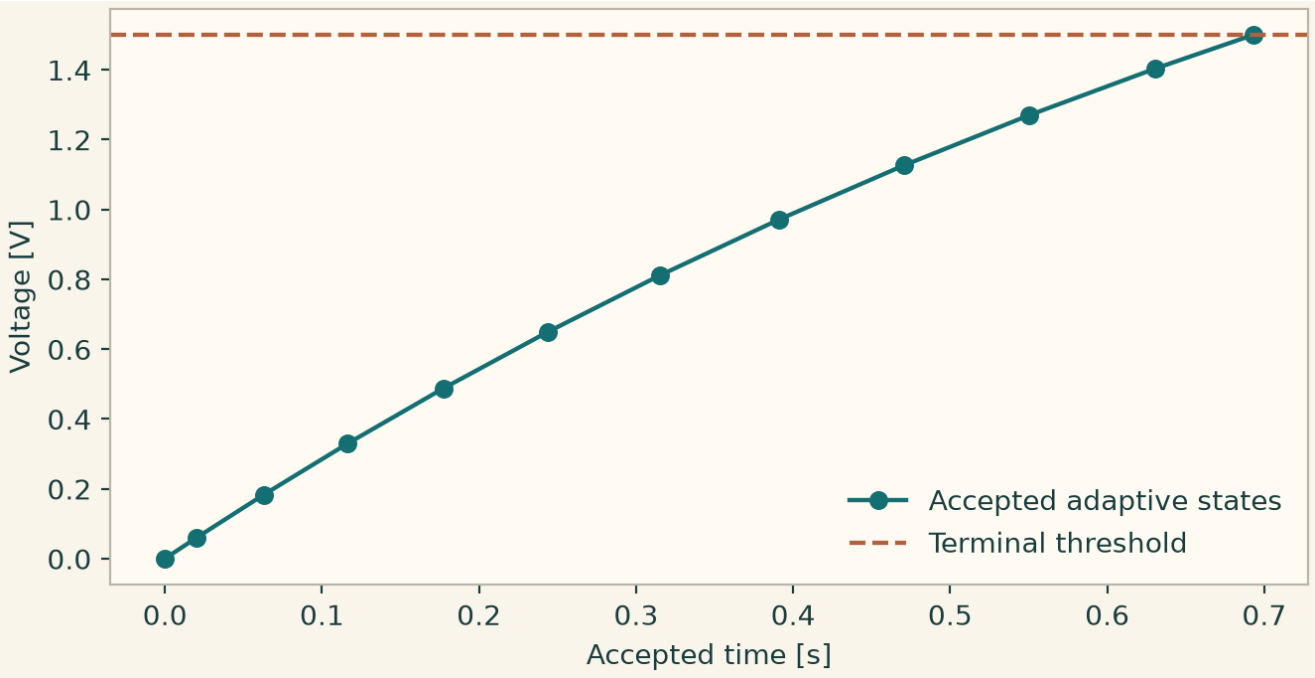
9.7.2 Adaptive field threshold and BDF continuation

Use a typed event on capacitor voltage to stop an adaptive run at a known threshold. Independently, checkpoint a fixed-step BDF run at an aligned time and compare its restart against a run with the same time history.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import AdaptiveTimeStep, ClassicalRungeKutta4, BDF
from onesim.physics.systems import ResistorCapacitor
from onesim.simulation import Problem, Simulation, ExecutionOptions, FieldThreshold
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
rc = ResistorCapacitor(
    domain=domain.system, resistance=2, capacitance=0.5, supply_voltage=3
)
problem = Problem(
    name="RC threshold",
    geometry=domain,
    physics=(rc,),
    initial_conditions=(InitialValue(rc.voltage, 0.0),),
)
mesh = LumpedMesh(domain)
```

```
explicit = Simulation(
    problem=problem,
    mesh=mesh,
    numerics=Numerics(
        spatial=LumpedParameters(), time_integrator=ClassicalRungeKutta4()
    ),
)
event = FieldThreshold(
    field=rc.voltage,
    value=1.5,
    event_id="half-charge",
    direction=1,
    value_tolerance=1e-10,
)
adaptive = explicit.run(
    start_time=0,
    end_time=1.2,
    time_step=AdaptiveTimeStep(
        initial=0.02, minimum=1e-6, maximum=0.08, relative_tolerance=1e-7
    ),
    execution_options=ExecutionOptions(events=(event,)),
)
assert abs(adaptive.times[-1] - np.log(2)) < 2e-5
implicit = Simulation(
    problem=problem,
    mesh=mesh,
    numerics=Numerics(
        spatial=LumpedParameters(),
        time_integrator=BDF(order=2),
        nonlinear_solver=Newton(linear_solver=SparseDirect(), jacobian="analytic"),
    ),
)
checkpoints = ExecutionOptions(checkpoint_interval=0.4)
first = implicit.run(
    start_time=0, end_time=0.4, time_step=0.02, execution_options=checkpoints
)
resumed = implicit.run(
    start_time=0,
    end_time=0.8,
    time_step=0.02,
    execution_options=ExecutionOptions(restart_checkpoint=first.last_checkpoint),
)
full = implicit.run(
    start_time=0, end_time=0.8, time_step=0.02, execution_options=checkpoints
)
np.testing.assert_allclose(
    resumed.field(rc.voltage).values, full.field(rc.voltage).values, atol=1e-12
)
t = np.asarray(adaptive.times)
y = np.asarray(
    [adaptive.field(rc.voltage).at(float(time)).values.item() for time in t]
```

```
)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.5), layout="constrained")
ax.plot(t, y, "o-", label="Accepted adaptive states")
ax.axhline(1.5, color=theme.COPPER, linestyle="--", label="Terminal threshold")
ax.set(xlabel="Accepted time [s]", ylabel="Voltage [V]")
ax.legend(frameon=False)
plt.show()
```



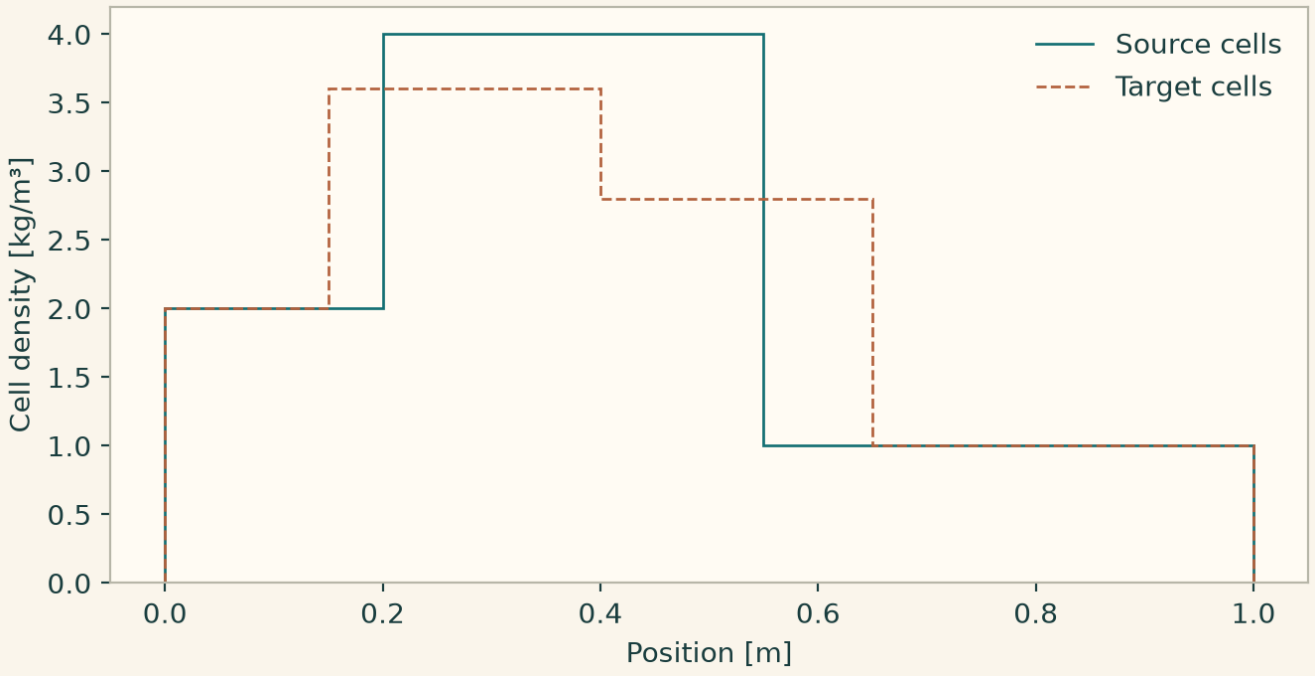
9.8 Field transfer lab

9.8.1 Conservative remap between nonmatching cell edges

Start with cell averages on one line partition and transfer them to a different partition. Verify the width-weighted integral. Plot native piecewise-constant values rather than drawing a smooth interpolant.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.numerics.field_transfer import operators

source_edges = np.array([0.0, 0.2, 0.55, 1.0])
target_edges = np.array([0.0, 0.15, 0.4, 0.65, 0.85, 1.0])
source = np.array([2.0, 4.0, 1.0])
mapping = operators.conservative_remap(
    "density transfer",
    "source",
    "target",
    source_edges=source_edges,
    target_edges=target_edges,
    source_unit="kg/m^3",
```

```
    target_unit="kg/m^3",
)
validation = operators.validate(mapping, source=source)
assert validation.accepted, validation.diagnostics
result = operators.apply(mapping, source)
np.testing.assert_allclose(
    np.diff(target_edges) @ result.value, np.diff(source_edges) @ source
)
assert result.report.conservation_verified
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.5), layout="constrained")
ax.stairs(source, source_edges, label="Source cells")
ax.stairs(result.value, target_edges, label="Target cells", linestyle="--")
ax.set(xlabel="Position [m]", ylabel="Cell density [kg/m³]")
ax.legend(frameon=False)
plt.show()
```



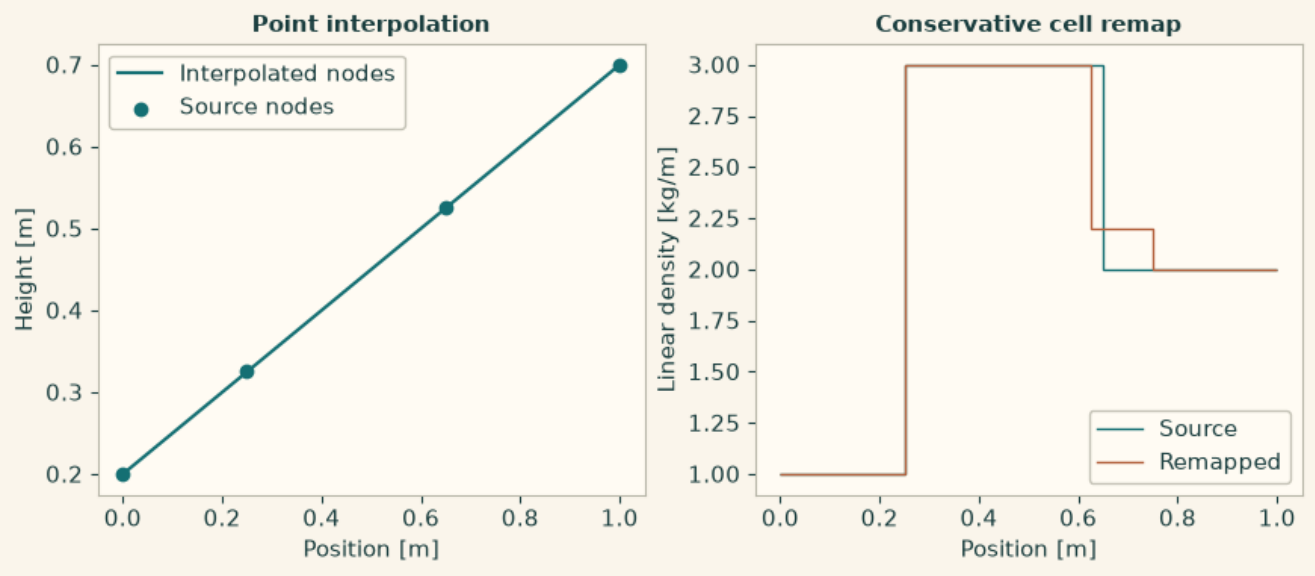
9.8.2 Copy, interpolate and conservatively remap on a line

Copy nodal heights on an unchanged grid. Interpolate them onto a finer grid and convert centimetres to metres. Check which provider validation selects and inspect the diagnostic for a nonfinite source value. For cell averages, multiply each mean by its cell width to verify that the overlap remap preserves the total mass.

```
import onesim
import onesimulation_plot_theme as theme

theme.use_theme()

import matplotlib.pyplot as plt
import numpy as np
from onesim.numerics.field_transfer import operators as transfer_ops
```

```
source_x = np.array([0.0, 0.25, 0.65, 1.0])
target_x = np.linspace(0, 1, 21)
source_cm = 100 * (0.2 + 0.5 * source_x)

same_grid = transfer_ops.identity("copy heights", "coarse nodes",
                                  unit="cm", topology="node")
copied = transfer_ops.apply(same_grid, source_cm)
np.testing.assert_array_equal(copied.value, source_cm)

linear = transfer_ops.interpolation(
    "interpolate heights", "coarse nodes", "fine nodes",
    source_coordinates=source_x, target_coordinates=target_x,
    source_unit="cm", target_unit="m",
    source_topology="node", target_topology="node",
)
preflight = transfer_ops.validate(linear, source=copied.value).require_accepted()
invalid = transfer_ops.validate(linear, source=[0.0, np.nan, 1.0, 2.0])
assert not invalid.accepted
heights = transfer_ops.apply(linear, copied.value)
np.testing.assert_allclose(heights.value, 0.2 + 0.5 * target_x)
assert heights.report.provider_id == preflight.provider_id

source_edges = np.array([0.0, 0.25, 0.65, 1.0])
target_edges = np.linspace(0, 1, 9)
source_density = np.array([1.0, 3.0, 2.0])
remap = transfer_ops.conservative_remap(
    "remap cell density", "source cells", "target cells",
    source_edges=source_edges, target_edges=target_edges,
    source_unit="kg/m", target_unit="kg/m",
)
density = transfer_ops.apply(remap, source_density)
source_mass = source_density @ np.diff(source_edges)
target_mass = density.value @ np.diff(target_edges)
np.testing.assert_allclose(target_mass, source_mass, rtol=1e-12)
assert density.report.conservation_verified

print(f"Interpolation provider: {preflight.provider_id}")
print(f"Rejected input: {invalid.diagnostics[0]}")
print(f"Integrated mass: {source_mass:.3f} kg → {target_mass:.3f} kg")

fig, (ax, cells_ax) = plt.subplots(1, 2, figsize=(8.6, 3.7), layout="constrained")
ax.plot(target_x, heights.value, label="Interpolated nodes")
ax.scatter(source_x, source_cm / 100, label="Source nodes", zorder=3)
ax.set(title="Point interpolation", xlabel="Position [m]", ylabel="Height [m]")
ax.legend()
cells_ax.stairs(source_density, source_edges, label="Source", baseline=None)
cells_ax.stairs(density.value, target_edges, label="Remapped", baseline=None)
cells_ax.set(title="Conservative cell remap", xlabel="Position [m]",
             ylabel="Linear density [kg/m]")
```

```
cells_ax.legend()
plt.show()
```



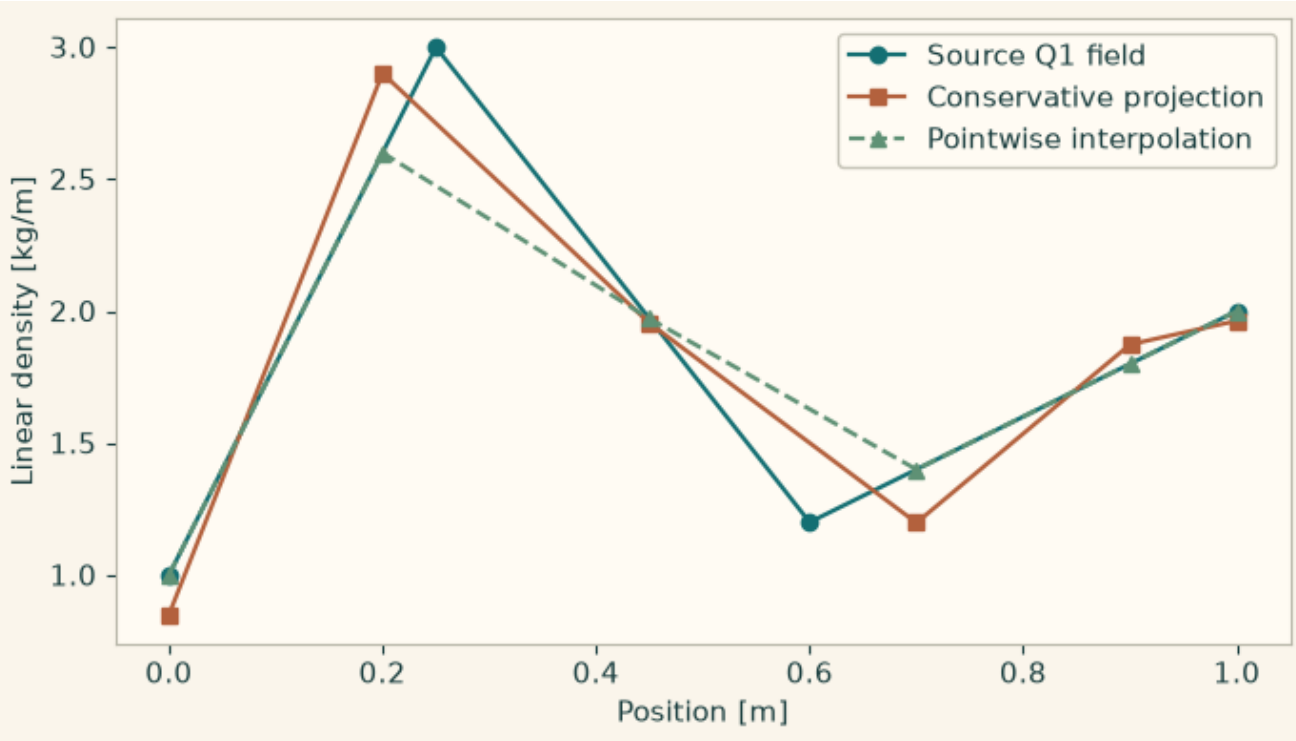
Linear interpolation and conservative transfer on different one-dimensional grids.

9.8.3 Configure a runtime for conservative nodal projection

Wrap a transfer descriptor in a named operator. Configure the runtime to assemble the Q1 projection with the NumPy kernel and solve its mass system with a sparse direct solver. Compare with pointwise interpolation on the same target nodes. The area under the piecewise-linear field is its total mass.

```
import onesim
import onesimulation_plot_theme as theme

theme.use_theme()

import matplotlib.pyplot as plt
import numpy as np
from onesim.numerics.field_transfer import operators as transfer_ops
from onesim.numerics.field_transfer.state.models.transfer_descriptor import
    TransferDescriptor
from onesim.numerics.linear import SparseDirect

source_x = (0.0, 0.25, 0.6, 1.0)
target_x = (0.0, 0.2, 0.45, 0.7, 0.9, 1.0)
source = np.array([1.0, 3.0, 1.2, 2.0])

descriptor = TransferDescriptor(
    source_layout_id="coarse Q1 line",
    target_layout_id="fine Q1 line",
    conservative=True,
    mapping_kind="cartesian_conservative_nodal_projection",
    source_axes=(source_x,),
    target_axes=(target_x,),
```

```
    source_topology="node",
    target_topology="node",
    source_unit="kg/m",
    target_unit="kg/m",
)
projection = transfer_ops.operator("project Q1 nodal density", descriptor)
runtime = transfer_ops.runtime(linear=SparseDirect().configuration(),
    projection_kernel="numpy")
transfer_ops.validate(projection, source=source, runtime=runtime).require_accepted()
projected = transfer_ops.apply(projection, source, runtime=runtime)
pointwise = np.interp(target_x, source_x, source)
m_source = np.trapezoid(source, source_x)
m_projected = np.trapezoid(projected.value, target_x)
m_pointwise = np.trapezoid(pointwise, target_x)

np.testing.assert_allclose(m_projected, m_source, rtol=1e-12)
assert projected.report.conservation_verified
assert projected.report.linear_solver_reports
assert abs(m_pointwise - m_source) > 1e-3
print(f"Projection provider: {projected.report.provider_id}")
print(f"Mass [kg] — source {m_source:.5f}, projected {m_projected:.5f}, "
      f"pointwise {m_pointwise:.5f}")

fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(source_x, source, "o-", label="Source Q1 field")
ax.plot(target_x, projected.value, "s-", label="Conservative projection")
ax.plot(target_x, pointwise, "^--", label="Pointwise interpolation")
ax.set(xlabel="Position [m]", ylabel="Linear density [kg/m]")
ax.legend()
plt.show()
```



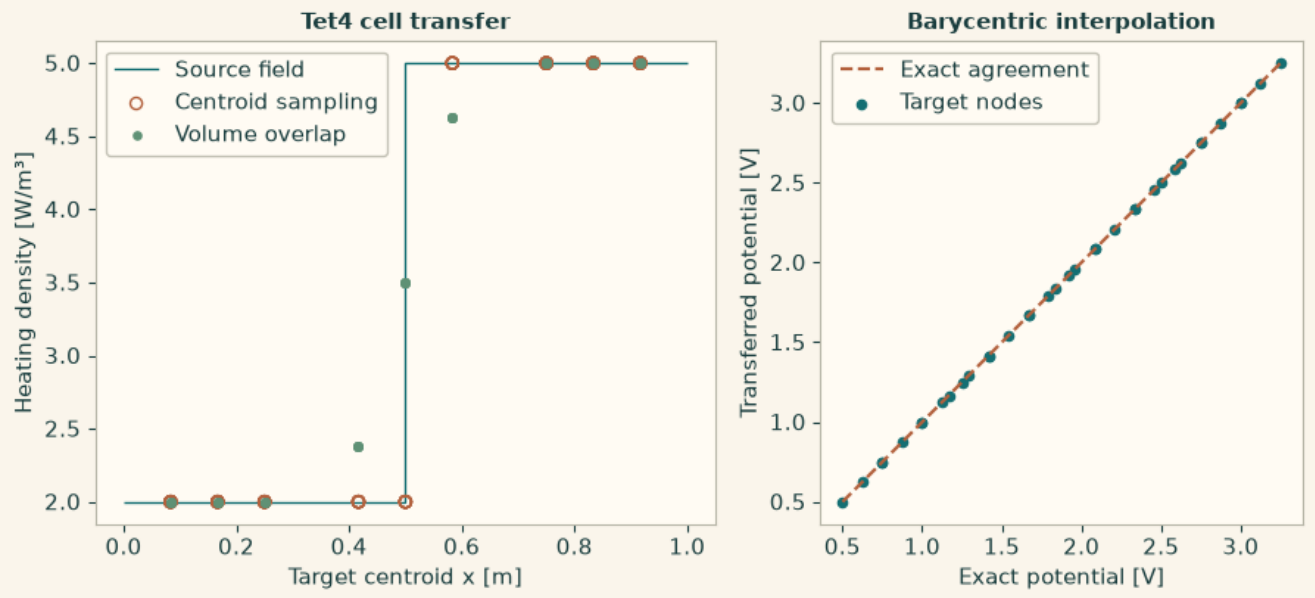
Nodal projection and pointwise interpolation on the same target grid.

9.8.4 Transfer cell and nodal fields between simplex meshes

Give two Tet4 meshes of the same box to the cell operators. Centroid projection samples the source cell containing each target centroid. The overlap remap integrates across cell intersections to preserve total power. For a nodal field, provide the ordered coordinates and connectivity of the source mesh and check that barycentric interpolation reproduces an affine potential.

```
import onesim
import onesimulation_plot_theme as theme

theme.use_theme()

import matplotlib.pyplot as plt
import numpy as np
from onesim.cad.d3 import Box
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics.field_transfer import operators as transfer_ops

box = Box(length=1, width=1, height=1)
source_mesh = StructuredTetrahedralMesher(cells=(2, 1, 1)).generate(box)
target_mesh = StructuredTetrahedralMesher(cells=(3, 2, 2)).generate(box)
source_cells = source_mesh.to_carrier()
target_cells = target_mesh.to_carrier()
source_x = source_mesh.coordinates[source_mesh.cells, 0].mean(axis=1)
target_x = target_mesh.coordinates[target_mesh.cells, 0].mean(axis=1)
source_density = np.where(source_x < 0.5, 2.0, 5.0)

sample = transfer_ops.tetrahedral_cell_projection(
    "sample at target centroids", source_cells, target_cells, unit="W/m^3",
)
overlap = transfer_ops.tetrahedral_conservative_remap(
    "remap by Tet4 intersection volumes", source_cells, target_cells, unit="W/m^3",
)
sampled = transfer_ops.apply(sample, source_density)
remapped = transfer_ops.apply(overlap, source_density)
power_in = source_density @ np.asarray(source_cells.measures)
power_out = remapped.value @ np.asarray(target_cells.measures)
np.testing.assert_allclose(power_out, power_in, rtol=1e-11)
assert not sampled.report.conservative
assert remapped.report.conservation_verified
assert np.any((remapped.value > 2.01) & (remapped.value < 4.99))

# Choose an affine nodal field with a known value at every target node.
def potential(points):
    return 1 + points @ np.array([2.0, -0.5, 0.25])

def nodal_support(mesh):
    return {
        "carrier_id": mesh.to_carrier(location="node").carrier_id,
```

```
        "carrier_kind": "node",
        "coordinate_unit": "m",
        "coordinates": mesh.coordinates.tolist(),
        "connectivity": mesh.cells.tolist(),
    }

interpolate = transfer_ops.simplex_nodal_interpolation(
    "barycentric interpolation",
    nodal_support(source_mesh), nodal_support(target_mesh),
    unit="V",
)
target_potential = transfer_ops.apply(interpolate, potential(source_mesh.coordinates))
reference = potential(target_mesh.coordinates)
np.testing.assert_allclose(target_potential.value, reference, atol=1e-12)
print(f"Integrated power: {power_in:.6f} W → {power_out:.6f} W")
print(f"Affine test: {reference.size} target nodes")

fig, (cell_ax, node_ax) = plt.subplots(1, 2, figsize=(9.3, 4.0), layout="constrained")
cell_ax.stairs([2, 5], [0, 0.5, 1], label="Source field", baseline=None)
cell_ax.scatter(target_x, sampled.value, s=34, facecolors="none",
                edgecolors="C1", label="Centroid sampling")
cell_ax.scatter(target_x, remapped.value, s=14, color="C2", label="Volume overlap")
cell_ax.set(title="Tet4 cell transfer", xlabel="Target centroid x [m]",
            ylabel="Heating density [W/m³]")
cell_ax.legend()
node_ax.plot([reference.min(), reference.max()], [reference.min(), reference.max()],
             "--", color="C1", label="Exact agreement")
node_ax.scatter(reference, target_potential.value, s=20, label="Target nodes")
node_ax.set(title="Barycentric interpolation", xlabel="Exact potential [V]",
            ylabel="Transferred potential [V]", aspect="equal")
node_ax.legend()
plt.show()
```



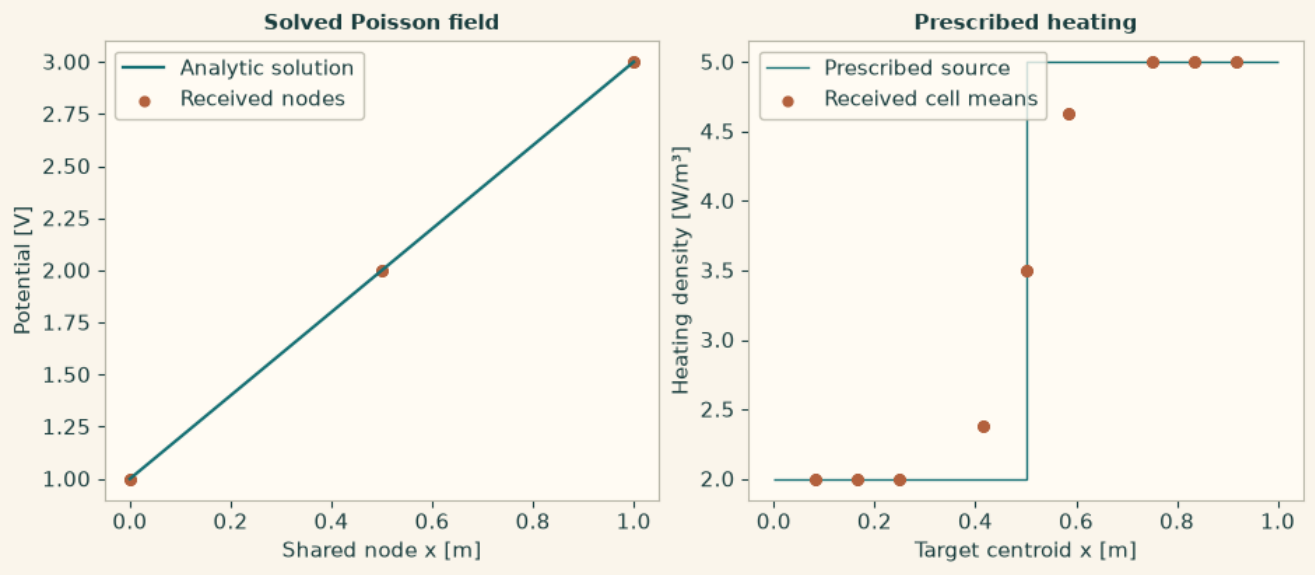
Centroid sampling, volume-overlap remapping and nodal interpolation on Tet4 meshes.

9.8.5 Resolve solved and prescribed fields for named inputs

Declare an input from a Poisson result and resolve it on the node layout published by the solve. Declare heating directly from cell data and resolve it on a different Tet4 mesh using conservative remapping. The two reports record the named receiving ports, mapping methods and source identities.

```
import onesim
import onesimulation_plot_theme as theme

theme.use_theme()

import matplotlib.pyplot as plt
import numpy as np
from onesim.cad.d3 import Box
from onesim.materials import VacuumPermittivity
from onesim.meshing import StructuredTetrahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.field_transfer import operators as transfer_ops
from onesim.numerics.linear import SparseDirect
from onesim.physics.electrostatics import FixedPotential, PoissonEquation
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
source_mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
target_mesh = StructuredTetrahedralMesher(cells=(3, 2, 2)).generate(box)

def exact(x, y, z):
    return 1 + 2 * x

physics = PoissonEquation(domain=box.volume, permittivity=VacuumPermittivity())
problem = Problem(
    name="Poisson source for field handoff",
    geometry=box,
    physics=(physics,),
    boundary_conditions=(
        FixedPotential(physics=physics, boundary=box.boundaries.all, value=exact),
    ),
)
result = Simulation(
    problem=problem,
    mesh=source_mesh,
    numerics=Numerics(
        spatial=FiniteElement(element=Lagrange(1)), linear_solver=SparseDirect(),
    ),
).solve_stationary()
voltage = transfer_ops.solved_field(
    result, physics.potential.field_id, unit="V",
    input_id="poisson_voltage", target_system_id="receiver", target_field_id="voltage",
)
# The voltage receiver shares the ordered node layout published by the solve.
```

```
voltage_support = result.result_set.field(physics.potential.field_id).metadata["spatial
    _support"]
received_voltage = voltage.resolve(voltage_support)
voltage_points = np.asarray(voltage_support["coordinates"])
expected = exact(*voltage_points.T)
np.testing.assert_allclose(received_voltage.values, expected, atol=1e-10)
assert received_voltage.report["source_result_id"] == result.result_set.result_id
assert received_voltage.report["mapping_kind"] == "identity"

source_cells = source_mesh.to_carrier()
target_cells = target_mesh.to_carrier()
source_centroids_x = source_mesh.coordinates[source_mesh.cells, 0].mean(axis=1)
target_centroids_x = target_mesh.coordinates[target_mesh.cells, 0].mean(axis=1)
power_density = np.where(source_centroids_x < 0.5, 2.0, 5.0)
heating = transfer_ops.prescribed_field(
    power_density, source_cells, "heating", unit="W/m^3", value_kind="scalar",
    input_id="specified_heating", target_system_id="receiver",
    target_field_id="power_density", transfer="tetrahedral_conservative_remap",
)
received_heat = heating.resolve(target_cells)
assert received_heat.report["source_result_id"].startswith("prescribed:heating:")
assert received_heat.report["conservation_verified"]
source_power = power_density @ np.asarray(source_cells.measures)
target_power = received_heat.values @ np.asarray(target_cells.measures)
np.testing.assert_allclose(target_power, source_power, rtol=1e-11)

for resolved in (received_voltage, received_heat):
    print(f"{resolved.report['input_id']} → {resolved.report['target_field_id']} "
          f"via {resolved.report['mapping_kind']}")
print(f"Received integrated power: {target_power:.6f} W")

fig, (ax, heat_ax) = plt.subplots(1, 2, figsize=(9.3, 4.0), layout="constrained")
ax.plot([0, 1], [1, 3], label="Analytic solution")
ax.scatter(voltage_points[:, 0], received_voltage.values, s=24, color="C1",
    label="Received nodes")
ax.set(title="Solved Poisson field", xlabel="Shared node x [m]", ylabel="Potential
    [V]")
ax.legend()
heat_ax.stairs([2, 5], [0, 0.5, 1], label="Prescribed source", baseline=None)
heat_ax.scatter(target_centroids_x, received_heat.values, s=24,
                color="C1", label="Received cell means")
heat_ax.set(title="Prescribed heating", xlabel="Target centroid x [m]",
            ylabel="Heating density [W/m³]")
heat_ax.legend()
plt.show()
```



Poisson potential on the shared node layout and prescribed heating on the receiving cells.

9.9 Coupled execution lab

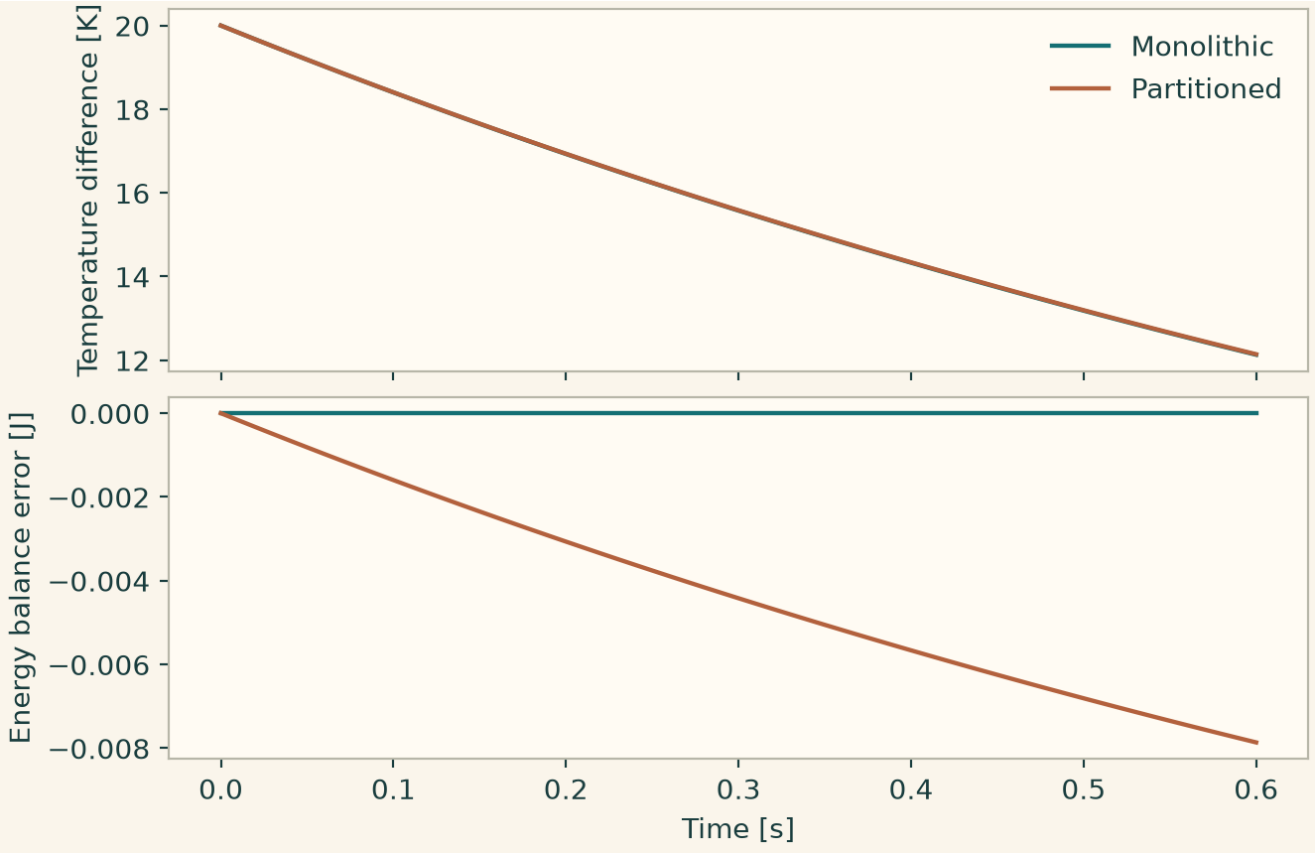
9.9.1 Compare monolithic and partitioned thermal exchange

Two transient heat capacities send their temperatures to one another through typed input ports. The monolithic method conserves their capacity-weighted sum to roundoff. The partitioned trajectory has a small time-discretization balance error, which is shown separately.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.coupled_systems import MonolithicCoupling, PartitionedCoupling
from onesim.numerics.discretization import LumpedParameters
from onesim.numerics.field_transfer import GlobalValueTransfer
from onesim.numerics.nonlinear import FixedPoint
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.systems import ThermalCapacitance
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.field_connections import FieldCoupling


class CoupledStep:
    def maximum_step(self, **kwargs):
        return 0.02


domain = LumpedDomain()
a = ThermalCapacitance(
    domain=domain.system, heat_capacity=2, conductance=1, ambient_temperature=None
)
```

```
b = ThermalCapacitance(
    domain=domain.system, heat_capacity=3, conductance=1, ambient_temperature=None
)
models = (a, b)
problem = Problem(
    name="Closed exchange",
    geometry=domain,
    physics=models,
    initial_conditions=(
        InitialValue(a.temperature, 320.0),
        InitialValue(b.temperature, 300.0),
    ),
    couplings=tuple(
        FieldCoupling(
            source=s.temperature, target=d.ambient, transfer=GlobalValueTransfer()
        )
        for s, d in ((a, b), (b, a))
    ),
)
mesh = LumpedMesh(domain)
method = ClassicalRungeKutta4()
options = {
    "Monolithic": (
        MonolithicCoupling(
            transient_systems=models, time_integrator=method, stability=CoupledStep()
        ),
        Numerics(spatial=LumpedParameters()),
    ),
    "Partitioned": (
        PartitionedCoupling(
            transient_systems=models,
            iteration=FixedPoint(
                relative_tolerance=1e-12,
                absolute_tolerance=1e-12,
                maximum_iterations=80,
            ),
        ),
        Numerics(spatial=LumpedParameters(), time_integrator=method),
    ),
}
theme.use_theme()
fig, (ax, balance) = plt.subplots(
    2, 1, sharex=True, figsize=(6.8, 4.4), layout="constrained"
)
for label, (strategy, numerical) in options.items():
    sim = Simulation(
        problem=problem,
        mesh={m: mesh for m in models},
        numerics={m: numerical for m in models},
        coupling_solver=strategy,
    )
```

```
    result = sim.run(start_time=0, end_time=0.6, time_step=0.01)
    t = np.asarray(result.times)
    Ta = np.array([result.field(a.temperature).at(float(x)).values.item() for x in t])
    Tb = np.array([result.field(b.temperature).at(float(x)).values.item() for x in t])
    drift = 2 * Ta + 3 * Tb - 1540.0
    assert np.max(np.abs(drift)) < (1e-9 if label == "Monolithic" else 0.01)
    ax.plot(t, Ta - Tb, label=label)
    balance.plot(t, drift, label=label)
ax.set_ylabel("Temperature difference [K]")
balance.set(xlabel="Time [s]", ylabel="Energy balance error [J]")
ax.legend(frameon=False)
plt.show()
```



CHAPTER 10

Executable numerical recipes

Run each listing in this chapter in its own notebook cell. Make the theme module available as described in the setup chapter before running a plot. Each listing checks at least one quantity, such as a conserved integral, a physical bound, a reference profile or an execution record. The selection index in Chapter 13 links every numerical export to an example here or to an earlier notebook.

10.1 Declarations and execution contracts

10.1.1 Inspect the IR, plan and compatibility error

Compile a concrete lumped model and inspect its IR. Run the model, then restore its resolved provider plan from the result metadata. The sample compatibility rejection carries the option and required capability as machine-readable fields.

This listing exercises `NumericalIR`, `NumericalExecutionPlan`, `NumericalCompatibilityError`.

```
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics.discretization import (
    LumpedParameters, Numerics, NumericalCompatibilityError,
    NumericalExecutionPlan, NumericalIR,
)
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.systems import ThermalCapacitance
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
model = ThermalCapacitance(
    domain=domain.system, heat_capacity=2, conductance=1,
    ambient_temperature=293,
)
problem = Problem(
    name="Plan audit", geometry=domain, physics=(model,),
    initial_conditions=(InitialValue(model.temperature, 313),),
)
simulation = Simulation(
    problem=problem, mesh=LumpedMesh(domain),
    numerics=Numerics(
        spatial=LumpedParameters(), time_integrator=ClassicalRungeKutta4()
    ),
)
ir = simulation.compile().numerical_ir(model)
assert isinstance(ir, NumericalIR)
print(ir.to_dict().keys())

result = simulation.run(start_time=0, end_time=0.2, time_step=0.02)
```

```
audit = result.result_set.metadata["numerical_plans"][model.model_id]
record = NumericalExecutionPlan.from_dict(audit["execution_plan"])
assert record.ir_content_sha256 == NumericalIR.from_dict(audit["numerical_ir"]).content
    _sha256
for requirement in record.stage_requirement_ids:
    print(requirement, record.provider_for(requirement))
restored = NumericalExecutionPlan.from_dict(record.to_dict())
assert restored.content_sha256 == record.content_sha256

def check_backend(requested, available):
    if requested not in available:
        raise NumericalCompatibilityError(
            f"Backend {requested} is unavailable",
            kind="unimplemented", option=requested,
            required_capabilities=("scalar:real",),
        )

try:
    check_backend("accelerator", {"cpu"})
except NumericalCompatibilityError as error:
    assert error.option == "accelerator"
    print(error.kind, error.required_capabilities)
```

10.1.2 Call optional operator protocols

A provider can implement these protocols structurally. This small two-cell operator supplies time-dependent conditions, identifies facet ownership and invalidates a cached evaluation after a coefficient update. A notebook can use the same typed function with a provider that implements these methods.

This listing exercises `OperatorConditions`, `OperatorFacetTopology`, `EvaluationCacheControl`.

```
from onesim.numerics.discretization import (
    EvaluationCacheControl, OperatorConditions, OperatorFacetTopology,
)

class TwoCellOperator:
    def __init__(self):
        self.conductivity = 2.0
        self._cache = {}

    def conditions(self, time):
        return (("left", 300 + 10 * time), ("right", 290.0))

    def facet_owners(self, facet):
        return ((0,), (0, 1), (1,))[facet]

    def invalidate_evaluation_cache(self):
        self._cache.clear()

    def heat_flux(self, time):
```

```
        if time not in self._cache:
            left, right = (value for _, value in self.conditions(time))
            self._cache[time] = self.conductivity * (left - right)
        return self._cache[time]

def report(conditions: OperatorConditions,
           topology: OperatorFacetTopology,
           cache: EvaluationCacheControl, time: float):
    cache.invalidate_evaluation_cache()
    return conditions.conditions(time), topology.facet_owners(1)

operator = TwoCellOperator()
assert operator.heat_flux(0.5) == 30
operator.conductivity = 4
values, owners = report(operator, operator, operator, 0.5)
assert values[0] == ("left", 305) and owners == (0, 1)
assert operator.heat_flux(0.5) == 60
```

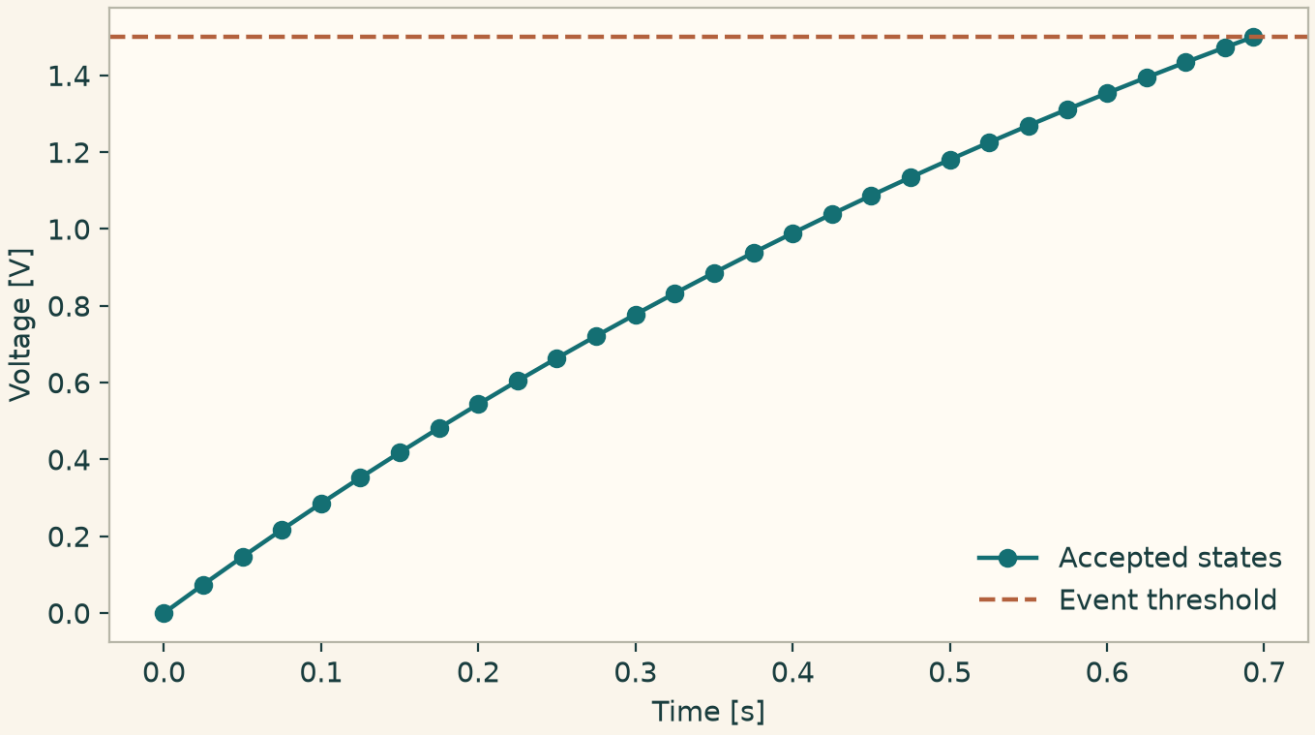
10.1.3 Stop on a custom event and inspect the selected time binding

The callback reads the voltage from the semantic state indexed by model and field identifiers. Run a terminal rising event with an independent cap on the time step. The provider binding reports the actual Runge–Kutta method and order.

This listing exercises `TimeEventDefinition`, `TimeStepLimits`, `TimeExecutionBinding`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics.discretization import LumpedParameters, Numerics
from onesim.numerics.time import (
    ClassicalRungeKutta4, TimeEventDefinition, TimeExecutionBinding, TimeStepLimits,
)
from onesim.numerics.time.selection.declared_time_provider import (
    create_declared_time_provider,
)
from onesim.physics.systems import ResistorCapacitor
from onesim.simulation import ExecutionOptions, Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
rc = ResistorCapacitor(
    domain=domain.system, resistance=2, capacitance=0.5, supply_voltage=3,
)
problem = Problem(
    name="Custom event", geometry=domain, physics=(rc,),
    initial_conditions=(InitialValue(rc.voltage, 0.0),),
)
```

```
def half_charge(time, state):
    value = state[rc.model_id][rc.voltage.field_id]
    return np.asarray(value).item() - 1.5

event = TimeEventDefinition(
    "half-charge", half_charge, direction=1, terminal=True, value_tolerance=1e-10,
)
limits = TimeStepLimits(
    maximum_step=0.025, stability_safety_factor=0.8, cfl_limit=0.4,
)
assert np.isclose(limits.bound(stability_step=0.2, courant_rate=40), 0.01)
method = ClassicalRungeKutta4()
provider = create_declared_time_provider(method)
binding = provider.numerical_execution_binding()
assert isinstance(binding, TimeExecutionBinding)
assert binding.order == 4
print(binding.method_id, binding.order)

result = Simulation(
    problem=problem, mesh=LumpedMesh(domain),
    numerics=Numerics(
        spatial=LumpedParameters(), time_integrator=method,
        step_limits=TimeStepLimits(maximum_step=0.025),
    ),
).run(
    start_time=0, end_time=1.2, time_step=0.025,
    execution_options=ExecutionOptions(events=(event,)),
)
assert abs(result.times[-1] - np.log(2)) < 2e-6
voltage = result.field(rc.voltage)
values = [np.asarray(voltage.at(t).values).item() for t in result.times]
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(result.times, values, "o-", label="Accepted states")
ax.axhline(1.5, linestyle="--", color=theme.COPPER, label="Event threshold")
ax.set(xlabel="Time [s]", ylabel="Voltage [V]")
ax.legend(frameon=False)
plt.show()
```



RC voltage up to the terminal half-charge event.

10.2 Coupled execution and iterative algebra

10.2.1 Solve instantaneous complex feedback

This algebraic state satisfies a linear complex feedback map at one time. The bound closure uses the chosen nonlinear solver and the exact feedback directional derivative. The four sparsity rows cover the real and imaginary coordinates.

This listing exercises `AlgebraicClosure`.

```
import numpy as np
from onesim.numerics.coupled_systems import AlgebraicClosure
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.simulation import SimulationDependencies

closure = AlgebraicClosure(
    nonlinear_solver=Newton(
        linear_solver=SparseDirect(), relative_tolerance=1e-12,
        absolute_tolerance=1e-13,
    ),
).bind(dependencies=SimulationDependencies(), context={})
gain = np.array([[0.1 + 0.2j, 0.2], [0.1j, 0.15 - 0.1j]])
drive = np.array([0.5 + 0.2j, 1.0 - 0.1j])
state = closure.solve(
    np.zeros(2, dtype=complex),
    lambda z: gain @ z + drive,
    lambda z, dz: gain @ dz,
    (tuple(range(4)),) * 4,
)
```

```
np.testing.assert_allclose(state, np.linalg.solve(np.eye(2) - gain, drive))
print(state)
```

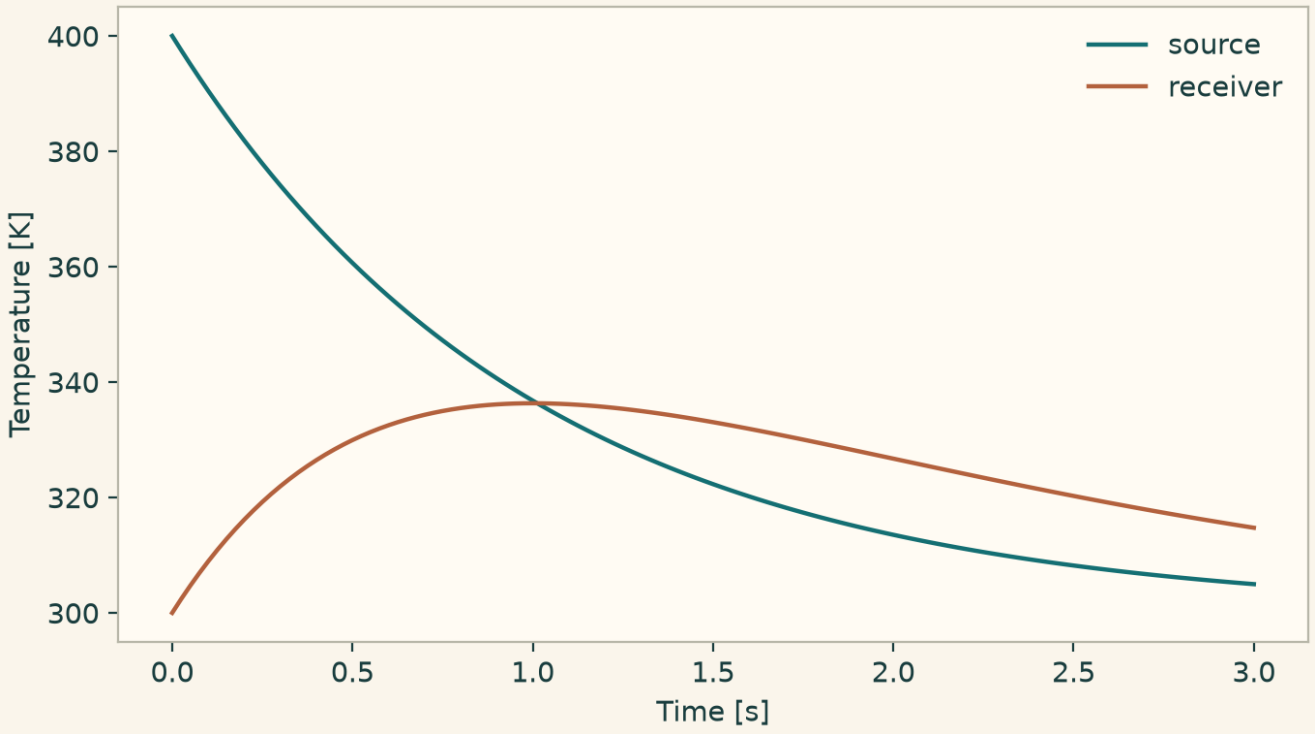
10.2.2 Advance a one-way thermal cascade

The source temperature becomes the receiver ambient temperature at the coupling boundary. The declared order follows that one-way dependency. A cooler receiver initially warms while the source cools.

This listing exercises `SequentialCoupling`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics.discretization import LumpedParameters, Numerics
from onesim.numerics.coupled_systems import SequentialCoupling
from onesim.numerics.field_transfer import GlobalValueTransfer
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.systems import ThermalCapacitance
from onesim.simulation import Problem, Simulation
from onesim.simulation.field_connections import FieldCoupling
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
source = ThermalCapacitance(
    domain=domain.system, model_id="source", heat_capacity=1,
    conductance=1, ambient_temperature=300,
)
receiver = ThermalCapacitance(
    domain=domain.system, model_id="receiver", heat_capacity=1,
    conductance=1, ambient_temperature=None,
)
problem = Problem(
    name="Thermal cascade", geometry=domain, physics=(source, receiver),
    initial_conditions=(
        InitialValue(source.temperature, 400),
        InitialValue(receiver.temperature, 300),
    ),
    couplings=(FieldCoupling(
        source=source.temperature, target=receiver.ambient,
        transfer=GlobalValueTransfer(),
    ),),
)
method = Numerics(
    spatial=LumpedParameters(), time_integrator=ClassicalRungeKutta4()
)
result = Simulation(
    problem=problem, mesh={body: LumpedMesh(domain) for body in (source, receiver)},
    numerics={source: method, receiver: method},
```

```
    coupling_solver=SequentialCoupling(
        transient_systems=(source, receiver), stationary_systems=(),
        order=(source, receiver),
    ),
).run(start_time=0, end_time=3, time_step=0.025)

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for body in (source, receiver):
    field = result.field(body.temperature)
    values = [np.asarray(field.at(t).values).item() for t in result.times]
    ax.plot(result.times, values, label=body.model_id)
assert max(values) > 300
ax.set(xlabel="Time [s]", ylabel="Temperature [K]")
ax.legend(frameon=False)
plt.show()
```



Temperature histories of a sequentially coupled thermal cascade.

10.2.3 Transport a periodic wave with three flux operators

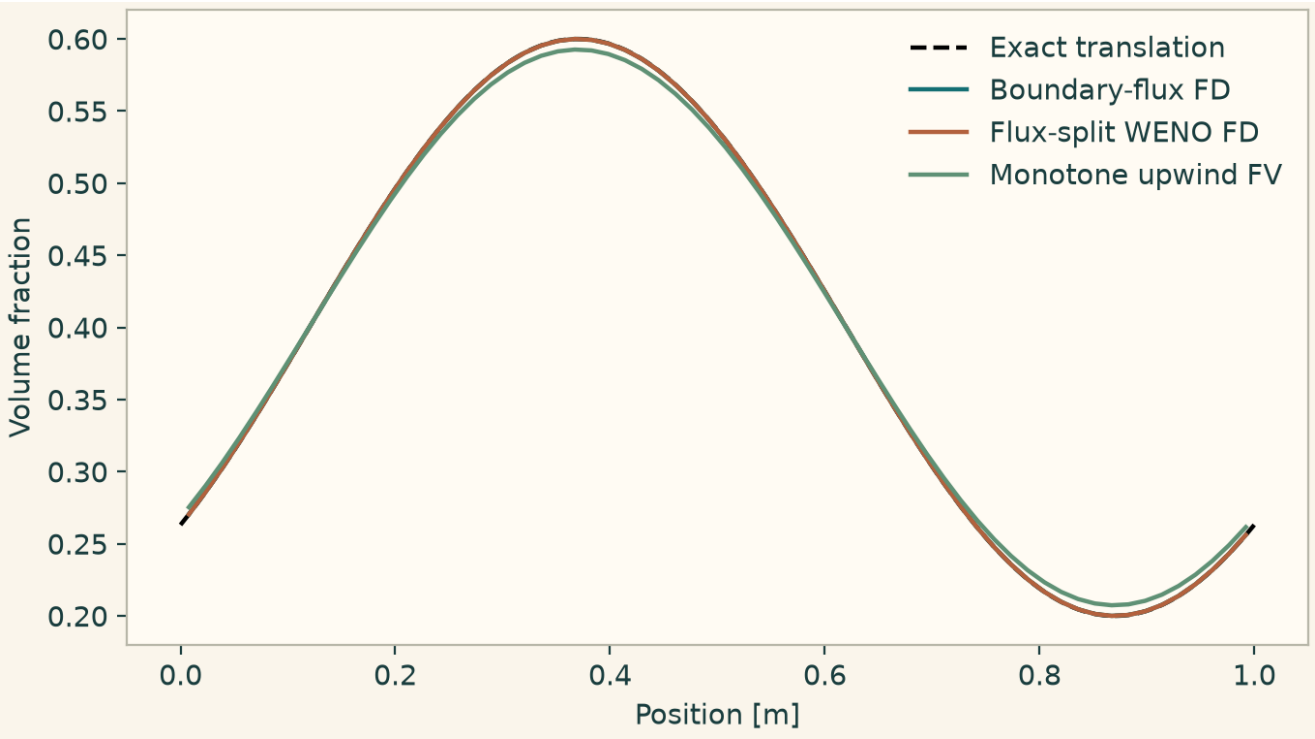
Compare the translated wave with a conservative boundary-flux FD method, flux-split WENO on points, and first-order upwind FV on cell averages. The periodic seam is supplied explicitly. FD WENO reconstructs split fluxes, while the FV reconstructors in a later listing reconstruct cell-average states.

This listing exercises `BoundaryFluxDifference`, `GlobalLaxFriedrichsSplit`, `WENO5Flux`, `WENOFluxDifference`, `MonotoneUpwind`.

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

```
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fdm import (
    BoundaryFluxDifference, FiniteDifference, GlobalLaxFriedrichsSplit,
    PolynomialDifference, WENO5Flux, WENOFluxDifference,
)
from onesim.numerics.fvm import FiniteVolume, MonotoneUpwind
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.hydrodynamics import PhaseAdvection
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

line = Line(start=(0, 0, 0), end=(1, 0, 0))
phase = PhaseAdvection(
    domain=line.line, velocity=1, first_density=1000, second_density=1,
)
wave = lambda x: 0.4 + 0.2 * np.sin(2 * np.pi * x)
problem = Problem(
    name="Periodic flux comparison", geometry=line, physics=(phase,),
    initial_conditions=(InitialValue(phase.volume_fraction, wave),),
    boundary_conditions=(PeriodicBoundary(
        physics=phase, pair=PeriodicPair(line.boundaries.start, line.boundaries.end),
    ),),
)
mesh = StructuredLineMesher(cells=64).generate(line)
methods = {
    "Boundary-flux FD": FiniteDifference(gradient=BoundaryFluxDifference(
        derivative=PolynomialDifference(accuracy_order=4)
    )),
    "Flux-split WENO FD": FiniteDifference(gradient=WENOFluxDifference(
        flux_splitting=GlobalLaxFriedrichsSplit(), reconstruction=WENO5Flux()
    )),
    "Monotone upwind FV": FiniteVolume(convection=MonotoneUpwind()),
}
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
x = np.linspace(0, 1, 250)
ax.plot(x, wave(x - 0.12), "k--", label="Exact translation")
for name, spatial in methods.items():
    result = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(spatial=spatial, time_integrator=ClassicalRungeKutta4()),
    ).run(start_time=0, end_time=0.12, time_step=0.003)
    field = result.field(phase.volume_fraction)
    values = np.asarray(field.values)
    assert np.all(np.isfinite(values))
    ax.plot(field.space.dof_coordinates[:, 0], values, label=name)
ax.set(xlabel="Position [m]", ylabel="Volume fraction")
```

```
ax.legend(frameon=False)
plt.show()
```



Periodic translation with three conservative spatial discretizations.

10.2.4 Balance water depth over a sloping bed

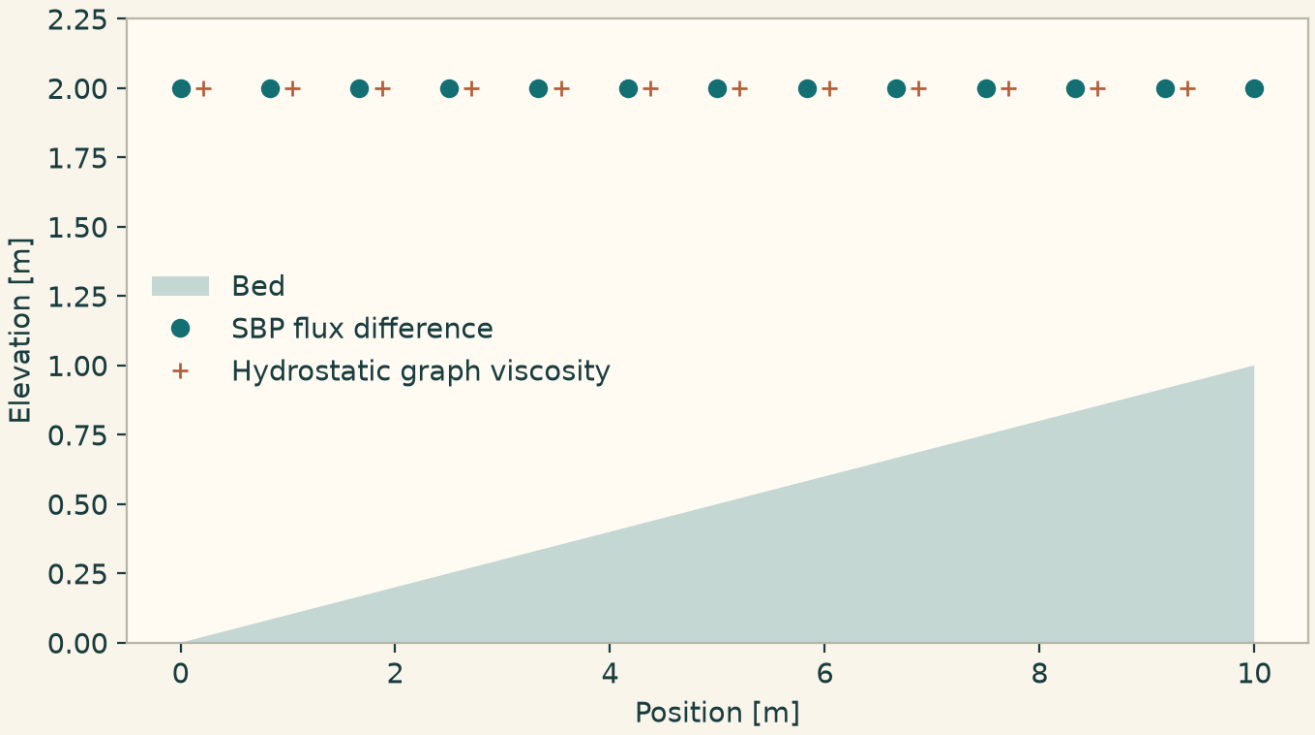
Use the same hydrostatic state in a summation-by-parts finite difference discretization and in a mass-lumped continuous finite element graph discretization. Inspect the surface elevation, which should remain level under the reflecting wall conditions.

This listing exercises `SummationByPartsDifference`, `HydrostaticFluxDifference`, `RowSumLumping`, `HydrostaticGraphViscosity`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import StructuredLineMesher
from onesim.numerics.discretization import NodalInterpolation, Numerics
from onesim.numerics.fdm import (
    FiniteDifference, HydrostaticFluxDifference, SummationByPartsDifference,
)
from onesim.numerics.fem import (
    FiniteElement, HydrostaticGraphViscosity, Lagrange, RowSumLumping,
)
from onesim.numerics.fvm import Rusanov
from onesim.numerics.linear import SparseDirect
from onesim.numerics.time import ForwardEuler, TimeStepLimits
from onesim.physics.hydrodynamics import TopographicShallowWater,
    WetDryShallowWaterDomain
```

```
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import ReflectingWall

line = Line(start=(0, 0, 0), end=(10, 0, 0))
bed = lambda x: 0.1 * x
water = TopographicShallowWater(domain=line.line, bed_elevation=bed)
problem = Problem(
    name="Lake at rest", geometry=line, physics=(water,),
    initial_conditions=(
        InitialValue(water.depth, lambda x: 2.0 - bed(x)),
        InitialValue(water.discharge, 0.0),
    ),
    boundary_conditions=(ReflectingWall(physics=water, boundary=line.boundaries.all),),
)
mesh = StructuredLineMesher(cells=48).generate(line)
domain = WetDryShallowWaterDomain()
methods = {
    "SBP flux difference": FiniteDifference(
        gradient=SummationByPartsDifference(accuracy_order=4),
        provider=HydrostaticFluxDifference(flux=Rusanov(), invariant_domain=domain),
    ),
    "Hydrostatic graph viscosity": FiniteElement(
        element=Lagrange(1), convection=Rusanov(),
        stabilization=HydrostaticGraphViscosity(
            mass=RowSumLumping(), invariant_domain=domain,
        ),
        initialization=NodalInterpolation(),
    ),
}

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
s = np.linspace(0, 10, 160)
ax.fill_between(s, 0, bed(s), alpha=0.25, label="Bed")
for j, (name, spatial) in enumerate(methods.items()):
    result = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(
            spatial=spatial, time_integrator=ForwardEuler(),
            linear_solver=SparseDirect() if isinstance(spatial, FiniteElement) else
            None,
            step_limits=TimeStepLimits(stability_safety_factor=0.8),
        ),
    ).run(start_time=0, end_time=0.02, time_step=0.001)
    field = result.field(water.depth)
    x = field.space.dof_coordinates[:, 0]
    elevation = np.asarray(field.values) + bed(x)
    np.testing.assert_allclose(elevation, 2.0, atol=1e-8)
    ax.plot(x[j::4], elevation[j::4], ("o" if j == 0 else "+"), label=name)
ax.set(xlabel="Position [m]", ylabel="Elevation [m]", ylim=(0, 2.25))
```

```
ax.legend(frameon=False)
plt.show()
```



Hydrostatic equilibrium over a sloping bed using FD and continuous FE.

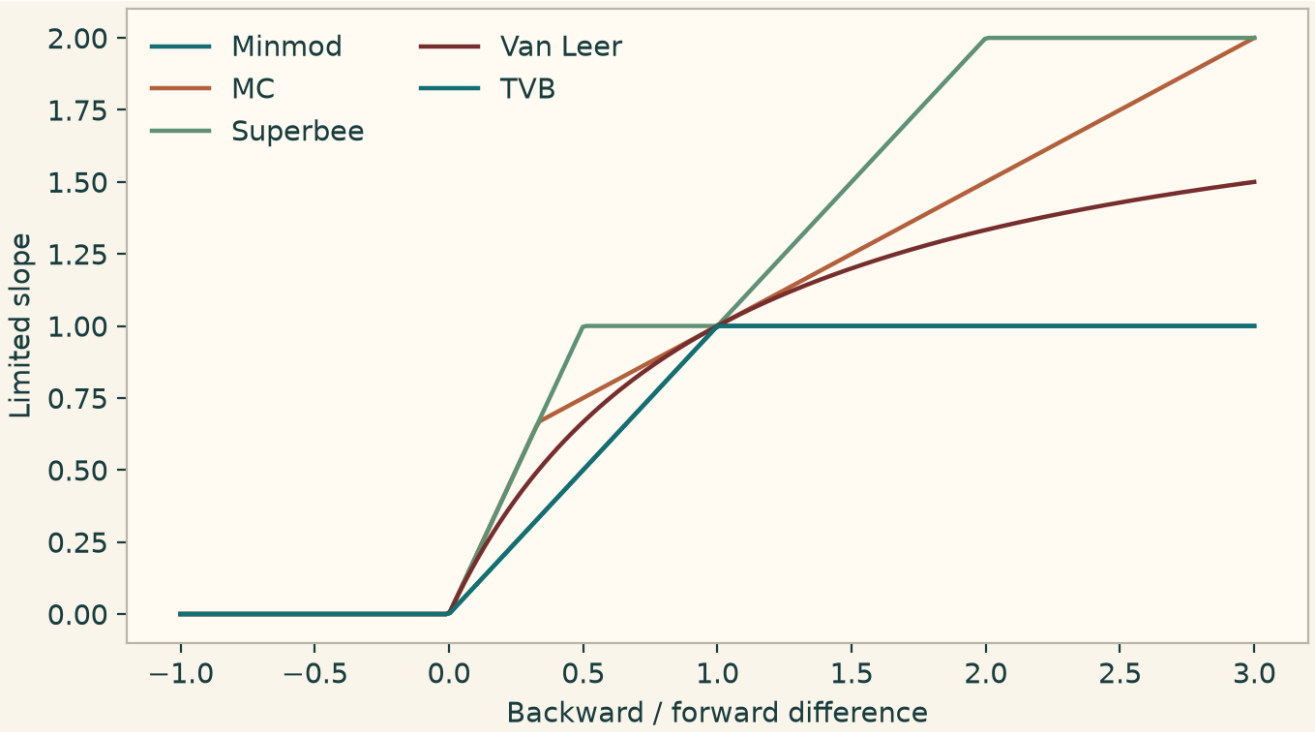
10.2.5 Compare slope limiter responses

Feed the same forward difference and a sweep of backward differences to the scalar limiters. The last two calls limit a cell using its neighbor values and a proposed face increment.

This listing exercises `MonotonizedCentralLimiter`, `SuperbeeLimiter`, `VenkatakrishnanLimiter`, `BarthJespersenLimiter`, `TvbLimiter`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.numerics.fvm import (
    BarthJespersenLimiter, MinmodLimiter, MonotonizedCentralLimiter,
    SuperbeeLimiter, TvbLimiter, VanLeerLimiter, VenkatakrishnanLimiter,
)

ratio = np.linspace(-1.0, 3.0, 300)
limiters = {
    "Minmod": MinmodLimiter(), "MC": MonotonizedCentralLimiter(),
    "Superbee": SuperbeeLimiter(), "Van Leer": VanLeerLimiter(),
    "TVB": TvbLimiter(base_limiter=MinmodLimiter()),
}
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for name, limiter in limiters.items():
    slope = [limiter.limit(r, 1.0) for r in ratio]
```

```
    ax.plot(ratio, slope, label=name)
neighbors = np.array([0.8, 1.3])
for limiter in (BarthJespersenLimiter(), VenkatakrishnanLimiter()):
    factor = limiter.cell_factor(1.0, neighbors, 0.8)
    print(type(limiter).__name__, factor)
    assert 0 <= factor <= 1
ax.set(xlabel="Backward / forward difference", ylabel="Limited slope")
ax.legend(frameon=False, ncol=2)
plt.show()
```



Slope limiter response at a fixed forward difference of one.

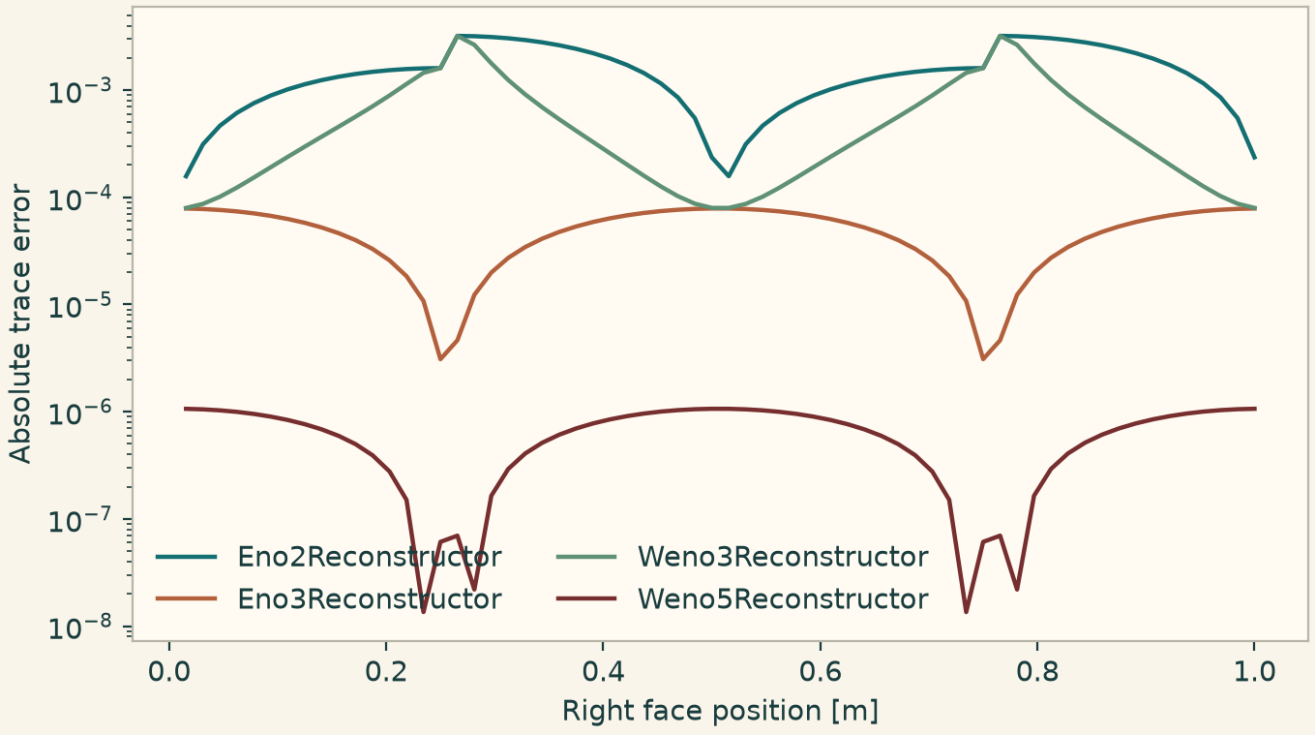
10.2.6 Reconstruct physical face traces with ENO and WENO

Start with exact cell averages of a smooth periodic wave. Compare the reconstructed right trace against the exact point value at each right face. The input has shape one field by number of cells.

This listing exercises `Eno2Reconstructor`, `Eno3Reconstructor`, `Weno3Reconstructor`, `Weno5Reconstructor`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.numerics.fvm import (
    Eno2Reconstructor, Eno3Reconstructor, Weno3Reconstructor, Weno5Reconstructor,
)

faces = np.linspace(0, 1, 65)
width = np.diff(faces)
primitive = lambda x: -np.cos(2 * np.pi * x) / (2 * np.pi)
```

```
mean = (np.diff(primitive(faces)) / width)[None, :]
exact = np.sin(2 * np.pi * faces[1:])

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for choice in (
    Eno2Reconstructor(), Eno3Reconstructor(),
    Weno3Reconstructor(), Weno5Reconstructor(),
):
    reconstruction = choice.bind(cell_widths=width)
    left, right = reconstruction.reconstruct(mean, periodic=True)
    error = np.abs(right[0] - exact)
    assert np.max(error) < 0.02
    ax.semilogy(faces[1:], np.maximum(error, 1e-15), label=type(choice).__name__)
ax.set(xlabel="Right face position [m]", ylabel="Absolute trace error")
ax.legend(frameon=False, ncol=2)
plt.show()
```



Right-trace errors from exact periodic cell averages.

10.2.7 Repair nonphysical face states

A convex guard scales Euler traces towards an admissible cell mean. A separate shallow-water trace policy limits negative depths and enforces the dry-state condition. The examples use component rows and cell columns.

This listing exercises `ConvexInvariantDomain`, `WetDryTraceScaling`.

```
import numpy as np
from onesim.numerics.fvm import ConvexInvariantDomain, WetDryTraceScaling
from onesim.physics.hydrodynamics import WetDryShallowWaterDomain
```

```
from onesim.physics.hydrodynamics.compressible.models.compressible_state import (
    recover_compressible_state,
)

def admissible_gas(values):
    recover_compressible_state(values, (values.shape[1],))

mean = np.array([[1.0, 0.1], [0.0, 0.0], [1.0, 0.01]])
slope = np.array([[2.0, 0.0], [0.0, 0.5], [0.0, 0.0]])
guard = ConvexInvariantDomain().bind(admissible=admissible_gas)
left, right = guard.limit(mean, mean - slope, mean + slope)
for trace in (left, right):
    admissible_gas(trace)
np.testing.assert_allclose((left + right) / 2, mean)

water = WetDryShallowWaterDomain()
guard = WetDryTraceScaling().bind(admissible=water.admissible)
mean = np.array([[1.0, 2.0, 0.0], [0.2, -0.1, 0.0]])
left = np.array([[-0.2, 2.1, 0.0], [0.8, -0.2, 0.0]])
right = np.array([[1.1, -0.3, 0.0], [-0.3, 0.2, 0.0]])
left, right = guard.limit(mean, left, right)
for trace in (left, right):
    water.admissible(trace)
print(left[0], right[0])
```

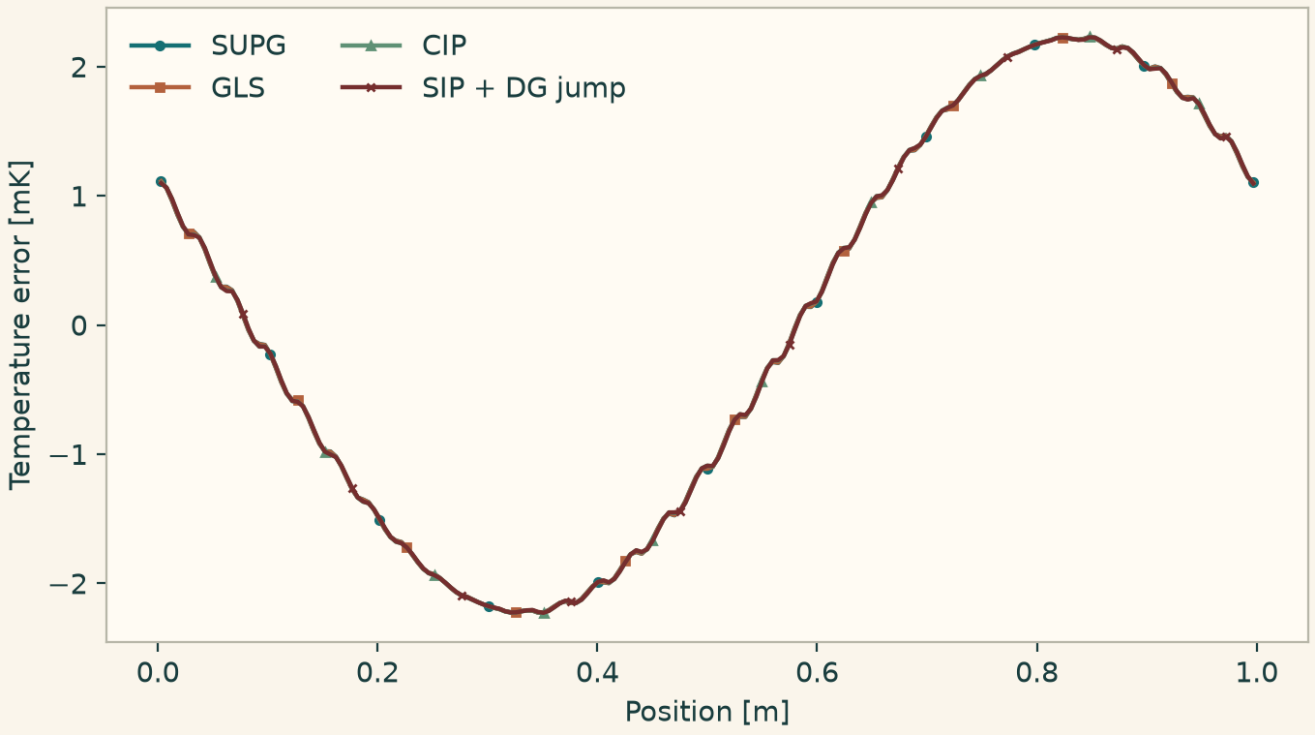
10.2.8 Compare continuous and discontinuous heat operators

Advect and diffuse a periodic temperature wave. Compare continuous residual tests and derivative-jump stabilization with a discontinuous space using symmetric interior penalty diffusion plus an optional solution-jump term. The analytic trace gives a shared reference. Distinct markers identify curves that nearly coincide.

This listing exercises `SUPG`, `GLS`, `CIP`, `DGJumpPenalty`, `SymmetricInteriorPenalty`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import (
    CIP, DGJumpPenalty, DiscontinuousLagrange, FiniteElement,
    GLS, Lagrange, SUPG, SymmetricInteriorPenalty,
)
from onesim.numerics.fvm import Rusanov
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.heat_transfer import HeatAdvectionDiffusion
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
```

```
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

line = Line(start=(0, 0, 0), end=(1, 0, 0))
speed, diffusivity = 0.6, 0.01
heat = HeatAdvectionDiffusion(
    domain=line.line, velocity=speed, thermal_diffusivity=diffusivity,
)
problem = Problem(
    name="Stabilized heat transport", geometry=line, physics=(heat,),
    initial_conditions=(InitialValue(
        heat.temperature, lambda x: 300 + np.sin(2 * np.pi * x)
    ),),
    boundary_conditions=(PeriodicBoundary(
        physics=heat, pair=PeriodicPair(line.boundaries.start, line.boundaries.end)
    ),),
)
mesh = StructuredLineMesher(cells=32).generate(line)
choices = {
    "SUPG": FiniteElement(element=Lagrange(2), stabilization=SUPG(tau=0.003)),
    "GLS": FiniteElement(element=Lagrange(2), stabilization=GLS(tau=0.003)),
    "CIP": FiniteElement(element=Lagrange(2), stabilization=CIP(penalty=0.01)),
    "SIP + DG jump": FiniteElement(
        element=DiscontinuousLagrange(2), convection=Rusanov(),
        diffusion=SymmetricInteriorPenalty(),
        stabilization=DGJumpPenalty(coefficient=0.01),
    ),
}
linear = SparseDirect()
x = np.linspace(0.003, 0.997, 201)
end = 0.08
exact = 300 + np.exp(-4 * np.pi**2 * diffusivity * end) * np.sin(
    2 * np.pi * (x - speed * end)
)
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for j, (label, spatial) in enumerate(choices.items()):
    result = Simulation(
        problem=problem, mesh=mesh, numerics=Numerics(
            spatial=spatial, linear_solver=linear, time_integrator=BackwardEuler(),
            nonlinear_solver=Newton(linear_solver=linear, jacobian="analytic"),
        ),
    ).run(start_time=0, end_time=end, time_step=0.004)
    actual = np.asarray(result.field(heat.temperature).evaluate(
        np.column_stack((x, np.zeros_like(x), np.zeros_like(x)))
    )).ravel()
    assert np.max(abs(actual - exact)) < 0.04
    ax.plot(x, 1000 * (actual - exact), label=label,
            marker=("o", "s", "^", "x")[j], markersize=3, markevery=(j * 5, 20))
ax.set(xlabel="Position [m]", ylabel="Temperature error [mK]")
ax.legend(frameon=False, ncol=2)
plt.show()
```



Final pointwise temperature error for four FE stabilizations.

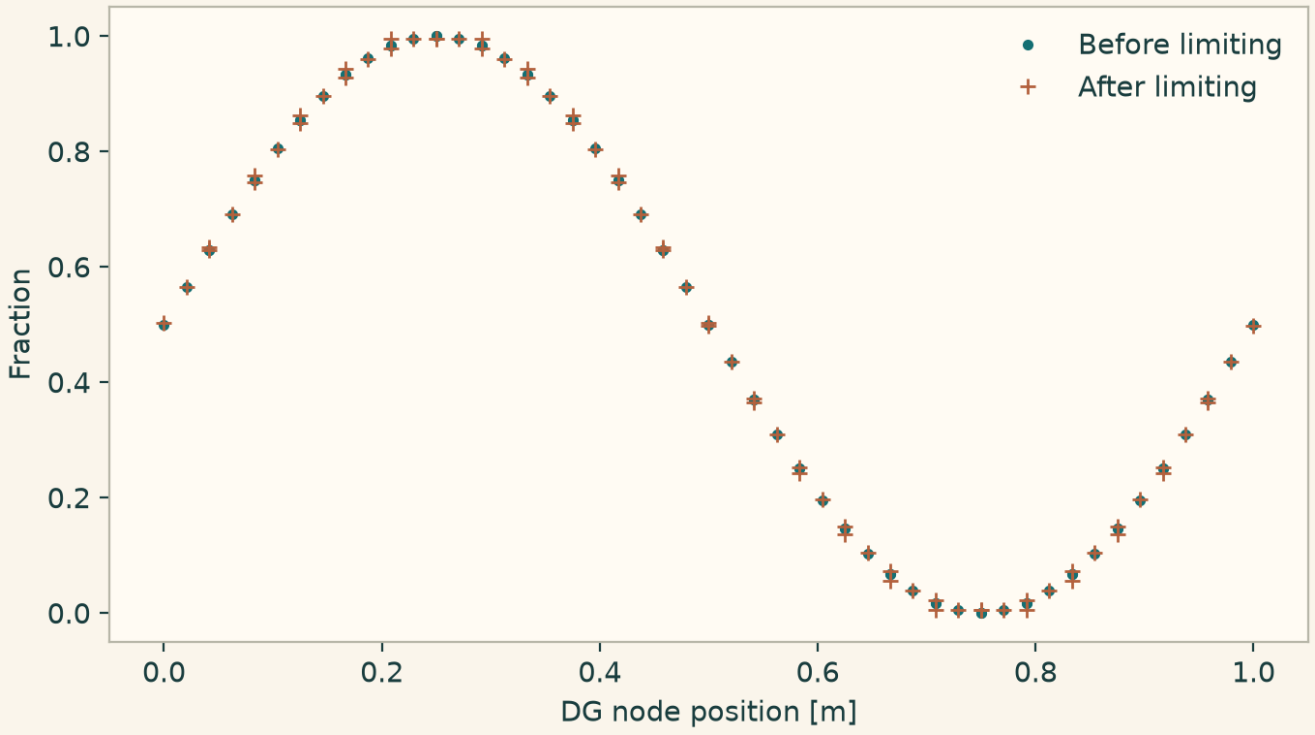
10.2.9 Limit a DG polynomial and check admissibility

Bind a DG limiter on a physical line and call it through the stage limiter protocol. The second part evaluates scalar bounds and two-field separation with mean-centered scaling. Each column is a sample of the same physical state.

This listing exercises `DGPolynomialLimiter`, `StateLimiterPort`, `ScalarInterval`, `OrderedFields`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import StructuredLineMesher
from onesim.numerics.fem import (
    DGPolynomialLimiter, DiscontinuousLagrange, OrderedFields,
    ScalarInterval, StateLimiterPort,
)
from onesim.numerics.fvm import MinmodLimiter

line = Line(start=(0, 0, 0), end=(1, 0, 0))
space = DiscontinuousLagrange(2).bind(
    mesh=StructuredLineMesher(cells=24).generate(line)
)
limiter = DGPolynomialLimiter(slope_limiter=MinmodLimiter()).bind(
    space=space, model=None, law=None, periodic=True
)

def stage(port: StateLimiterPort, values):
    return port.limit(values, time=0.0, stage_id=1)
```

```
x = space.dof_arclengths
original = np.array([0.5 + 0.5 * np.sin(2 * np.pi * x)])
limited = stage(limiter, original)
assert np.max(abs(limited - original)) > 1e-6

interval = ScalarInterval("fraction", 0.0, 1.0)
mean = np.array([0.5])
samples = np.array([[-0.1, 0.9, 0.7]])
theta = interval.maximum_scaling(mean, samples)
interval.admissible(mean[:, None] + theta * (samples - mean[:, None]))

order = OrderedFields(lower="cold", upper="warm", minimum_separation=0.25)
mean = np.array([0.4, 0.9])
samples = np.array([[0.4, 0.6], [0.8, 0.7]])
theta = order.maximum_scaling(mean, samples)
order.admissible(mean[:, None] + theta * (samples - mean[:, None]))
assert 0 < theta < 1

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(x, original[0], ".", label="Before limiting")
ax.plot(x, limited[0], "+", label="After limiting")
ax.set(xlabel="DG node position [m]", ylabel="Fraction")
ax.legend(frameon=False)
plt.show()
```



Changes to the DG nodal values after the stage limiter.

10.2.10 Estimate an explicit stability bound from mass and stiffness

Expose the weak mass and stiffness blocks through a small rate fixture. Compare the generalized eigenvalue bound for symmetric diffusion with a contraction bound for a dissipative nonsymmetric operator. The bound is for the frozen Forward Euler update.

This listing exercises `ConsistentMassStability`, `DissipativeMassStability`.

```
from types import SimpleNamespace
import numpy as np
from scipy.sparse import csr_matrix
from onesim.numerics.fem import ConsistentMassStability, DissipativeMassStability
from onesim.numerics.time import ForwardEuler

def weak_block(values):
    matrix = csr_matrix(values)
    return SimpleNamespace(matrix=SimpleNamespace(
        values=matrix.data, shape=matrix.shape, tocsr=lambda: matrix,
        pattern=SimpleNamespace(indices=matrix.indices, indptr=matrix.indptr),
    ))

mass = np.diag([2.0, 2.0])
rules = (
    (ConsistentMassStability(), np.array([[4.0, -1.0], [-1.0, 4.0]])),
    (DissipativeMassStability(), np.array([[4.0, 1.0], [-1.0, 4.0]])),
)
for rule, stiffness in rules:
    rate = SimpleNamespace(
        operator=object(), problem_size=2,
        weak=lambda t: (
            SimpleNamespace(mass=weak_block(mass)),
            weak_block(stiffness), np.zeros(2), (),
        ),
    )
    dt = rule.maximum_step(
        rate=rate, state=np.zeros(2), time=0.0, declaration=ForwardEuler()
    )
    update = np.eye(2) - dt * np.linalg.solve(mass, stiffness)
    assert np.linalg.norm(update, 2) <= 1 + 1e-8
    print(type(rule).__name__, dt)
```

10.2.11 Evaluate a volume flux divergence and a facet penalty

Bind the flux interpolator to a real nodal hexahedral space. It returns the physical gradients needed to differentiate local Cartesian flux samples. The facet calculation uses a unit quadrature weight and a normal-gradient jump to show the sign and scaling of the CIP block.

This listing exercises `VolumeCIP`, `InterpolatedFluxDivergence`.

```
import numpy as np
from onesim.cad.d3 import Box
from onesim.meshing import StructuredHexahedralMesher
```

```
from onesim.numerics.fem import (
    FiniteElement, InterpolatedFluxDivergence, Lagrange, VolumeCIP,
)

box = Box(length=1, width=1, height=1)
mesh = StructuredHexahedralMesher(cells=(2, 2, 2)).generate(box)
space = FiniteElement(element=Lagrange(1)).bind_space(mesh)

flux = InterpolatedFluxDivergence().bind(space=space)
_, weights, _, _, nodes, gradients = flux.element(0)
samples = nodes * np.array([2.0, -1.0, 3.0])
divergence = np.einsum("qia,ia->q", gradients, samples)
np.testing.assert_allclose(divergence, 4.0, atol=1e-12)

cip = VolumeCIP(diffusivity=0.1).bind(space=space, balance=None)
g = np.array([[[-1.0, 0, 0], [1.0, 0, 0]]])
penalty = cip.volume_facet_contributions(
    field="temperature", left_cell=0, right_cell=1,
    points=np.array([[0.5, 0.25, 0.25]]), weights=np.array([1.0]),
    normals=np.array([[1.0, 0, 0]]), left_gradient=g, right_gradient=g,
    width=0.5, time=0.0,
)
assert np.linalg.eigvalsh(penalty)[-1] > 0
np.testing.assert_allclose(penalty @ np.ones(4), 0, atol=1e-14)
print(weights.sum(), penalty)
```

10.2.12 Stabilize a mixed velocity and pressure space

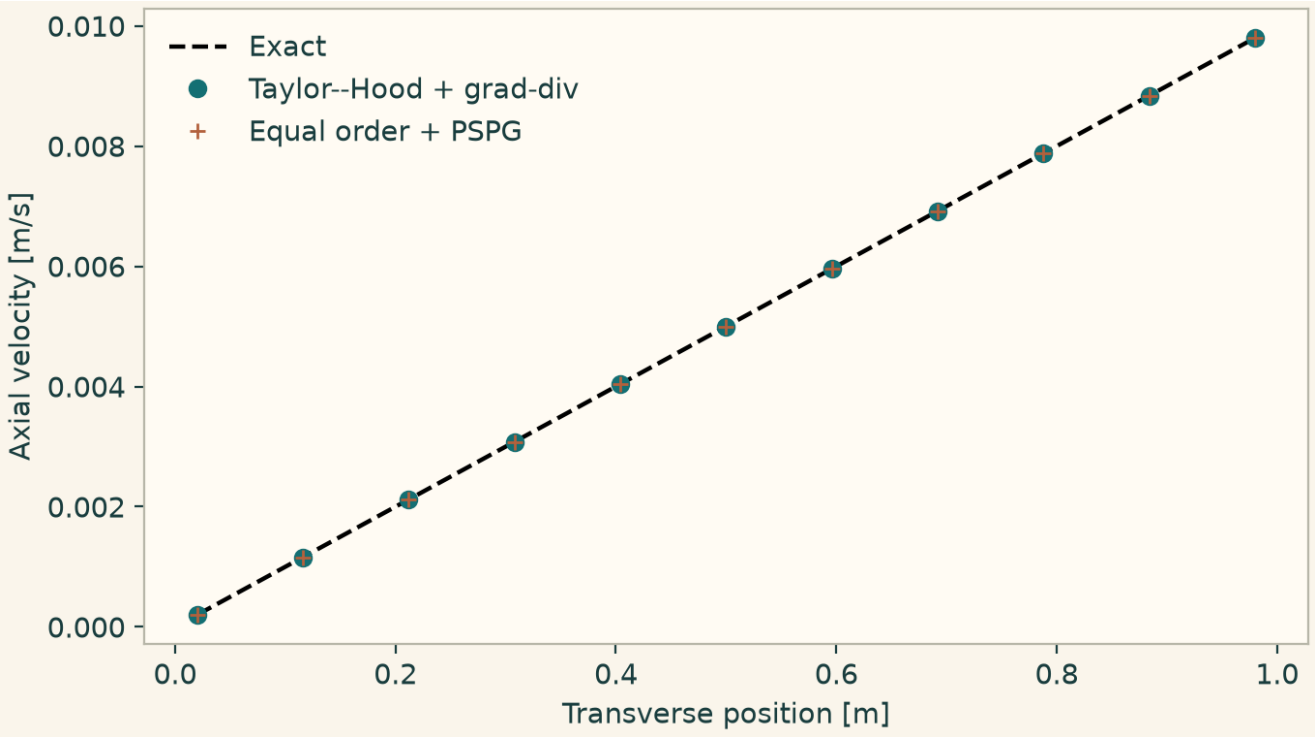
Compare Taylor–Hood with grad-div and equal-order elements with PSPG on accelerating Couette flow. The two contribution classes describe local residual blocks from the bound providers. The plot overlays the exact velocity slice and the computed fields.

This listing exercises `GradDiv`, `MixedMomentumContribution`, `PSPG`, `MixedContinuityContribution`, `EqualOrderVelocityPressure`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import NewtonianFluid
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import (
    EqualOrderVelocityPressure, FiniteElement, GradDiv,
    MixedContinuityContribution, MixedMomentumContribution, PSPG, TaylorHood,
)
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.hydrodynamics import FixedVelocity, IncompressibleNavierStokes
from onesim.simulation import Problem, Simulation
```

```
from onesim.simulation.field_constraints import MeanValue
from onesim.simulation.initial_conditions import InitialValue

box = Box(length=1, width=1, height=1)
def acceleration(y):
    return np.column_stack((y, np.zeros_like(y), np.zeros_like(y)))

def wall(y, t):
    return t * acceleration(y)

wall.time_derivative = lambda y, t: acceleration(y)
flow = IncompressibleNavierStokes(
    domain=box.volume, material=NewtonianFluid(density=1, dynamic_viscosity=0.7),
    body_force=acceleration,
)
problem = Problem(
    name="Accelerating Couette flow", geometry=box, physics=(flow,),
    boundary_conditions=(FixedVelocity(
        physics=flow, boundary=box.boundaries.all, value=wall,
    ),),
    constraints=(MeanValue(flow.pressure, 2.0),),
    initial_conditions=(InitialValue(flow.velocity, (0, 0, 0)),),
)
mesh = StructuredHexahedralMesher(cells=(2, 2, 2)).generate(box)
methods = (
    ("Taylor--Hood + grad-div", TaylorHood(), GradDiv(penalty=0.4)),
    ("Equal order + PSPG", EqualOrderVelocityPressure(order=1), PSPG(time_scale=0.02)),
)
linear = SparseDirect()
end = 0.01
y = np.linspace(0.02, 0.98, 51)
points = np.column_stack((np.full_like(y, 0.44), y, np.full_like(y, 0.58)))
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(y, end * y, "k--", label="Exact")
for label, element, stabilization in methods:
    result = Simulation(
        problem=problem, mesh=mesh, numerics=Numerics(
            spatial=FiniteElement(element=element, stabilization=stabilization),
            linear_solver=linear, time_integrator=BackwardEuler(),
            nonlinear_solver=Newton(linear_solver=linear, jacobian="analytic"),
        ),
    ).run(start_time=0, end_time=end, time_step=0.005)
    actual = result.field(flow.velocity).evaluate(points)[:, 0]
    np.testing.assert_allclose(actual, end * y, atol=1e-8)
    ax.plot(y[::5], actual[::5], "o" if isinstance(stabilization, GradDiv) else "+",
            label=label)

def local_shapes(momentum: MixedMomentumContribution,
                 continuity: MixedContinuityContribution):
    return momentum.momentum.shape, continuity.pressure.shape
```

```
assert local_shapes(
    MixedMomentumContribution(momentum=np.eye(3)),
    MixedContinuityContribution(pressure=np.array([[0.01]])),
) == ((3, 3), (1, 1))
ax.set(xlabel="Transverse position [m]", ylabel="Axial velocity [m/s]")
ax.legend(frameon=False)
plt.show()
```



Mixed FE velocity slices at the end of the accelerating Couette run.

10.2.13 Compare pressure correction and a coupled FV solve

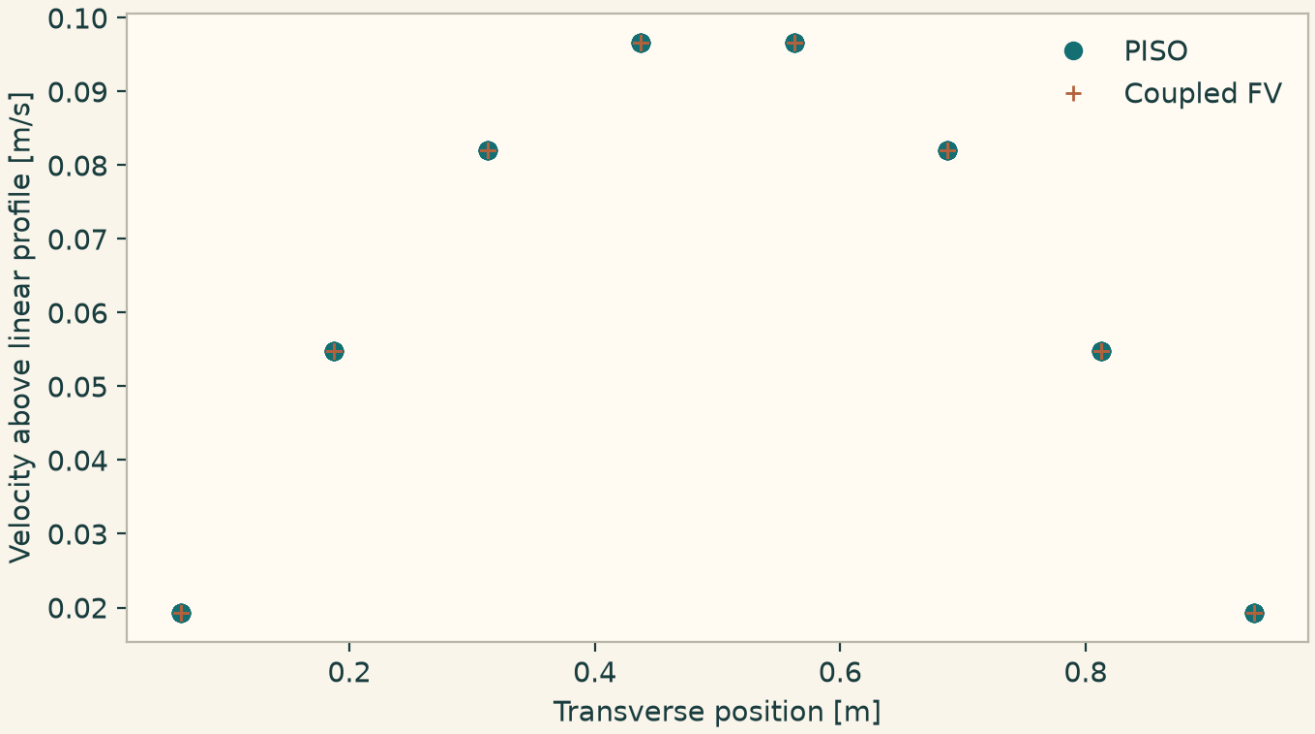
For an initially perturbed Couette profile, use Rhie–Chow face fluxes with two different velocity-pressure algorithms. PISO corrects pressure and face flux in a loop. The coupled algorithm solves a larger stage with the same physical constraints. Compare the final cell values.

This listing exercises `PISO`, `CoupledVelocityPressure`, `RhieChow`, `AlgebraicMultigrid`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import NewtonianFluid
from onesim.meshing import PeriodicPair, StructuredHexahedralMesher
from onesim.numerics.coupled_systems import CoupledVelocityPressure, PISO
from onesim.numerics.discretization import Numerics
from onesim.numerics.fvm import (
    CentralDifference, FiniteVolume, LeastSquaresGradient, LinearUpwind, RhieChow,
)
```

```
from onesim.numerics.linear import AlgebraicMultigrid, ConjugateGradient, SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.hydrodynamics import FixedVelocity, IncompressibleNavierStokes
from onesim.simulation import Problem, Simulation
from onesim.simulation.field_constraints import MeanValue
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

box = Box(length=1, width=1, height=1)
model = IncompressibleNavierStokes(
    domain=box.volume, material=NewtonianFluid(density=1, dynamic_viscosity=0.02),
)
def initial(y):
    return np.column_stack((y + 0.1 * np.sin(np.pi * y), 0 * y, 0 * y))

periods = (
    PeriodicPair(box.boundaries.x_min, box.boundaries.x_max),
    PeriodicPair(box.boundaries.z_min, box.boundaries.z_max),
)
problem = Problem(
    name="FV Couette flow", geometry=box, physics=(model,),
    boundary_conditions=(
        FixedVelocity(physics=model, boundary=box.boundaries.y_min, value=(0, 0, 0)),
        FixedVelocity(physics=model, boundary=box.boundaries.y_max, value=(1, 0, 0)),
        *(PeriodicBoundary(physics=model, pair=pair) for pair in periods),
    ),
    initial_conditions=(InitialValue(model.velocity, initial),),
    constraints=(MeanValue(model.pressure, 2.0),),
)
mesh = StructuredHexahedralMesher(cells=(3, 8, 3)).generate(box)
spatial = FiniteVolume(
    convection=LinearUpwind(), diffusion=CentralDifference(),
    gradient=LeastSquaresGradient(), velocity_pressure_interpolation=RhieChow(),
    face_quadrature_order=1,
)
direct = SparseDirect()
methods = {
    "PISO": PISO(
        momentum_solver=direct,
        pressure_solver=ConjugateGradient(preconditioner=AlgebraicMultigrid()),
        relative_tolerance=1e-10, absolute_tolerance=1e-12,
    ),
    "Coupled FV": CoupledVelocityPressure(
        momentum_solver=direct, pressure_solver=direct,
        nonlinear_solver=Newton(linear_solver=direct, jacobian="analytic"),
    ),
}
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
solutions = []
```

```
for j, (label, coupling) in enumerate(methods.items()):
    result = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(
            spatial=spatial, coupling=coupling, time_integrator=BackwardEuler(),
        ),
    ).run(start_time=0, end_time=0.04, time_step=0.01)
    field = result.field(model.velocity)
    x = field.space.dof_coordinates[:, 1]
    values = np.asarray(field.values)[:, 0]
    solutions.append(values)
    ax.plot(x, values - x, "o" if j == 0 else "+", label=label)
np.testing.assert_allclose(*solutions, atol=1e-5)
ax.set(xlabel="Transverse position [m]", ylabel="Velocity above linear profile [m/s]")
ax.legend(frameon=False)
plt.show()
```



Cell-centered velocity perturbation after pressure correction or a coupled stage.

10.2.14 Select the full Jacobian-free solver name

Run the same radiation balance from the nonlinear comparison with the full declaration name. The implicit time integrator supplies the nonlinear stage and GMRES solves its Krylov systems.

This listing exercises `JacobianFreeNewtonKrylov`.

```
import numpy as np
from onesim.cad.geometry import LumpedDomain
from onesim.meshing import LumpedMesh
from onesim.numerics import Numerics
from onesim.numerics.discretization import LumpedParameters
```

```
from onesim.numerics.linear import GMRES
from onesim.numerics.nonlinear import JacobianFreeNewtonKrylov
from onesim.numerics.time import BackwardEuler
from onesim.physics.heat_transfer import (
    RadiationCooling,
    ThermalGreyDiffuseRadiationBoundary,
)
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue

domain = LumpedDomain()
model = RadiationCooling(
    domain=domain.system,
    heat_capacity=2,
    radiating_area=0.3,
    radiation=ThermalGreyDiffuseRadiationBoundary(
        condition_id="radiation",
        emissivity=0.5,
        ambient_temperature=1,
        stefan_boltzmann=0.02,
    ),
)
problem = Problem(
    name="Nonlinear cooling stage",
    geometry=domain,
    physics=(model,),
    initial_conditions=(InitialValue(model.temperature, 2.0),),
)

solver = JacobianFreeNewtonKrylov(
    linear_solver=GMRES(), relative_tolerance=1e-11, absolute_tolerance=1e-12
)
result = Simulation(
    problem=problem, mesh=LumpedMesh(domain),
    numerics=Numerics(
        spatial=LumpedParameters(), time_integrator=BackwardEuler(),
        nonlinear_solver=solver,
    ),
).run(start_time=0, end_time=1, time_step=0.05)
temperature = np.asarray(result.field(model.temperature).values).item()
assert 1.0 < temperature < 2.0
print(temperature)
```

10.2.15 Apply block and Schur preconditioners to a saddle system

The last two rows are exact constraints with a zero diagonal block. The block sweep uses independently invertible diagonal subblocks of a shifted preconditioning matrix. The Schur method works on the original saddle matrix and its action agrees with the full solve.

This listing exercises `BlockPreconditioner`, `SchurPreconditioner`.

```
from types import SimpleNamespace
import numpy as np
from scipy.sparse import csr_matrix
from onesim.numerics.linear import BlockPreconditioner, GMRES, SchurPreconditioner,
    SparseDirect
from onesim.numerics.linear.contracts.operators.linear_operator import
    LinearOperatorProperties

a = np.array([[3.0, 0.2], [0.2, 2.0]])
b = np.array([[1.0, 2.0], [2.0, -1.0]])
matrix = np.block([[a, b.T], [b, np.zeros((2, 2))]])
shifted = matrix + np.diag([0.0, 0.0, 1.0, 1.0])
right = np.array([1.0, 2.0, -3.0, 4.0])

def port(values):
    return SimpleNamespace(
        properties=LinearOperatorProperties(
            "saddle", values.shape, "assembled", block_sizes=(2, 2),
        ),
        to_scipy_sparse=lambda: csr_matrix(values),
    )

block = BlockPreconditioner(
    solvers=(SparseDirect(), SparseDirect()), sweep="lower",
).build()
block.setup(port(shifted))
u = np.linalg.solve(a, right[:2])
np.testing.assert_allclose(block.apply(right), np.r_[u, right[2:] - b @ u])

schur = SchurPreconditioner(
    solvers=(SparseDirect(), GMRES(relative_tolerance=1e-13)),
).build()
schur.setup(port(matrix))
np.testing.assert_allclose(matrix @ schur.apply(right), right, atol=1e-12)
print(schur.apply(right))
```

10.2.16 Enforce tangential potential and augment the Coulomb constraint

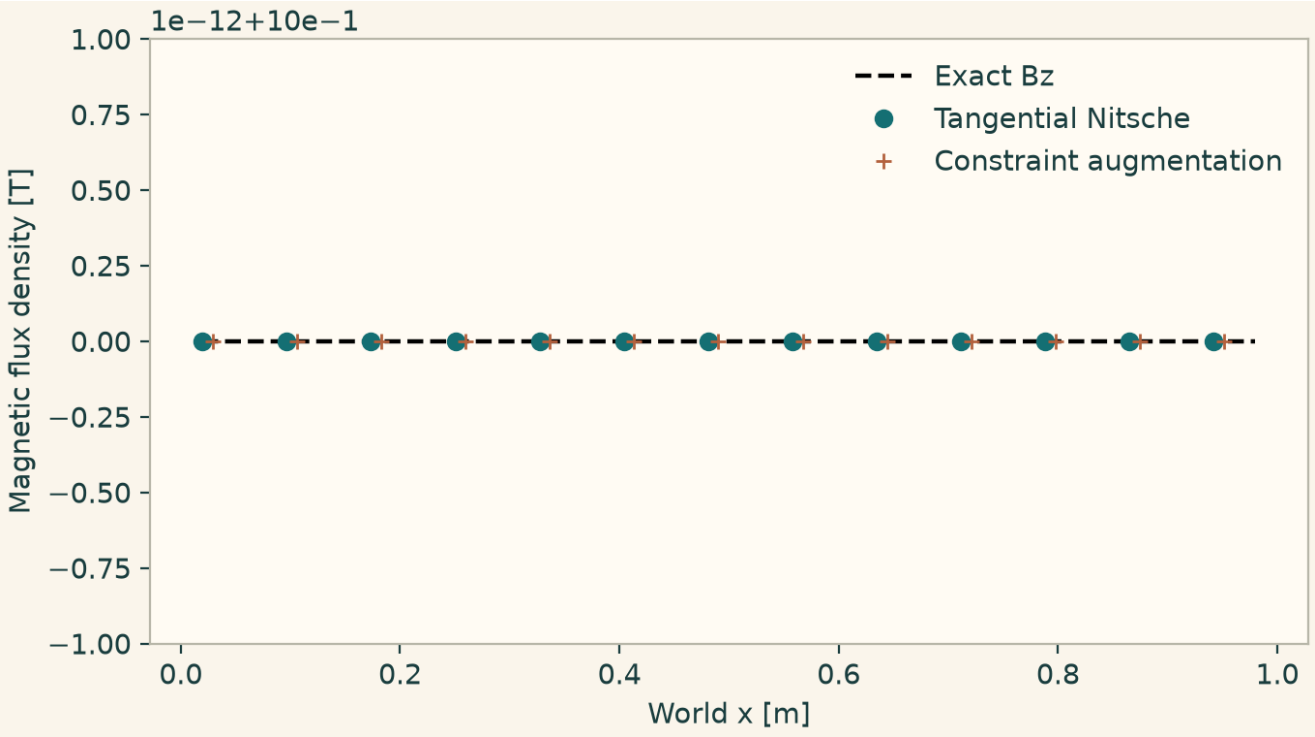
Solve a uniform magnetic patch with a weak tangential potential and compare it with constraint augmentation under strong tangential enforcement. The augmentation changes the primal matrix while retaining the mixed constraint. Check the resulting magnetic flux against the exact curl.

This listing exercises `HcurlNitsche`, `ConstraintAugmentation`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials import VacuumPermeability
from onesim.meshing import StructuredTetrahedralMesher
```

```
from onesim.numerics import Numerics
from onesim.numerics.fem import ConstraintAugmentation, FiniteElement, HcurlNitsche,
    Nedelec
from onesim.numerics.linear import SparseDirect
from onesim.physics.magnetostatics import (
    CoulombGauge,
    MagneticPotentialBoundary,
    MagneticVectorPotential,
    Magnetostatics,
)
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
model = Magnetostatics(
    domain=box.volume,
    permeability=VacuumPermeability(),
    formulation=MagneticVectorPotential(
        gauge=CoulombGauge(multiplier_reference_dof=0)
    ),
)


def vector_potential(x, y, z):
    return np.column_stack((-0.5 * y, 0.5 * x, np.zeros_like(x)))


problem = Problem(
    name="Uniform B patch",
    geometry=box,
    physics=(model,),
    boundary_conditions=(
        MagneticPotentialBoundary(
            physics=model, boundary=box.boundaries.all, value=vector_potential
        ),
    ),
)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)

choices = {
    "Tangential Nitsche": FiniteElement(
        element=Nedelec(1), boundary_enforcement=HcurlNitsche(50)
    ),
    "Constraint augmentation": FiniteElement(
        element=Nedelec(1), stabilization=ConstraintAugmentation(0.7)
    ),
}
x = np.linspace(0.02, 0.98, 101)
points = np.column_stack((x, np.full_like(x, 0.37), np.full_like(x, 0.51)))
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(x, np.ones_like(x), "k--", label="Exact Bz")
```

```
for i, (label, spatial) in enumerate(choices.items()):
    result = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(spatial=spatial, linear_solver=SparseDirect()),
    ).solve_stationary()
    flux = result.field(model.magnetic_flux_density).evaluate(points)
    np.testing.assert_allclose(flux[:, 2], 1.0, atol=1e-8)
    ax.plot(x[i::8], flux[i::8, 2], "o" if i == 0 else "+", label=label)
ax.set(xlabel="World x [m]", ylabel="Magnetic flux density [T]")
ax.legend(frameon=False)
plt.show()
```



Computed axial magnetic flux for two H(curl) boundary and constraint selections.

10.2.17 Recover an affine displacement and Cauchy stress

Impose an affine displacement on a hexahedral block. The boundary-aware least-squares gradient reconstructs the strain and the selected central flux applies the full elastic stress. Affine data provide a direct displacement check at every cell center.

This listing exercises `BoundaryLeastSquaresGradient`, `CentralElasticStress`.

```
import numpy as np
from onesim.cad.d3 import Box
from onesim.materials import IsotropicSolid
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fvm import BoundaryLeastSquaresGradient, CentralElasticStress,
    FiniteVolume
from onesim.numerics.linear import SparseDirect
from onesim.physics.solid_mechanics import FixedDisplacement, LinearElasticity
```

```
from onesim.simulation import Problem, Simulation

box = Box(length=1, width=1, height=1)
material = IsotropicSolid(
    density=1, specific_heat_capacity=1, thermal_conductivity=1,
    youngs_modulus=100, poisson_ratio=0.3, thermal_expansion_coefficient=0,
)
solid = LinearElasticity(domain=box.volume, material=material)
def displacement(x, y, z):
    return np.stack((0.001 * x + 0.0003 * y, -0.0002 * y, 0.0004 * z), axis=-1)

problem = Problem(
    name="FV elasticity patch", geometry=box, physics=(solid,),
    boundary_conditions=(FixedDisplacement(
        physics=solid, boundary=box.boundaries.all, value=displacement,
    ),),
)
result = Simulation(
    problem=problem, mesh=StructuredHexahedralMesher(cells=(4, 4, 4)).generate(box),
    numerics=Numerics(
        spatial=FiniteVolume(
            gradient=BoundaryLeastSquaresGradient(), diffusion=CentralElasticStress(),
        ),
        linear_solver=SparseDirect(),
    ),
).solve_stationary()
field = result.field(solid.displacement)
np.testing.assert_allclose(
    field.values, displacement(*field.space.dof_coordinates.T), atol=1e-11
)
stress = np.asarray(result.field(solid.stress).values)
np.testing.assert_allclose(stress, np.broadcast_to(stress[0], stress.shape),
    atol=1e-10)
print(stress[0])
```

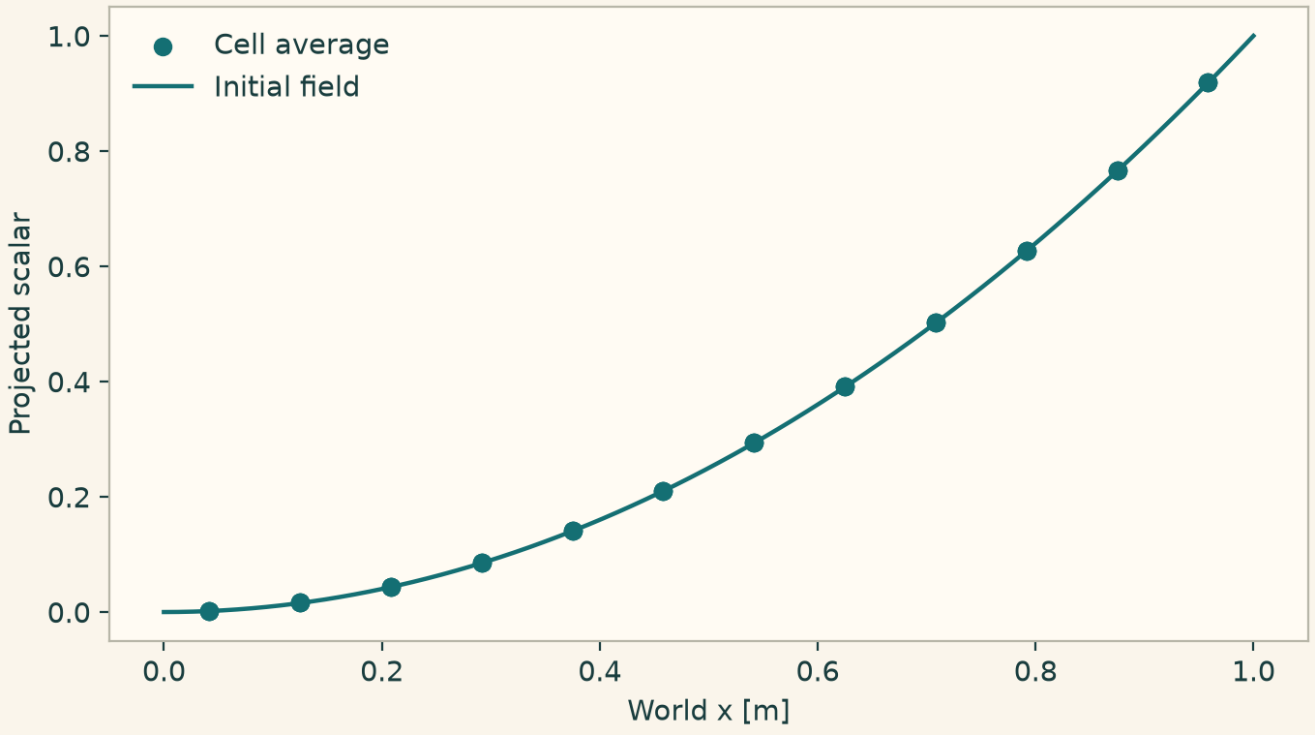
10.2.18 Project a quadratic onto physical cell averages

Project the initial function using volume quadrature. Its cell means differ from the values at the centers for a quadratic. The total discrete integral still matches the exact volume integral.

This listing exercises `CellAverageProjection`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics.discretization import CellAverageProjection
from onesim.numerics.fvm.backends.volume.scalar_diffusion_model import
    PolyhedralCellSpace
from onesim.numerics.fvm.kernels.diffusion.polyhedral_geometry import
```

```
    PolyhedralFiniteVolumeGeometry
from onesim.simulation import FieldDefinition, PhysicsModel
from onesim.simulation.initial_conditions import InitialValue

class Scalar(PhysicsModel):
    FIELD_DEFINITIONS = (FieldDefinition("scalar", "1"),)

box = Box(length=1, width=1, height=1)
mesh = StructuredHexahedralMesher(cells=(12, 2, 2)).generate(box)
space = PolyhedralCellSpace(PolyhedralFiniteVolumeGeometry(mesh), None)
project = CellAverageProjection(quadrature_order=4)
model = Scalar(domain=box.volume)
initial = InitialValue(model.field("scalar"), lambda x: x ** 2)
mean = project.project(initial=initial, space=space, time=0.0)
x = space.dof_coordinates[:, 0]
dx = 1 / 12
np.testing.assert_allclose(mean, x ** 2 + dx ** 2 / 12, atol=1e-13)
np.testing.assert_allclose(np.mean(mean), 1 / 3, atol=1e-13)

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.scatter(x, mean, label="Cell average")
s = np.linspace(0, 1, 150)
ax.plot(s, s ** 2, label="Initial field")
ax.set(xlabel="World x [m]", ylabel="Projected scalar")
ax.legend(frameon=False)
plt.show()
```



Physical cell means compared with the underlying quadratic.

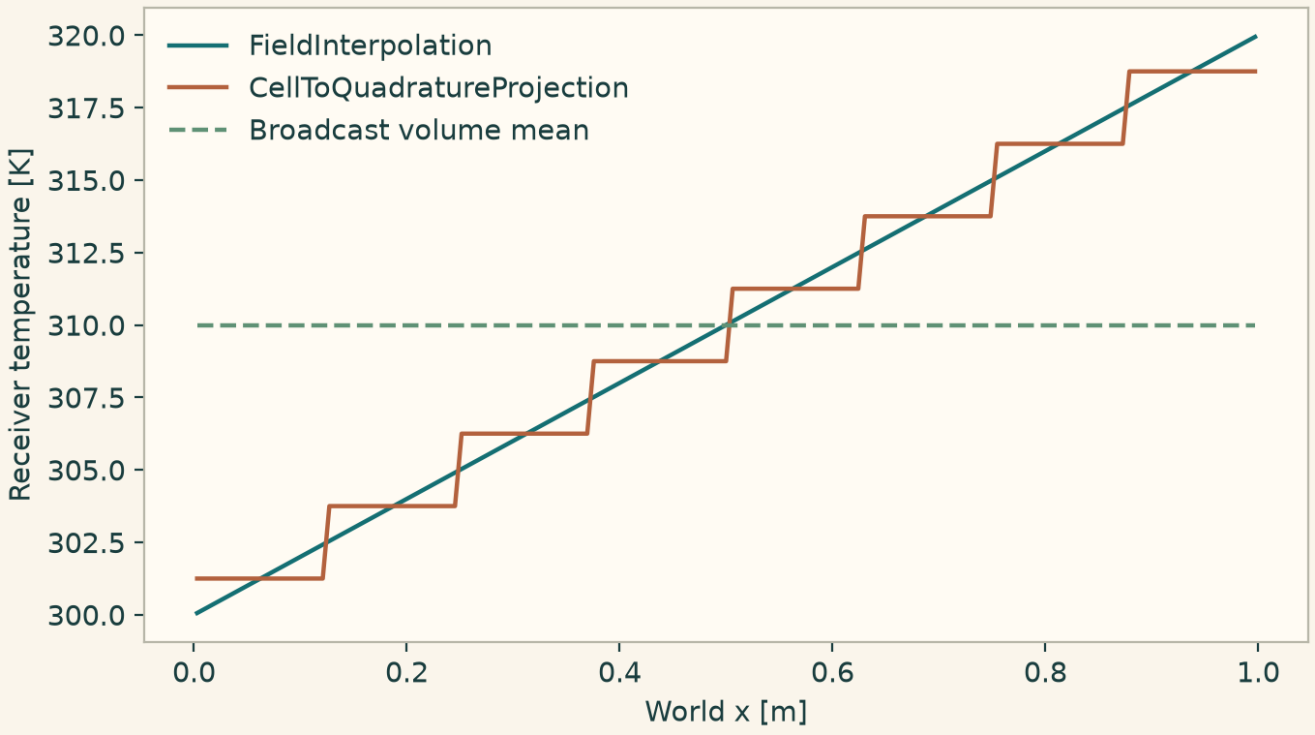
10.2.19 Transfer solved temperatures to receiver points and a global input

Solve the same affine temperature field on FE nodes and FV cells. Evaluate the FE interpolant and the piecewise-constant FV field at receiver points. Average the FE field by physical volume and broadcast the global value uniformly back to spatial coordinates.

This listing exercises `GlobalBroadcastTransfer`, `SpatialAverageTransfer`, `CellToQuadratureProjection`, `FieldInterpolation`.

```
from types import SimpleNamespace
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.cad.geometry import LumpedDomain
from onesim.materials import IsotropicSolid
from onesim.meshing import LumpedMesh, StructuredHexahedralMesher
from onesim.meshing.generation.backends.structured.lumped import GlobalSpace
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.field_transfer import (
    CellToQuadratureProjection, FieldInterpolation, GlobalBroadcastTransfer,
    SpatialAverageTransfer,
)
from onesim.numerics.fvm import CentralDifference, FiniteVolume, LeastSquaresGradient
from onesim.numerics.linear import SparseDirect
from onesim.physics.heat_transfer import FixedTemperature, HeatConduction
from onesim.physics.systems import ThermalCapacitance
from onesim.simulation import FieldDefinition, PhysicsModel, Problem, Simulation

class Receiver(PhysicsModel):
    FIELD_DEFINITIONS = (FieldDefinition("temperature", "K", role="input"),)

box = Box(length=1, width=1, height=1)
material = IsotropicSolid(
    density=1, specific_heat_capacity=1, thermal_conductivity=1,
    youngs_modulus=100, poisson_ratio=0.3, thermal_expansion_coefficient=0,
)
heat = HeatConduction(domain=box.volume, material=material)
problem = Problem(
    name="Affine transfer source", geometry=box, physics=(heat,),
    boundary_conditions=(FixedTemperature(
        physics=heat, boundary=box.boundaries.all, value=lambda x: 300 + 20 * x,
    ),),
)
mesh = StructuredHexahedralMesher(cells=(8, 3, 3)).generate(box)
receiver_mesh = StructuredHexahedralMesher(cells=(9, 2, 2)).generate(box)
target = Receiver(domain=box.volume).field("temperature")
x = np.linspace(0.003, 0.997, 161)
points = np.column_stack((x, np.full_like(x, 0.32), np.full_like(x, 0.41)))

theme.use_theme()
```

```
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
for spatial, transfer in (
    (FiniteElement(element=Lagrange(1)), FieldInterpolation()),
    (FiniteVolume(diffusion=CentralDifference(), gradient=LeastSquaresGradient()),
        CellToQuadratureProjection()),
):
    result = Simulation(
        problem=problem, mesh=mesh,
        numerics=Numerics(spatial=spatial, linear_solver=SparseDirect()),
    ).solve_stationary()
    source = result.field(heat.temperature)
    samples = transfer.bind(
        source=source, target=target, target_mesh=receiver_mesh
    ).evaluate(points)
    ax.plot(x, samples, label=type(transfer).__name__)
    if isinstance(transfer, FieldInterpolation):
        finite_element_temperature = source
        np.testing.assert_allclose(samples, 300 + 20 * x, atol=1e-9)

domain = LumpedDomain()
global_mesh = LumpedMesh(domain)
body = ThermalCapacitance(
    domain=domain.system, heat_capacity=1, conductance=1, ambient_temperature=None,
)
average = SpatialAverageTransfer(quadrature_order=4).bind(
    source=finite_element_temperature, target=body.ambient, target_mesh=global_mesh,
)
mean = np.asarray(average.samples).reshape(-1)
np.testing.assert_allclose(mean, [310.0], atol=1e-9)

global_source = SimpleNamespace(reference=body.temperature,
    space=GlobalSpace(global_mesh))
broadcast = GlobalBroadcastTransfer().receiver_evaluator(
    evaluator=global_source, target_mesh=receiver_mesh
)
uniform = broadcast.restore_samples(mean).evaluate(points)
np.testing.assert_allclose(uniform, 310.0, atol=1e-9)
ax.plot(x, uniform, "--", label="Broadcast volume mean")
ax.set(xlabel="World x [m]", ylabel="Receiver temperature [K]")
ax.legend(frameon=False)
plt.show()
```



Pointwise FE and FV transfer values, with the spatial mean broadcast as a constant.

10.2.20 Transfer an electric field through a periodic FV seam

Solve a linear Poisson potential on a box and evaluate its field on the interior cylindrical receiver. The physical voltage jump over the axial period is supplied explicitly. Check both the vector field and the signed potential difference across the periodic link.

This listing exercises `ElectricFieldToFaces`, `PeriodicPotentialTrace`.

```
import numpy as np
from onesim.cad.d3 import Box, Cylinder
from onesim.materials import VacuumPermittivity
from onesim.meshing import PeriodicPair, StructuredCylindricalMesher,
    StructuredTetrahedralMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange
from onesim.numerics.field_transfer import ElectricFieldToFaces, PeriodicPotentialTrace
from onesim.numerics.fvm.kernels.diffusion.cylindrical_geometry import
    CylindricalFiniteVolumeGeometry
from onesim.numerics.linear import SparseDirect
from onesim.physics.electrostatics import FixedPotential, PoissonEquation
from onesim.simulation import FieldDefinition, PhysicsModel, Problem, Simulation

class Receiver(PhysicsModel):
    FIELD_DEFINITIONS = (
        FieldDefinition("electric_field", "V/m", ("x", "y", "z"), "input"),
    )

box = Box(length=1, width=1, height=1)
mesh = StructuredTetrahedralMesher(cells=(2, 2, 2)).generate(box)
model = PoissonEquation(domain=box.volume, permittivity=VacuumPermittivity())
```

```
source = Simulation(
    problem=Problem(
        name="Affine Poisson potential", geometry=box, physics=(model,),
        boundary_conditions=(FixedPotential(
            physics=model, boundary=box.boundaries.all,
            value=lambda x, y, z: 2 * x - 3 * y + 4 * z,
        ),),
    ),
    mesh=mesh, numerics=Numerics(
        spatial=FiniteElement(element=Lagrange(2)), linear_solver=SparseDirect(),
    ),
).solve_stationary().field(model.electric_field)

tube = Cylinder(radius=0.05, length=0.3, origin=(0.3, 0.3, 0.3))
receiver_mesh = StructuredCylindricalMesher(
    radial_cells=2, azimuthal_cells=8, axial_cells=4,
    periodic_pairs=(PeriodicPair(tube.boundaries.bottom, tube.boundaries.top),),
).generate(tube)
target = Receiver(domain=tube.volume).field("electric_field")
transfer = ElectricFieldToFaces(
    source_mesh=mesh, target_mesh=receiver_mesh,
    periodic_trace=PeriodicPotentialTrace(potential_jump=4 * tube.length),
).bind(source=source, target=target, target_mesh=receiver_mesh)
expected = np.array([-2.0, 3.0, -4.0])
seams = 0
for face in CylindricalFiniteVolumeGeometry(receiver_mesh).faces:
    data = transfer.face_data(face)
    np.testing.assert_allclose(
        data.electric_field, np.broadcast_to(expected, data.electric_field.shape),
        atol=1e-9,
    )
    if face.axis == 2 and face.index[2] == receiver_mesh.shape[2]:
        delta = receiver_mesh.cell_centers[face.right] -
            receiver_mesh.cell_centers[face.left]
        delta += tube.length * np.asarray(tube.axis)
        np.testing.assert_allclose(data.potential_difference, -expected @ delta)
        seams += 1
assert seams == 16
print("Verified periodic faces:", seams)
```

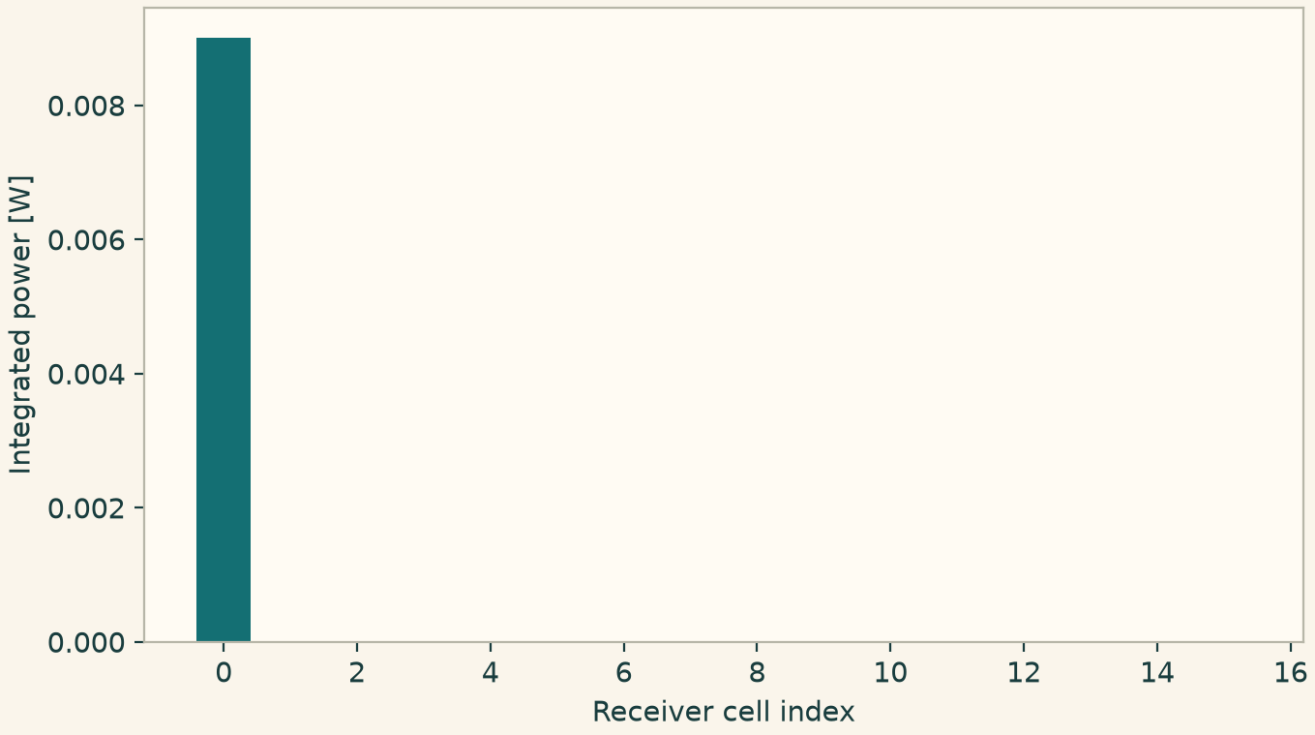
10.2.21 Transfer native FE density conservatively to FV cells

Integrate a constant native FE density over its overlap with the cylindrical FV cells. The receiver uses full cell volumes in the denominator. Multiplying the received means by those volumes recovers the integrated power.

This listing exercises `ConservativeVolumeTransfer`.

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

```
from onesim.cad.d3 import Cylinder
from onesim.meshing import StructuredCylindricalMesher
from onesim.numerics.fem.backends.spaces.tetrahedral.spaces import
    LagrangeTetrahedralSpace
from onesim.numerics.field_transfer import ConservativeVolumeTransfer
from onesim.numerics.field_transfer.evaluation.execution.conservative.conservative_volu
    me_transfer import (
    CylindricalOverlapPlan,
)
from onesim.results.sampling.samples.field_samples import FEStateEvaluator
from onesim.simulation import FieldDefinition, PhysicsModel

class Heating(PhysicsModel):
    FIELD_DEFINITIONS = (FieldDefinition("power", "W/m^3", role="derived"),)

tube = Cylinder(radius=1.0, length=1.0)
mesh = StructuredCylindricalMesher(
    radial_cells=2, azimuthal_cells=4, axial_cells=2,
).generate(tube)
vertices = np.array([
    [0.1, 0.1, 0.1], [0.4, 0.1, 0.1], [0.1, 0.4, 0.1], [0.1, 0.1, 0.4],
])
space = LagrangeTetrahedralSpace(vertices, np.array([[0, 1, 2, 3]]), order=2)
field = Heating(domain=tube.volume).field("power")
source = FEStateEvaluator(field, space)
plan = CylindricalOverlapPlan(ConservativeVolumeTransfer(), source, mesh)
density, evidence = plan.integrate(np.full(space.dof_count, 2.0), layout="owned")
power = np.asarray(density) * mesh.cell_volumes
expected = 2.0 * sum(space.geometry.volumes)
np.testing.assert_allclose(power.sum(), expected, rtol=2e-8)
np.testing.assert_allclose(evidence.whole_source_integral, expected, rtol=2e-8)

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.bar(np.arange(mesh.cell_count), power)
ax.set(xlabel="Receiver cell index", ylabel="Integrated power [W]")
plt.show()
```



Integrated power deposited in the receiving FV cells.

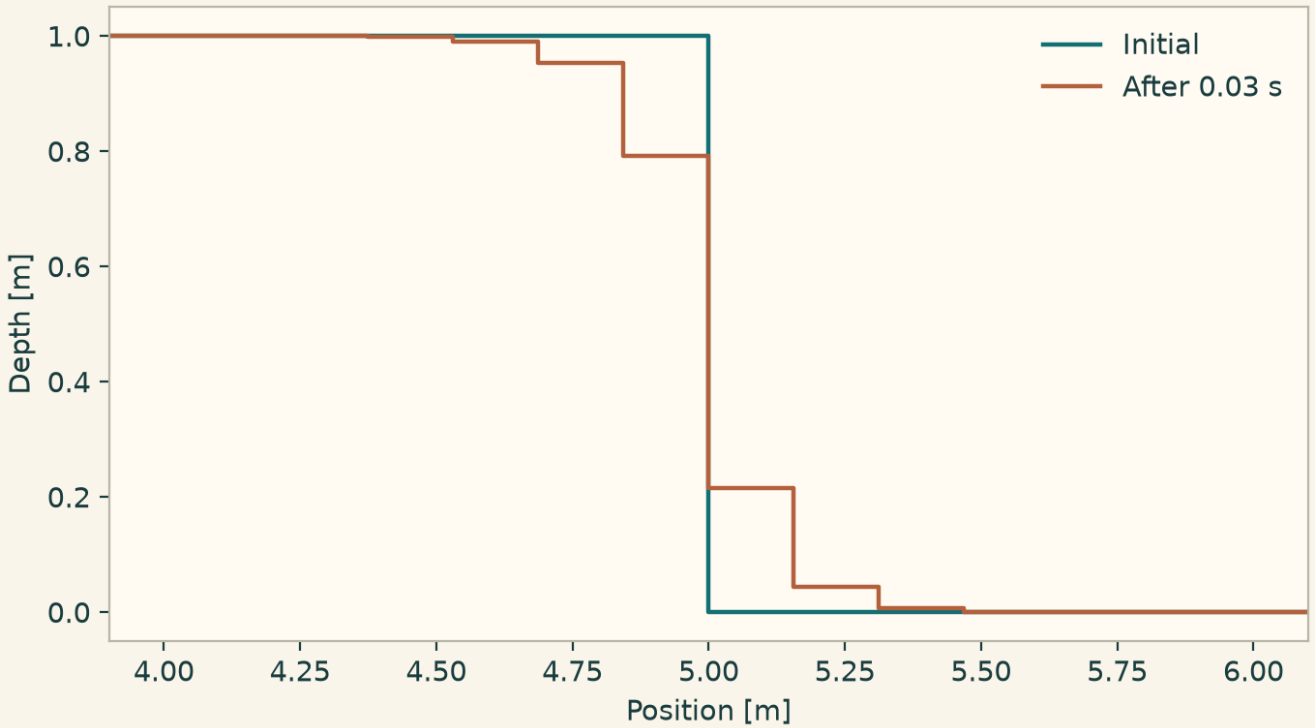
10.2.22 Advance a wet front with an active-set Jacobian

Use the same wet/dry policy in the hydrostatic flux and in Newton. The selected generalized Jacobian releases dry cells when the physical flux brings water into them. Check positivity and integrated water volume at the accepted state.

This listing exercises `WetDryActiveSet`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import StructuredLineMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fvm import FiniteVolume, HydrostaticReconstruction, Rusanov,
    WetDryActiveSet
from onesim.numerics.linear import SparseDirect
from onesim.numerics.nonlinear import Newton
from onesim.numerics.time import BackwardEuler
from onesim.physics.hydrodynamics import TopographicShallowWater
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import ReflectingWall

line = Line(start=(0, 0, 0), end=(10, 0, 0))
water = TopographicShallowWater(domain=line.line, bed_elevation=0.0)
problem = Problem(
    name="Wet front", geometry=line, physics=(water,),
    initial_conditions=(
        InitialValue(water.depth, lambda x: np.where(x < 5, 1.0, 0.0)),
```

```
        InitialValue(water.discharge, 0.0),
    ),
    boundary_conditions=(ReflectingWall(physics=water, boundary=line.boundaries.all),),
)
policy = WetDryActiveSet()
result = Simulation(
    problem=problem, mesh=StructuredLineMesher(cells=64).generate(line),
    numerics=Numerics(
        spatial=FiniteVolume(
            convection=Rusanov(),
            provider=HydrostaticReconstruction(dry_linearization=policy),
        ),
        time_integrator=BackwardEuler(),
        nonlinear_solver=Newton(
            linear_solver=SparseDirect(), jacobian=policy,
            relative_tolerance=1e-10, absolute_tolerance=1e-12,
        ),
    ),
).run(start_time=0, end_time=0.03, time_step=0.003)
h = result.field(water.depth)
np.testing.assert_allclose(h.integrate().value, h.integrate(time=0).value, atol=1e-9)
assert np.min(h.values) >= 0
x = h.space.dof_coordinates[:, 0]
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.step(x, h.at(0).values, where="mid", label="Initial")
ax.step(x, h.values, where="mid", label="After 0.03 s")
ax.set(xlabel="Position [m]", ylabel="Depth [m]", xlim=(3.9, 6.1))
ax.legend(frameon=False)
plt.show()
```



Depth before and after a wet/dry implicit step sequence.

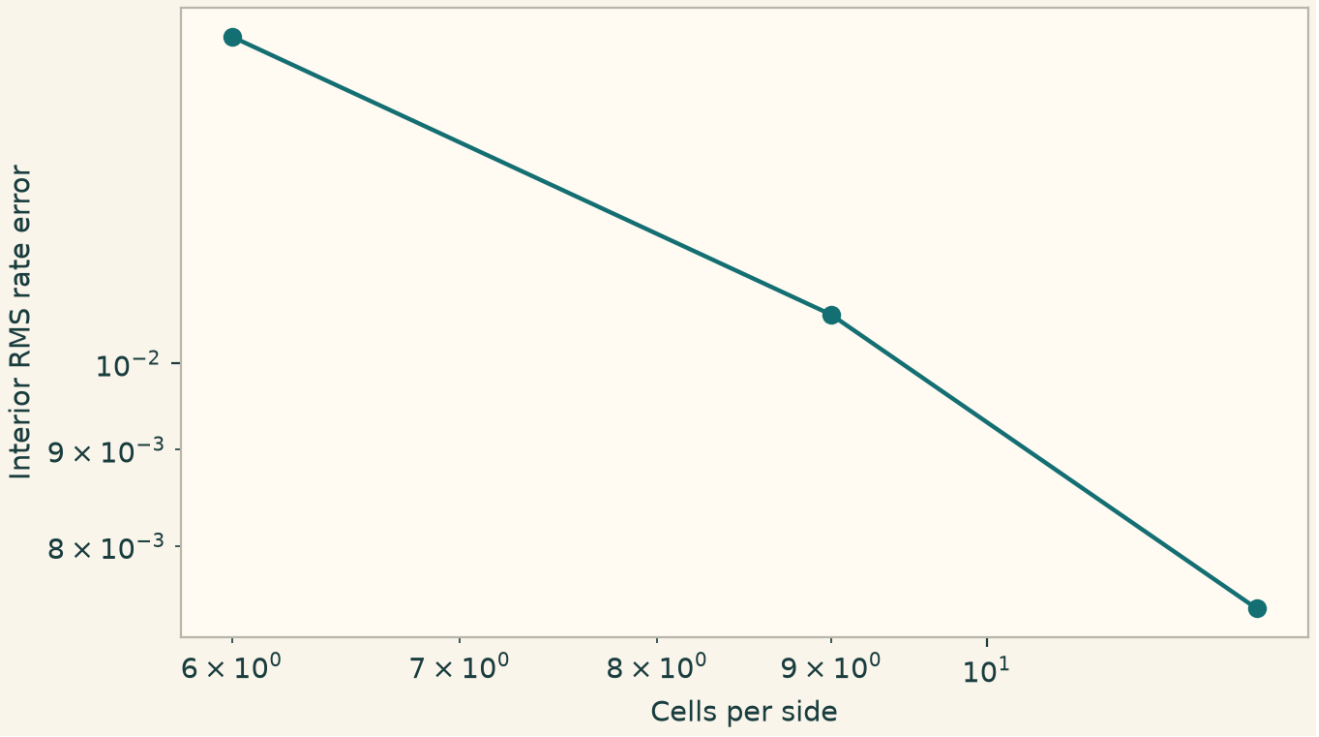
10.2.23 Reconstruct cell-polynomial drift and diffusion fluxes

Use conservative cell-gradient reconstruction in a multidimensional Scharfetter–Gummel flux. Interior cells provide a resolution comparison without the imposed external face laws. The reference differentiates the sinusoidal density analytically.

This listing exercises `CellGradientReconstruction`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics.fvm import (
    BarthJespersenLimiter, CellGradientReconstruction, FiniteVolume, ScharfetterGummel,
)
from onesim.numerics.fvm.backends.volume.plasma_geometry import bind_plasma_geometry

velocity = np.array([0.2, -0.3, 0.4])
diffusion = 0.03
k = 2 * np.pi
sizes = (6, 9, 12)
errors = []
box = Box(length=1, width=1, height=1)
for n in sizes:
    mesh = StructuredHexahedralMesher(cells=(n,) * 3).generate(box)
    g, _, _ = bind_plasma_geometry(mesh, FiniteVolume())
    flux = ScharfetterGummel().bind(
        geometry=g, reconstruction=CellGradientReconstruction(),
        limiter=BarthJespersenLimiter(),
    )
    x, y, z = g.mesh.cell_centers.T
    mode = 0.1 * np.sin(k * x) + 0.07 * np.cos(k * y) + 0.05 * np.sin(k * z)
    derivative = k * (
        velocity[0] * 0.1 * np.cos(k * x)
        - velocity[1] * 0.07 * np.sin(k * y)
        + velocity[2] * 0.05 * np.cos(k * z)
    )
    mean_factor = np.sinc(1 / n)
    values = 2 + mean_factor * mode
    exact = mean_factor * (-derivative - diffusion * k ** 2 * mode)

    face = g.selected(g.internal)
    count = len(face.points)
    speed = tuple(np.full(count, value) for value in velocity)
    tensor = tuple(tuple(np.full(count, value) for value in row)
                   for row in diffusion * np.eye(3))
    integrated_flux = flux.internal_flux(
        values=values, velocity=speed, diffusivity=tensor, face_geometry=face,
    )
    rate = -g.divergence[:, g.internal] @ integrated_flux
```

```
    interior = np.all((g.mesh.cell_centers > 1 / n) &
                      (g.mesh.cell_centers < 1 - 1 / n), axis=1)
    errors.append(np.sqrt(np.mean((rate[interior] - exact[interior]) ** 2)))
assert errors[-1] < 0.55 * errors[0]

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.loglog(sizes, errors, "o-")
ax.set(xlabel="Cells per side", ylabel="Interior RMS rate error")
plt.show()
```



Interior rate error for multidimensional cell-gradient reconstruction.

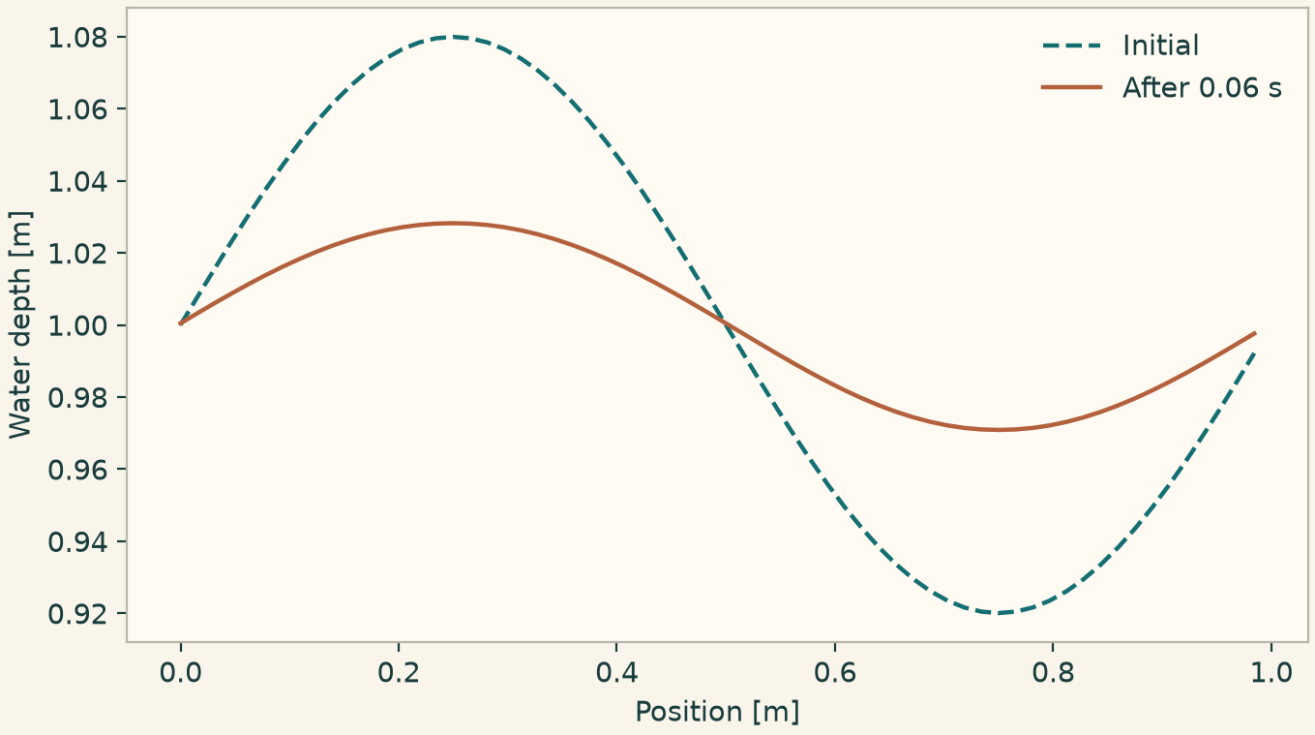
10.2.24 Evolve a positive shallow-water wave on an FE graph

Use a row-sum mass and a selected depth floor for an ordinary flat-bottom shallow-water balance. The graph viscosity adds a conservative edge dissipation. The accepted depth stays positive and its integral stays constant under periodic boundaries.

This listing exercises `InvariantDomainGraphViscosity`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d1 import Line
from onesim.meshing import PeriodicPair, StructuredLineMesher
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import FiniteElement, InvariantDomainGraphViscosity,
    Lagrange, RowSumLumping
from onesim.numerics.fvm import Rusanov
from onesim.numerics.linear import SparseDirect
```

```
from onesim.numerics.time import ClassicalRungeKutta4
from onesim.physics.hydrodynamics import ShallowWater, ShallowWaterInvariantDomain
from onesim.simulation import Problem, Simulation
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

line = Line(start=(0, 0, 0), end=(1, 0, 0))
water = ShallowWater(domain=line.line)
problem = Problem(
    name="Graph shallow water", geometry=line, physics=(water,),
    initial_conditions=(
        InitialValue(water.depth, lambda x: 1 + 0.08 * np.sin(2 * np.pi * x)),
        InitialValue(water.discharge, 0.0),
    ),
    boundary_conditions=(PeriodicBoundary(
        physics=water, pair=PeriodicPair(line.boundaries.start, line.boundaries.end),
    ),),
)
graph = InvariantDomainGraphViscosity(
    mass=RowSumLumping(), invariant_domain=ShallowWaterInvariantDomain(depth_floor=0.1)
        ,
)
result = Simulation(
    problem=problem, mesh=StructuredLineMesher(cells=64).generate(line),
    numerics=Numerics(
        spatial=FiniteElement(element=Lagrange(1), convection=Rusanov(),
            stabilization=graph),
        time_integrator=ClassicalRungeKutta4(), linear_solver=SparseDirect(),
    ),
).run(start_time=0, end_time=0.06, time_step=0.001)
h = result.field(water.depth)
assert min(h.values) >= 0.1
np.testing.assert_allclose(h.integrate().value, h.integrate(time=0).value, atol=1e-10)
x = h.space.dof_coordinates[:, 0]
theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(x, h.at(0).values, "--", label="Initial")
ax.plot(x, h.values, label="After 0.06 s")
ax.set(xlabel="Position [m]", ylabel="Water depth [m]")
ax.legend(frameon=False)
plt.show()
```



Depth evolution with mass-lumped invariant-domain graph viscosity.

10.2.25 Solve a periodic electromagnetic wave with a mimetic curl

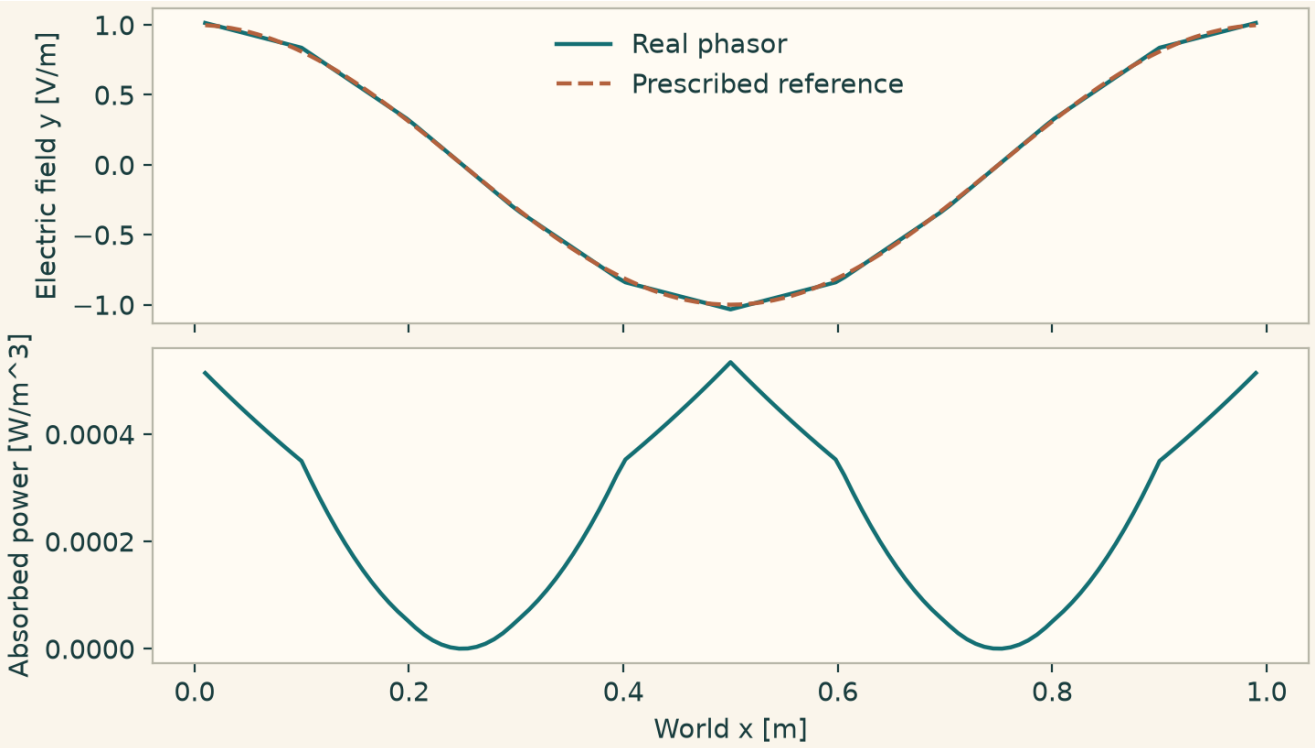
Select the primal-edge Stokes curl on an orthogonal periodic mesh. The impressed current is chosen so that the analytic transverse phasor is a cosine. Plot the phasor and the absorbed power from the same solution.

This listing exercises `MimeticCurl`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.materials.electromagnetic import IsotropicElectromagneticMaterial
from onesim.materials.properties.units.constants import VACUUM_PERMEABILITY as MU0
from onesim.materials.properties.units.constants import VACUUM_PERMITTIVITY as EPS0
from onesim.meshing import PeriodicPair, StructuredHexahedralMesher
from onesim.numerics import Numerics
from onesim.numerics.fvm import FiniteVolume, MimeticCurl
from onesim.numerics.linear import ComplexSparseDirect
from onesim.physics.electromagnetics import FrequencyDomainMaxwell
from onesim.simulation import Problem, Simulation
from onesim.simulation.problem_definition.boundaries import PeriodicBoundary

box = Box(length=1, width=1, height=1)
frequency, conductivity = 1e7, 0.001
omega = 2 * np.pi * frequency
epsilon = EPS0 + 1j * conductivity / omega
factor = (4 * np.pi**2 - omega**2 * MU0 * epsilon) / (1j * omega * MU0)
```

```
def current(x, y, z):
    return np.cos(2 * np.pi * x)[:, None] * np.array([0, factor, 0])


model = FrequencyDomainMaxwell(
    domain=box.volume,
    frequency=frequency,
    impressed_current=current,
    material=IsotropicElectromagneticMaterial(
        relative_permittivity=1,
        relative_permeability=1,
        conductivity=conductivity,
    ),
    phasor_convention="exp(-i*omega*t)",
    amplitude_convention="peak",
)
conditions = tuple(
    PeriodicBoundary(
        physics=model,
        pair=PeriodicPair(
            getattr(box.boundaries, axis + "_min"),
            getattr(box.boundaries, axis + "_max"),
        ),
    )
    for axis in "xyz"
)
mesh = StructuredHexahedralMesher(cells=(10, 10, 10)).generate(box)
sim = Simulation(
    problem=Problem(
        name="Mimetic Maxwell wave",
        geometry=box,
        physics=(model,),
        boundary_conditions=conditions,
    ),
    mesh=mesh,
    numerics=Numerics(
        spatial=FiniteVolume(curl=MimeticCurl()),
        linear_solver=ComplexSparseDirect(),
    ),
)
sim.validate().require_valid()
result = sim.solve_stationary()
x = np.linspace(0.01, 0.99, 121)
points = np.column_stack((x, np.full_like(x, 0.31), np.full_like(x, 0.47)))
ey = result.field(model.electric_field).evaluate(points)[:, 1]
q = np.asarray(result.field(model.absorbed_power_density).evaluate(points)).ravel()
assert np.max(np.abs(ey - np.cos(2 * np.pi * x))) < 0.25
assert q.min() >= 0
theme.use_theme()
fig, (ax, power) = plt.subplots(
    2, 1, sharex=True, figsize=(7.2, 4.1), layout="constrained"
```

```
)
ax.plot(x, ey.real, label="Real phasor")
ax.plot(x, np.cos(2 * np.pi * x), "--", label="Prescribed reference")
ax.set_ylabel("Electric field y [V/m]")
ax.legend(frameon=False)
power.plot(x, q)
power.set(xlabel="World x [m]", ylabel="Absorbed power [W/m^3]")
plt.show()
```



Mimetic curl solution for a driven periodic electromagnetic wave.

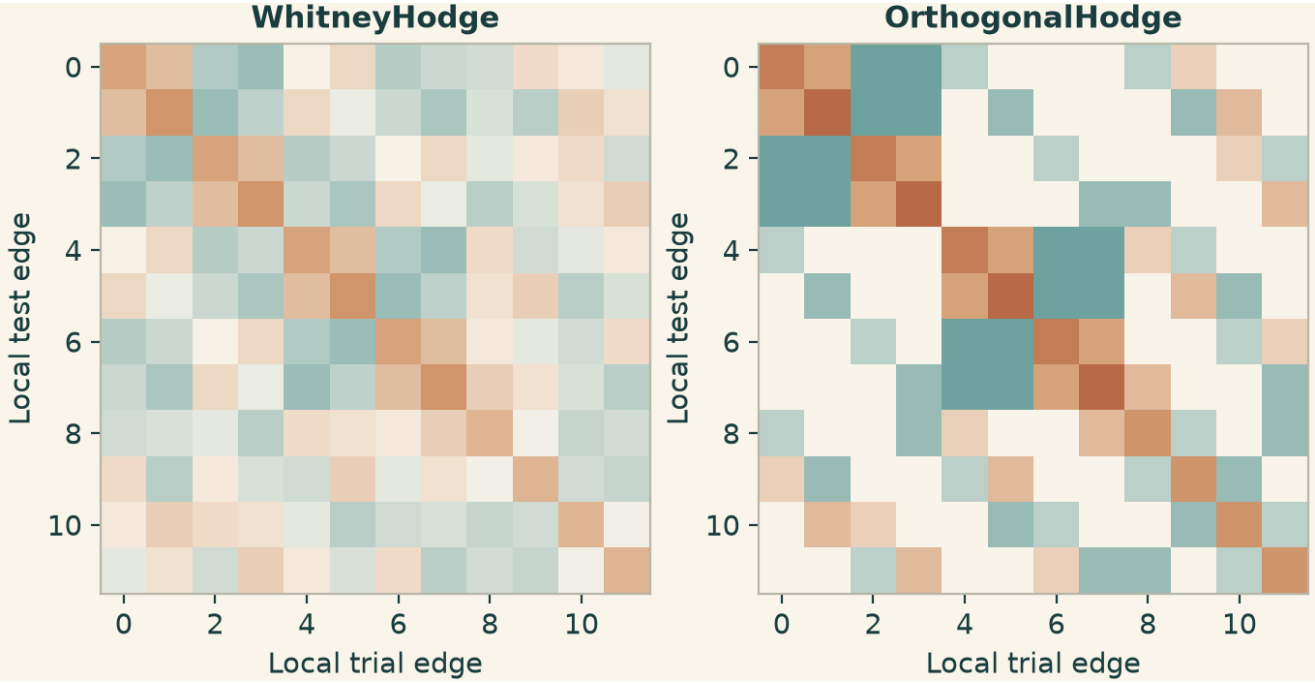
10.2.26 Build the oriented complex and compare Hodge maps

Use the same oriented gradient, curl and divergence with two metric maps. The topological compositions vanish independently of the Hodge choice. Interpolate a vector potential with known curl and compare the local constitutive matrices.

This listing exercises `FiniteIntegration`, `WhitneyHodge`, `OrientedCellComplex`, `OrthogonalHodge`.

```
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Box
from onesim.meshing import StructuredHexahedralMesher
from onesim.numerics.fvm import (
    FiniteIntegration, OrientedCellComplex, OrthogonalHodge, WhitneyHodge,
)

box = Box(length=1.0, width=0.8, height=1.2)
mesh = StructuredHexahedralMesher(cells=(3, 3, 3)).generate(box)
expected = np.array([0.3, 0.4, 0.7])
```

```
theme.use_theme()
fig, axes = plt.subplots(1, 2, figsize=(6.8, 3.5), layout="constrained")
for ax, metric in zip(axes, (WhitneyHodge(), OrthogonalHodge())):
    selected = FiniteIntegration(hodge=metric, topology=OrientedCellComplex())
    operator = selected.bind_complex(
        mesh=mesh, cell_quadrature_order=4, face_quadrature_order=4
    )
    topology = operator.complex
    assert (topology.curl @ topology.gradient).nnz == 0
    assert (topology.divergence @ topology.curl).nnz == 0
    coefficients = operator.space.interpolate(
        lambda p: np.cross(np.broadcast_to(expected, p.shape), p) / 2
    )
    points, _ = operator.space.quadrature(0)
    flux = operator.flux(coefficients, points)
    np.testing.assert_allclose(flux, np.broadcast_to(expected, flux.shape), atol=1e-12)
    matrix, _ = operator.cell_contribution(
        cell=0, curl_coefficient=1.0, mass_coefficient=1.0,
    )
    ax.imshow(matrix, cmap=theme.DIVERGING, vmin=-5, vmax=5)
    ax.set(title=type(metric).__name__, xlabel="Local trial edge", ylabel="Local test
        edge")
plt.show()
```



Local edge matrices for consistent Whitney and diagonal orthogonal metrics.

10.2.27 Replay a thermal step with a stationary Maxwell solve

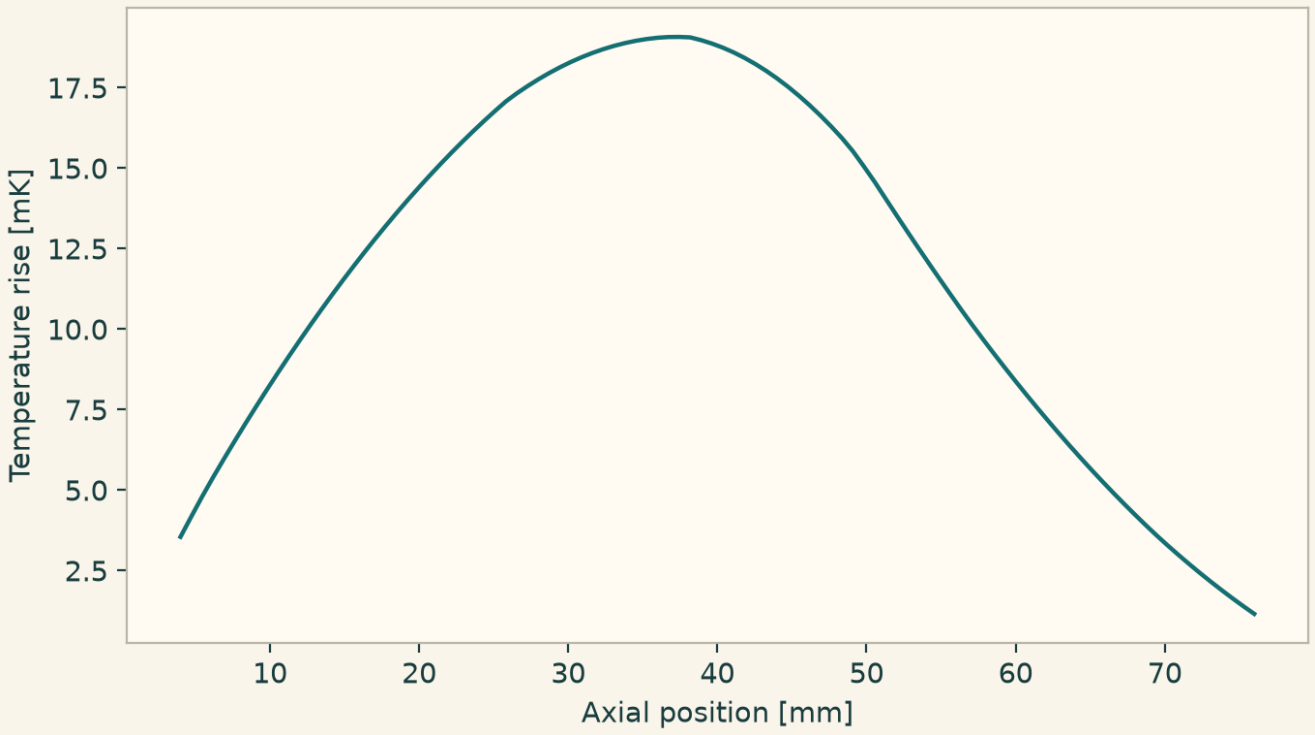
At each accepted thermal step, solve the stationary electromagnetic field, transfer its absorbed power to heat conduction and replay the step while the coupled inputs converge. The conductivity also receives the current temperature through an explicit material input.

This listing exercises `QuasiStationaryCoupling`.

```
from dataclasses import dataclass
import numpy as np
import matplotlib.pyplot as plt
import onesimulation_plot_theme as theme
from onesim.cad.d3 import Cylinder
from onesim.materials import IsotropicSolid
from onesim.materials.properties.units.constants import (
    VACUUM_PERMEABILITY as MU0, VACUUM_PERMITTIVITY as EPS0,
)
from onesim.meshing import TetrahedralMesher
from onesim.numerics.coupled_systems import QuasiStationaryCoupling
from onesim.numerics.discretization import Numerics
from onesim.numerics.fem import FiniteElement, Lagrange, Nedelec
from onesim.numerics.field_transfer import FiniteElementEvaluation
from onesim.numerics.linear import ComplexSparseDirect, SparseDirect
from onesim.numerics.nonlinear import FixedPoint
from onesim.numerics.time import BackwardEuler
from onesim.physics.electromagnetics import (
    CircularWaveguideMode, FrequencyDomainMaxwell, PerfectElectricConductor,
        WaveguidePort,
)
from onesim.physics.heat_transfer import FixedTemperature, HeatConduction
from onesim.physics.plasma.fluid.models.materials.materials import ColdPlasmaResponse
from onesim.simulation import FieldDefinition, Problem, Simulation
from onesim.simulation.field_connections import FieldCoupling
from onesim.simulation.initial_conditions import InitialValue
from onesim.simulation.problem_definition.declarations.physics.fields.fields import
    InputPort

@dataclass(frozen=True)
class ThermalConductor:
    model_id: str = "temperature-dependent-conductor"

    @property
    def temperature(self):
        return InputPort(self, FieldDefinition("temperature", "K", role="input"))

    def fields(self):
        return (self.temperature,)

    def evaluate(self, *, temperature, frequency, phasor_convention,
                 amplitude_convention, points, time):
        sigma = (0.001 * (1 + 0.01 * (temperature - 300)))[:, None, None] * np.eye(3)
        omega = 2 * np.pi * frequency
        epsilon = EPS0 * np.eye(3) + 1j * sigma / omega
        mu = np.broadcast_to(MU0 * np.eye(3), sigma.shape)
        return ColdPlasmaResponse(
            epsilon, mu, sigma, np.zeros(len(points)), frequency,
            phasor_convention, amplitude_convention,
        )
```

```
tube = Cylinder(radius=0.05, length=0.08)
mesh = TetrahedralMesher(target_size=0.045).generate(tube)
solid = IsotropicSolid(
    density=1, specific_heat_capacity=1, thermal_conductivity=1,
    youngs_modulus=1, poisson_ratio=0.3, thermal_expansion_coefficient=0,
)
heat = HeatConduction(domain=tube.volume, material=solid)
material = ThermalConductor()
wave = FrequencyDomainMaxwell(
    domain=tube.volume, frequency=2.45e9, material=material,
    phasor_convention="exp(-i*omega*t)", amplitude_convention="peak",
)
problem = Problem(
    name="Quasistationary heating", geometry=tube, physics=(heat, wave),
    boundary_conditions=(
        FixedTemperature(physics=heat, boundary=tube.boundaries.all, value=300),
        WaveguidePort(physics=wave, boundary=tube.boundaries.bottom,
                      mode=CircularWaveguideMode("TE11"), incident_power=1),
        PerfectElectricConductor(
            physics=wave, boundary=tube.boundaries.all - tube.boundaries.bottom,
        ),
    ),
    initial_conditions=(InitialValue(heat.temperature, 300),),
    couplings=(
        FieldCoupling(source=wave.absorbed_power_density, target=heat.heating,
                      transfer=FiniteElementEvaluation()),
        FieldCoupling(source=heat.temperature, target=material.temperature,
                      transfer=FiniteElementEvaluation()),
    ),
)
result = Simulation(
    problem=problem, mesh={heat: mesh, wave: mesh},
    numerics={
        heat: Numerics(spatial=FiniteElement(element=Lagrange(2)),
                       linear_solver=SparseDirect(), time_integrator=BackwardEuler()),
        wave: Numerics(spatial=FiniteElement(element=Nedelec(2)),
                       linear_solver=ComplexSparseDirect()),
    },
    coupling_solver=QuasiStationaryCoupling(
        transient_systems=(heat,), stationary_systems=(wave,),
        iteration=FixedPoint(relative_tolerance=1e-8, maximum_iterations=30),
    ),
).run(start_time=0, end_time=4e-5, time_step=1e-5)
z = np.linspace(0.004, 0.076, 100)
points = np.column_stack((np.full_like(z, 0.01), np.zeros_like(z), z))
rise = result.field(heat.temperature).evaluate(points) - 300
assert np.max(rise) > 0

theme.use_theme()
fig, ax = plt.subplots(figsize=(6.8, 3.8), layout="constrained")
ax.plot(1000 * z, 1000 * rise)
```

```
ax.set(xlabel="Axial position [mm]", ylabel="Temperature rise [mK]")
plt.show()
```



Axial temperature rise after quasistationary electromagnetic heating.

CHAPTER 11

Define and solve your own equation

OneSimulation can execute an equation that you declare yourself. This chapter builds a stationary reaction–diffusion problem from its coefficients, source and boundary values, solves it on three meshes, and checks the computed field against an exact solution. The equation declaration and numerical configuration remain separate, so the same mathematical problem can be refined without rewriting its physical terms.

The complete example is supplied as `Custom_Equation.ipynb` and `Custom_Equation.py` beside the Showcase sources. The three code cells below reproduce the notebook: declaration, calculation and verification, then plotting. They use the equation interfaces exercised in the inspected source checkout on 11 November 2026. NumPy and Matplotlib are the only additional imports used by the recipe.

11.1 Start with a well-posed mathematical problem

Let $u(x, y)$ be a dimensionless scalar on the square $\Omega = [0, 1] \times [0, 1]$ m. Prescribe a spatially varying diffusion coefficient and a positive linear reaction coefficient:

$$D(x) = D_0(1 + x/L), \qquad D_0 = 1 \text{ m}^2/\text{s}, \quad L = 1 \text{ m}, \quad \kappa = 2 \text{ s}^{-1}.$$

The equation and boundary condition are

$$-\nabla \cdot (D(x)\nabla u) + \kappa u = f(x, y) \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega.$$

Diffusion varies continuously from 1 to 2 m^2/s across the square. The reaction term removes the scalar at a rate proportional to its local value. The source is specified below so that the exact solution is known independently of the numerical solve.

This is a stationary boundary-value problem, so no physical initial condition or time step is supplied. A linear solver's initial iterate belongs to the algebraic calculation; it is not an initial state at a physical time.

Quantity	Meaning	Unit
u	Dependent scalar field	1
x, y, L	Coordinates and reference length	m
$D_0, D(x)$	Diffusion coefficient	m^2/s
κ	Linear reaction coefficient	s^{-1}
f	Distributed source in the equation	s^{-1}

The diffusion, reaction and source terms all have units of s^{-1}. The unit strings in the declaration document those choices. Keep the numerical values and expressions consistent with the stated SI units; changing a unit string does not rescale a coefficient.

11.2 Derive a verification target and its forcing

Choose the exact field

$$u_\star(x,y) = \sin(\alpha x)\sin(\alpha y), \qquad \alpha = \pi/L.$$

For $L = 1$ m, it vanishes on all four edges of the specified square and has unit value at its centre. The forcing must include the derivative of the variable diffusion coefficient. The product rule gives

$$-\nabla\cdot(D\nabla u_\star) = -D\nabla^2 u_\star - \nabla D\cdot\nabla u_\star .$$

Here $\nabla^2 u_\star = -2\alpha^2 u_\star$, $\partial_x D = D_0/L$, and $\partial_y D = 0$. Substitution gives the source used in the notebook:

$$f = (2D(x)\alpha^2 + \kappa)u_\star - \frac{D_0\alpha}{L}\cos(\alpha x)\sin(\alpha y).$$

The second term is essential. Omitting it would solve a different equation even if the resulting plot looked smooth. This construction is a manufactured-solution test: define an exact field first, derive the forcing from the differential equation, and measure how the numerical solution approaches that field as the mesh is refined.

11.3 Declare the field, coefficients and boundary values

The first cell contains the complete mathematical declaration. `EquationModel` identifies the dependent variable, coordinate system and coefficient-form equation. Each `EquationFormula` assigns an expression and unit to a specific role. The role named `absorption` supplies the coefficient κ multiplying u; it does not prescribe the field itself. The name `unit_square` is a domain identifier for this equation declaration.

```
import json
from pathlib import Path
from time import perf_counter

import numpy as np
from onesim.simulation.problem_definition.declarations.physics.models.contracts.models.
    custom_models import (
    EquationFormula, EquationModel, EquationModelForm,
)
from onesim.simulation.problem_definition.declarations.physics.equations.equation_solve
    r import (
    solve_coefficient_form,
)

# Declare the mathematical problem explicitly; all quantities use SI units.
parameters = {"D0": 1.0, "L": 1.0, "kappa": 2.0}  # m^2/s, m, 1/s
source = (
    "(2*D0*(pi/L)**2*(1+x/L)+kappa)*sin(pi*x/L)*sin(pi*y/L)"
    "-D0*pi/L**2*cos(pi*x/L)*sin(pi*y/L)"
)
equation = EquationModel(
```

```
    equation_id="user.variable_diffusion", name="Your own reaction-diffusion equation",
    form=EquationModelForm.COEFFICIENT_PDE, dependent_variables=("u",),
    domain_ids=("unit_square",), dimension=2, coordinate_system="cartesian",
    formulas=(
        EquationFormula("diffusion", "D0*(1+x/L)", "m^2/s"),
        EquationFormula("absorption", "kappa", "1/s"),
        EquationFormula("source", source, "1/s"),
        EquationFormula("dirichlet", "0", "1"),
    ),
)
```

The two adjacent strings assigned to `source` are concatenated by Python. Their mathematical content is exactly the forcing derived above. The coefficient-form execution route evaluates the expressions using the supplied coordinate and parameter values. No predefined heat-transfer, electrostatics or species-transport model appears in this declaration.

11.4 Choose the discretization, solve and verify

The selected coefficient-form execution route assembles a conservative, second-order flux-difference discretization on a structured unit square. `resolution=N` means N intervals along each coordinate: N^2 cells and $(N+1)^2$ reported nodal values. Homogeneous Dirichlet values are prescribed on the boundary, leaving $(N-1)^2$ interior unknowns in the linear solve.

The diffusion coefficient is positive and the reaction coefficient is nonnegative, giving a symmetric positive-definite interior system for this case. The cell below selects conjugate gradients with Jacobi preconditioning, relative tolerance 10^{-10}, absolute tolerance 10^{-12}, and at most 2,000 iterations. The provider and selected method are recorded in the returned solver report and numerical execution plan.

The second cell performs three independent solves. It extracts the computed values, evaluates the exact solution at the same nodes, calculates errors, checks refinement, and saves native arrays plus numerical metadata. The output directory is local to the notebook's working directory.

```
# The current coefficient-form helper assembles a structured unit square.
# Its mesh parameter counts cells per direction; boundaries are prescribed.
out = Path("custom_equation_results")
out.mkdir(parents=True, exist_ok=True)
records = []
for cells in (24, 48, 96):
    started = perf_counter()
    result = solve_coefficient_form(
        equation, resolution=cells, parameter_values=parameters,
        linear_method="cg", preconditioner="jacobi",
        relative_tolerance=1e-10, absolute_tolerance=1e-12,
        maximum_iterations=2000,
    )
    elapsed = perf_counter() - started
    axis = np.asarray(result["coordinates"])
    x, y = np.meshgrid(axis, axis, indexing="ij")
```

```
    u = np.asarray(result["solution"])
    exact = np.sin(np.pi*x/parameters["L"]) * np.sin(np.pi*y/parameters["L"])
    error = u - exact
    max_error = float(np.max(np.abs(error)))
    relative_l2 = float(np.linalg.norm(error) / np.linalg.norm(exact))
    order = None if not records else float(np.log2(records[-1]["max_error"] /
        max_error))
    record = {
        "cells_per_axis": cells, "cells": cells**2, "nodes": u.size,
        "unknowns": result["unknowns"], "nonzeros": result["nonzeros"],
        "max_error": max_error, "relative_l2_error": relative_l2,
        "observed_max_error_order": order, "elapsed_seconds": elapsed,
        "residual_norm": result["residual_norm"], "solver": result["solver"],
        "numerical_plan": result["numerical_plan"],
    }
    assert np.isfinite(u).all() and u.shape == (cells+1, cells+1)
    assert result["solver"]["relative_residual"] < 2e-10
    assert order is None or 1.9 < order < 2.1
    np.savez_compressed(out/f"reaction_diffusion_{cells}.npz", x=x, y=y, u=u,
                        exact=exact, error=error, axis=axis)
    records.append(record)
    print(f"{cells:2d} x {cells:2d} cells | {result['unknowns']:5d} unknowns | "
          f"max error {max_error:.8e} | relative L2 {relative_l2:.8e}", flush=True)

metrics = {"equation": equation.to_dict(), "parameters_SI": parameters,
           "exact_solution": "sin(pi*x/L)*sin(pi*y/L)",
           "domain": "[0, 1] m x [0, 1] m", "runs": records}
(out/"metrics.json").write_text(json.dumps(metrics, indent=2)+"\n")
assert records[-1]["max_error"] < 1e-4
print(f"Verified second-order refinement. Results: {out}")
```

The exact field is used only after each solve to calculate the verification error. The equation receives the derived source, coefficients and boundary condition. Saving `u`, `exact` and `error` separately makes that distinction inspectable in the resulting NPZ files.

The first assertion checks that the field is finite and has the expected nodal shape. The solver-report assertion checks convergence of the assembled algebraic equations. The observed-order assertion and final error threshold check the spatial approximation against the independent reference. These checks answer different questions and should remain separate in a notebook.

11.5 Read the measured refinement results

Define the maximum nodal error and relative discrete ℓ^2 error by

$$E_\infty = \max_{i,j} |u_{ij} - u_{\star,ij}|, \qquad E_{2,\mathrm{rel}} = \frac{\left(\sum_{i,j} |u_{ij} - u_{\star,ij}|^2\right)^{1/2}}{\left(\sum_{i,j} |u_{\star,ij}|^2\right)^{1/2}}.$$

These sums use the reported nodes, including the prescribed boundary values. The second quantity is the discrete relative norm used in the code, rather than a separately integrated

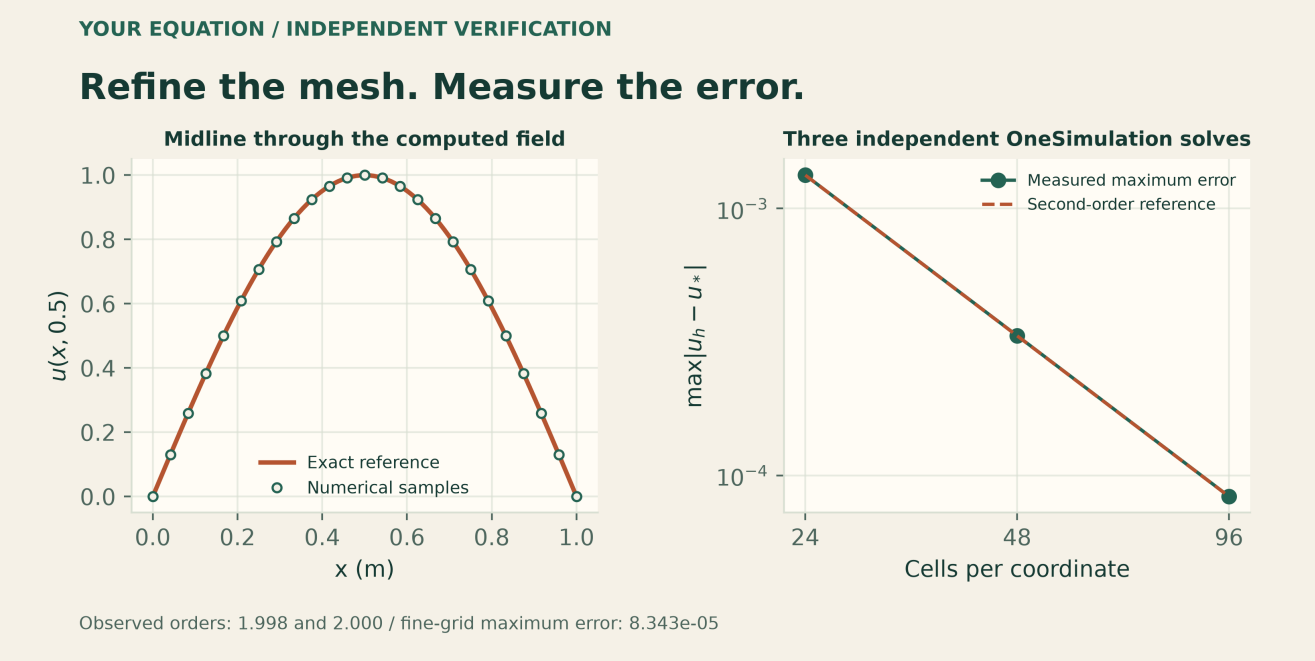
continuum error. Halving the uniform spacing gives the observed maximum-error order

$$p = \log_2 \left(E_\infty(h)/E_\infty(h/2)\right), \qquad h = 1/N \text{ m}.$$

The executed example produced the following results. Values are rounded for display; `metrics.json` retains the full recorded precision.

Cells	Interior unknowns	E_∞	$E_{2,\text{rel}}$	Observed p
24×24	529	1.333454×10^{-3}	1.327800×10^{-3}	—
48×48	2,209	3.337633×10^{-4}	3.317508×10^{-4}	1.99827
96×96	9,025	8.342833×10^{-5}	8.292525×10^{-5}	2.00022

The finest calculation reports 9,409 nodal values, of which 384 are prescribed boundary values, and a sparse interior matrix with 44,745 nonzero entries. Conjugate gradients converged in 138 iterations with a reported relative residual of 4.0647×10^{-11}. The maximum field error is about four times smaller after each mesh refinement, consistent with second-order spatial convergence for this smooth problem.



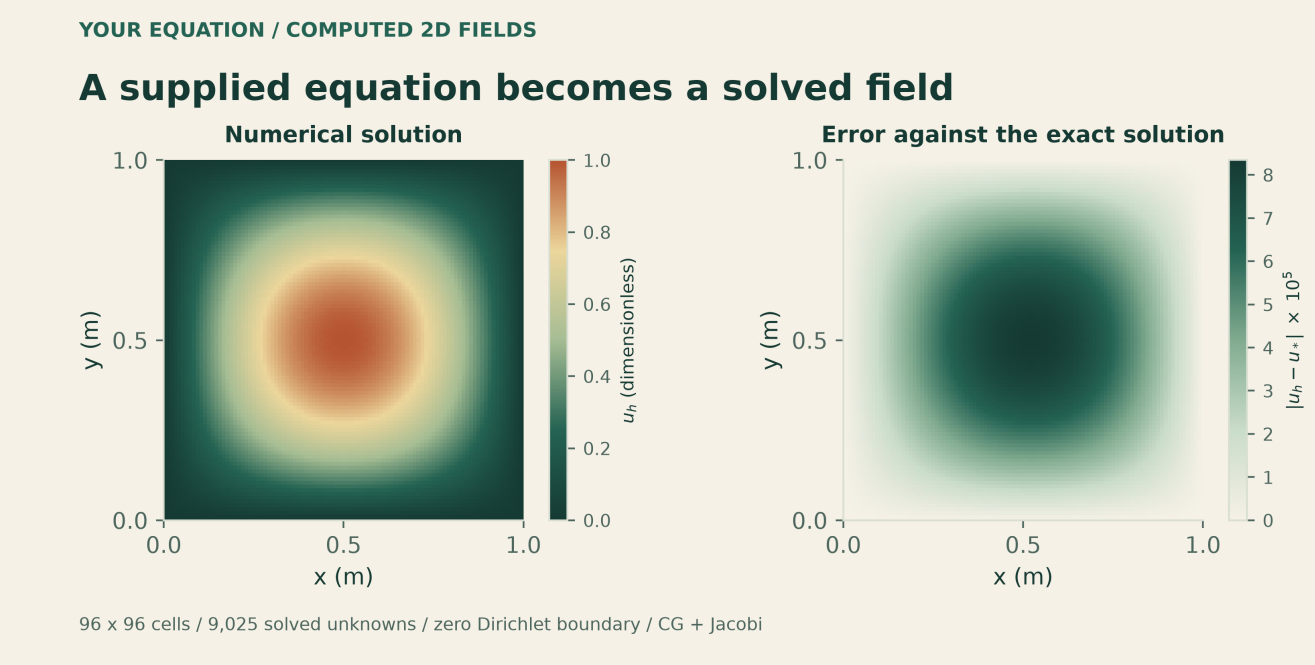
Measured errors from three separate coefficient-form solves. The reference slope describes second-order convergence; it is not a fitted replacement for the computed values.

A small algebraic residual does not make the spatial discretization exact. Here the residual is already much smaller than the error against $u_\star$. Tightening the linear tolerance alone would not remove the leading mesh error. Conversely, a poorly converged linear solve can obscure a genuine mesh-convergence trend.

11.6 Plot the checked solution and its error

The third cell plots the final computed field and its signed difference from the reference, followed by the refinement curve. It uses the numerical arrays left by the finest solve and the recorded errors from all three resolutions. The two-dimensional spatial support is labelled explicitly; adding a perspective view would not turn this calculation into a three-dimensional volume problem.

```
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap

cream, forest, copper = "#f4f1e8", "#173f37", "#b75b36"
cmap = LinearSegmentedColormap.from_list("OneSimulation", [forest, "#90b39a",
    "#e4d5a0", copper])
plt.rcParams.update({"font.family": "DejaVu Sans", "text.color": forest,
                     "axes.labelcolor": forest, "xtick.color": forest,
                     "ytick.color": forest, "axes.edgecolor": forest})
fig, axes = plt.subplots(1, 3, figsize=(13.5, 4.2), constrained_layout=True)
fig.patch.set_facecolor(cream)
for ax in axes:
    ax.set_facecolor(cream)
for ax, values, title in zip(axes[:2], (u, error), ("Your computed field", "Error
    against the exact solution")):
    image = ax.pcolormesh(x, y, values, shading="auto", cmap=cmap)
    ax.set(title=title, xlabel="x [m]", ylabel="y [m]", aspect="equal")
    fig.colorbar(image, ax=ax, shrink=0.75, label="u [1]" if ax is axes[0] else "u -
        exact [1]")
h = np.array([1/r["cells_per_axis"] for r in records])
e = np.array([r["max_error"] for r in records])
axes[2].loglog(h, e, "o-", color=copper, label="Measured maximum error")
axes[2].loglog(h, e[-1]*(h/h[-1])**2, "--", color=forest, label="Second-order
    reference")
axes[2].set(title="Verified mesh refinement", xlabel="Cell width h [m]",
    ylabel="Maximum absolute error [1]")
axes[2].legend(fontsize=8)
fig.suptitle("OneSimulation / Your own equation", fontsize=17, fontweight="bold")
fig.savefig(out/"custom_equation.png", dpi=220, facecolor=cream)
fig.savefig(out/"custom_equation.svg", facecolor=cream)
plt.show()
```



The computed scalar field and its error on the 96×96 *grid. The source includes both the Laplacian contribution and the gradient of the spatially varying diffusion coefficient.*

The notebook exports a PNG and an editable SVG beside its numerical results. Keep the NPZ files and `metrics.json` with the figure so another reader can reproduce the error definitions and identify the actual solver mesh. Increasing the image resolution changes presentation only; changing `resolution` triggers a new numerical solve.

11.7 Adapt the example within its execution contract

To make a new case, edit the coefficient expressions, parameter values, source and boundary value in the first cell, then repeat the verification workflow. Retain a simple exact solution or a known limiting case while changing the physics. Once an independent reference is unavailable, use suitable balance checks and mesh refinement of the quantities you intend to report.

The executed route has specific limits. It solves one stationary scalar coefficient-form diffusion–reaction equation on a structured one- or two-dimensional Cartesian domain. This two-dimensional helper uses a unit square with a prescribed Dirichlet value on its boundary. In the inspected implementation, the two-dimensional resolution limit is 96 intervals per direction. The parameter `L` in this example appears in the coefficient and reference expressions; changing its value does not resize the helper's domain. The value $L = 1$ m is therefore part of the manufactured reference used here.

This recipe exercises spatially variable scalar diffusion and a linear reaction term. It does not establish a general three-dimensional custom-PDE route, an arbitrary CAD-domain solve, or an arbitrary nonlinear constitutive law. Other equation forms and numerical routes have their own supported fields, dimensions, boundary conditions and integration contracts. Check the selected execution path rather than inferring support from the ability to store an equation declaration.

The measured convergence establishes numerical consistency for this manufactured problem. It does not validate a particular material, device or experiment. The useful result is a complete workflow you can inspect and modify: state your equation, derive a checkable reference, choose numerics, execute the model, and measure what changes under refinement.

CHAPTER 12

Diagnose a notebook result

Symptom	Check in order
Validation fails	Domain and selection ownership, fields initialized for their roles, support for each boundary and spatial method, solver and scalar type.
Singular or inconsistent solve	Pressure, potential or rigid-body nullspaces, gauge constraints, global source–flux balance and redundant conditions.
Transient diverges or clips the step	Physical source scales, explicit stability and CFL, stage solver residual, time-step limits and mesh resolution.
Apparently smooth but wrong field	Carrier type, coordinates, units, component order, real versus complex representation and reference sampling points.
Fine and coarse pictures differ	Match physical points and color limits. Compute the observable and error on both meshes using the same definition.
Results missing at a requested time	Check accepted output times, retention policy and which references were written to the sink.

12.1 Repository examples

Radiation `packaging/runtime_customer/01_Waermestrahlung.ipynb`

Vibration `packaging/runtime_customer/02_Gedaempfte_Schwingung.ipynb`

Cantilever `packaging/runtime_customer/03_Kragtraeger_3D.ipynb`

Recirculation `packaging/runtime_customer/04_Waermetransport_im_Umlauf.ipynb`

Solenoid `packaging/runtime_customer/05_Magnetostatik_Spule_3D.ipynb`

The public API layout is described in `docs/api-target/README.md`. For release-specific installation and feature coverage see `packaging/runtime_customer/README_DE.md` and `packaging/runtime_customer/FUNKTIONSUEBERBLICK_DE.md`. The following reference reflects the inspected Python exports rather than the historical showcase inventory.

CHAPTER 13

Coverage and navigation by symbol

This lookup covers the 138 canonical declarations and functions in the nine numerical subpackages. The package-root aliases point to the same declarations. Each entry links to an executable listing in the guide. Chapter B contains complete signatures, and Chapter C contains functions exported through the specialized operator namespaces.

13.1 Physics model and service coverage

This index covers the 73 public model declarations that enter the field-based simulation route. The service links identify independently callable facades and directly exercised providers. Constitutive laws and boundary declarations appear with the cases that consume them and retain their own signatures in Chapter A.

13.1.1 Acoustics

`PressureDrivenMode` (6.4.1), `RadiatingMode` (6.4.2), `UltrasonicAcoustics` (6.4.4), `LinearAcoustics` (6.4.3).

Service routes include `AcousticsPublicApi` (6.4.5), `StructuralAcousticsPublicApi` (6.4.6), `UltrasoundPublicApi` (6.4.7), `VibroacousticsPublicApi` (6.4.8).

13.1.2 Biophysics

`OrganClearance` (6.5.1), `AxialTissue` (6.5.4), `PassiveMembrane` (6.5.2), `CompliantVessels` (6.5.3).

Service routes include `BioelectricalPublicApi` (6.5.5), `CirculatoryPublicApi` (6.5.6), `OrganPublicApi` (6.5.7), `TissuePublicApi` (6.5.8), `BioheatProvider` (6.5.9).

13.1.3 Chemistry

`SpeciesTransport` (5.4), `LumpedElectrode` (6.6.1), `LumpedBattery` (6.6.2), `ReactionKinetics` (6.6.3).

13.1.4 Electromagnetics

`FrequencyDomainMaxwell` (6.7.5), `SeriesRLC` (6.7.1), `AveragedBuckConverter` (6.7.2), `DCMotor` (6.7.3), `MagneticDiffusion` (6.7.4).

13.1.5 Electrostatics

`PoissonEquation` (5.6).

13.1.6 Geotechnics

`GroundwaterFlow` (7.5), `BiotConsolidation` (6.9.3), `TerzaghiConsolidation` (6.9.1), `BuckleyLeverettFlow` (6.9.2).

Service routes include `GroundwaterPublicApi` (6.9.4), `PoromechanicsPublicApi` (6.9.5), `ReservoirPublicApi` (6.9.6), `RockMechanicsPublicApi` (6.9.7), `SoilMechanicsPublicApi` (6.9.8).

13.1.7 Heat transfer

`HeatConduction` (3.2), `RadiationCooling` (2.1), `PhaseChangeHeating` (6.10.1), `HeatAdvectionDiffusion` (6.10.2), `ConvectiveHeatTransfer` (3.2).

13.1.8 Hydrodynamics

`IncompressibleNavierStokes` (6.11.7), `CompressibleEuler` (6.11.2), `ShallowWater` (6.11.3), `PhaseAdvection` (5.9.1), `ReactiveMixtureTransport` (6.11.6), `OldroydBShear` (6.11.1), `DarcyPressureFlow` (6.11.5), `CahnHilliard` (6.11.8), `TopographicShallowWater` (6.11.4).

13.1.9 Magnetohydrodynamics

`TransverseMHDWaves` (5.10), `TwoFluidExchange` (6.12.1), `InductionlessMomentum` (6.12.3), `RadiationMatterExchange` (6.12.2).

Service routes include `MagnetohydrodynamicPublicApi` (6.12.4).

13.1.10 Magnetostatics

`Magnetostatics` (3.3).

13.1.11 Multibody dynamics

`LinearJointSystem` (6.14.5), `DrivenFourBar` (6.14.3), `TorsionalDrivetrain` (6.14.4), `EulerTop` (6.14.2), `SimplePendulum` (6.14.1).

Service routes include `DrivetrainPublicApi` (6.14.6), `JointSystemsPublicApi` (6.14.7), `RigidBodyDynamicsPublicApi` (6.14.8), `KinematicsPublicApi` (6.14.9), `MechanismsPublicApi` (6.14.10).

13.1.12 Multiphysics

`HeatedResistor` (6.15.1), `FluidColumnPiston` (6.15.2), `VoiceCoil` (6.15.3), `ThermalExchange` (6.15.4), `AdiabaticReaction` (6.15.5).

Service routes include `ElectroThermalPublicApi` (6.15.6), `FluidStructurePublicApi` (6.15.7), `MagnetoMechanicsPublicApi` (6.15.8), `ThermoMechanicsPublicApi` (6.15.9), `GeneralCoupledSubsystemsPublicApi` (6.15.10), `ReactiveCouplingPublicApi` (6.15.11).

13.1.13 Optics and photonics

`ComplexParaxialEnvelope` (5.14), `ParaxialEnvelope` (6.16.2), `PhotonEnvelope` (6.16.1).

Service routes include `OpticsPublicApi` (6.16.3), `PhotonicsPublicApi` (6.16.4).

13.1.14 Plasma physics

`NativeEcrPlasma` (6.17.7), `DriftDiffusionPlasma` (6.17.6), `ChargedParticle` (6.17.1), `UniformGuidingCenter` (6.17.2), `ColdPlasmaOscillation` (6.17.3), `FreeStreamingWaterbag` (6.17.4), `IsothermalElectronWaterbag` (6.17.5).

13.1.15 Solid mechanics

`LinearElasticity` (3.1), `StructuralOscillator` (2.2), `MaxwellRelaxation` (6.18.1), `FatigueCrackGrowth` (6.18.2), `HertzImpact` (6.18.3), `CohesiveOpening` (6.18.4).

13.1.16 Systems and networks

`HydraulicAccumulator` (6.19.1), `ThermalCapacitance` (6.19.2), `ResistorCapacitor` (6.19.3), `FirstOrderControl` (6.19.4), `WellMixedTank` (6.19.5), `SealedGasChamber` (6.19.6).

Service routes include `ControlSignalFlowPublicApi` (6.19.7), `HydraulicSystemsPublicApi` (6.19.8), `PneumaticSystemsPublicApi` (6.19.9), `ProcessModelsPublicApi` (6.19.10), `ThermalNetworksPublicApi` (6.19.11), `ElectricalNetworksPublicApi` (6.19.12), `ZeroDOneDModelsPublicApi` (6.19.13), `FieldSystemCouplingProvider` (6.19.14).

13.2 discretization selection index

See Section 9.1 for a worked comparison in this layer.

`count`. Validate integer controls before building a declaration. For example, `count(cells, "cells", minimum=2)` returns the integer or raises `ValueError`. Example in Section 9.1.2.

`Numerics`. Compose independent spatial, linear, nonlinear, time, and coupling selections. Initial guesses and scales are mappings keyed by owned field references. Example in Section 9.9.1.

`NumericalExecutionPlan`. Inspect resolved `bindings` and `provider_for(...)` after numerical resolution. Use `to_dict()` to serialize the resolved record. Example in Section 10.1.1.

`NumericalIR`. Inspect equation, field and capability requirements before provider resolution. Use its mapping and content fingerprint for diagnostics. Example in Section 10.1.1.

`NodalInterpolation`. Evaluate initial values at native nodes through the spatial initialization port. Example in Section 9.1.1.

`CellAverageProjection`. Initialize FV cell means by physical volume quadrature. Set quadrature_order when the model needs a different rule. Example in Section 10.2.18.

`L2Projection`. Project initial values by the supported mass solve with a selectable quadrature order. Example in Section 9.1.1.

`LumpedParameters`. Choose the zero-dimensional spatial contract for a LumpedMesh. Optional collaborators supply stability or algebraic operations. Example in Section 9.9.1.

`OperatorConditions`. Implement the structural condition-evaluation interface when binding a custom spatial operator. The operator owns the physical boundary support and receives evaluation time. Example in Section 10.1.2.

`OperatorFacetTopology`. Implement the facet-topology interface when a custom operator needs to expose physical face IDs, orientations and boundary membership. Example in Section 10.1.2.

`EvaluationCacheControl`. Expose the evaluation-cache lifecycle on a bound operator when its residual or tangent reuses geometry-dependent data. Example in Section 10.1.2.

`NumericalCompatibilityError`. Exception with a machine-readable rejection kind, rejected option and required capabilities. Example in Section 10.1.1.

`MethodProviderOffer`. Pass a provider's actual capability offer to compatibility evaluation. Availability and diagnostics describe the supplied context. Example in Section 9.1.2.

`evaluate_method_provider`. Compare the supplied provider's offer against method requirements and any additional requirements. Returns a compatibility report. Example in Section 9.1.2.

`CapabilitySet`. Describe the feature and guarantee identifiers offered by a provider. Example in Section 9.1.2.

`RequirementSet`. Specify required and preferred feature identifiers for compatibility checks. Example in Section 9.1.2.

13.3 fdm selection index

See Section 9.2 for a worked comparison in this layer.

`FiniteDifference`. Select the named stencil, flux, boundary, curl or source collaborators required by the model's FD binding. Example in Section 10.2.3.

`CentralDifference`. Use as gradient for a second-order centered first derivative on supported uniform samples. Example in Section 7.5.

`CentralLaplacian`. Use as diffusion for a second-order scalar Laplacian on a uniform sample axis. Example in Section 7.5.

`PolynomialDifference`. Use as gradient for a polynomial stencil at the requested accuracy order on a supported mesh. Example in Section 9.2.1.

`PolynomialLaplacian`. Use as diffusion for a polynomial second-derivative stencil on supported coordinates. Example in Section 6.12.3.

`SummationByPartsDifference`. Select a first derivative with a summation-by-parts boundary structure for a compatible line binding. Example in Section 10.2.4.

`HydrostaticFluxDifference`. Select shallow-water hydrostatic flux differences with a matching Riemann flux, invariant domain and optional dry linearization. Example in Section 10.2.4.

`BoundaryFluxDifference`. Wrap a first derivative and its optional boundary flux to use physical exterior traces in conservative transport. Example in Section 10.2.3.

`GlobalLaxFriedrichsSplit`. Supply a flux splitting based on a global wave-speed bound to WENOFluxDifference. Example in Section 10.2.3.

`WENO5Flux`. Supply fifth-order WENO flux reconstruction for the supported split-flux stencil. Example in Section 10.2.3.

`WENOFluxDifference`. Combine a flux splitter and a reconstruction in the conservative gradient port. A diagnostic gradient may supply a separate derivative for output. Example in Section 10.2.3.

13.4 fvm selection index

See Section 9.3 for a worked comparison in this layer.

`MonotoneUpwind`. Select scalar upwinding with a physical flux monotone in the declared direction. Example in Section 10.2.3.

`FiniteVolume`. Compose control-volume term operators, reconstruction, limiter, initialization, and cell and face quadrature. Example in Section 10.2.3.

`Rusanov`. Select the local Lax–Friedrichs flux using the physical model's wave-speed estimate. Example in Section 10.2.4.

`LinearUpwind`. Select a linear upwind reconstruction for the compatible FV convective operator. Example in Section 3.2.

`CentralDifference`. Select the central FV diffusion contribution on supported carriers. This is a different declaration from fdm.CentralDifference. Example in Section 10.2.13.

`LeastSquaresGradient`. Choose least-squares reconstruction of cell gradients for supported face fluxes and field diagnostics. Example in Section 3.2.

`RhieChow`. Select collocated face-velocity interpolation with pressure correction in a supported flow model. Example in Section 10.2.13.

`ScharfetterGummel`. Choose an exponential-fitting flux for drift–diffusion. Supply a gradient collaborator if the binding requires one. Example in Section 6.17.6.

`ImplicitLocalSources`. Select the implicit local source treatment for models exposing a compatible reaction or source operator. Example in Section 6.17.6.

`DifferentiableSamples`. Create differentiable samples with `DifferentiableSamples.variables(values)` and propagate the sparse tangent through supported arithmetic. Inspect `.jacobian` alongside `.values`. Example in Section 9.3.2.

`MimeticCurl`. Choose the mimetic discrete curl route for supported electromagnetic models. Example in Section 10.2.25.

`TwoPointDiffusion`. Select a two-point diffusion flux based on adjacent cell values. Example in Section 7.5.

`MUSCL`. Reconstruct conservative piecewise-linear face states. Select the limiter separately. Example in Section 5.4.

`ConvexInvariantDomain`. Retract face states toward their unchanged cell mean to satisfy the bound invariant domain. Example in Section 10.2.7.

`MinmodLimiter`. Limit two scalar one-sided slopes with minmod to obtain a TVD slope on the supported stencil. Example in Section 9.3.1.

`MonotonizedCentralLimiter`. Use a monotonized central TVD slope with less flattening than minmod in smooth regions. Example in Section 10.2.5.

`VanLeerLimiter`. Use a harmonic TVD slope where the one-sided slopes agree in sign. Example in Section 9.3.1.

`HydrostaticReconstruction`. Reconstruct depth and discharge consistently with the bed for a supported well-balanced method. Example in Section 6.11.4.

`WetDryActiveSet`. Select the wet/dry active-set treatment for a compatible shallow-water linearization. Example in Section 10.2.22.

`SuperbeeLimiter`. Choose a compressive TVD limiter for a compatible scalar slope reconstruction. Example in Section 10.2.5.

`VenkatakrishnanLimiter`. Use smooth slope limiting with a regularizing epsilon. Example in Section 10.2.5.

`BarthJespersenLimiter`. Limit reconstructed values to the local neighbor range on a supported multidimensional stencil. Example in Section 10.2.5.

`TvbLimiter`. Wrap a base limiter with a mesh-width and curvature threshold for suppressing limiting near smooth extrema. Example in Section 10.2.5.

`BoundaryLeastSquaresGradient`. Reconstruct cell gradients with boundary data through the supported least-squares port. Example in Section 10.2.17.

`CentralElasticStress`. Choose the central FV stress contribution for a compatible elasticity formulation. Example in Section 10.2.17.

`WetDryTraceScaling`. Retract depth and discharge traces by one factor toward admissible means. Example in Section 10.2.7.

`Eno2Reconstructor`. Select a smooth second-order scalar ENO stencil at the face. Example in Section 10.2.6.

`Eno3Reconstructor`. Select a smooth quadratic scalar ENO stencil at the face. Example in Section 10.2.6.

`Weno3Reconstructor`. Reconstruct a scalar left trace with WENO3 weights on the supported uniform line stencil. Example in Section 10.2.6.

`Weno5Reconstructor`. Reconstruct a scalar left trace with WENO5 weights on the supported uniform line stencil. Example in Section 10.2.6.

`CellGradientReconstruction`. Declare a native cell-gradient reconstruction component for compatible volume carriers. Example in Section 10.2.23.

`FiniteIntegration`. Select a discrete incidence complex and metric Hodge maps for a compatible curl operator. Example in Section 10.2.26.

`WhitneyHodge`. Use Whitney reconstruction for metric edge and face maps, with optional reconstruction collaborators. Example in Section 10.2.26.

`OrientedCellComplex`. Represent oriented incidence between discrete geometric entities, independently of the metric. Example in Section 10.2.26.

`OrthogonalHodge`. Select orthogonal metric Hodge maps on a supported orthogonal carrier. Example in Section 10.2.26.

13.5 fem selection index

See Section 9.4 for a worked comparison in this layer.

`HcurlNitsche`. Use in boundary_enforcement for weak tangential traces with a dimensionless penalty. Example in Section 10.2.16.

`ConstraintAugmentation`. Add a weighted constraint contribution to a compatible primal block while preserving the exact constraint. Example in Section 10.2.16.

`FiniteElement`. Choose an element space, integration rule, stabilization, boundary enforcement and compatible term components. Example in Section 9.5.1.

`Lagrange`. Use as element for a continuous nodal space. Order is total degree on simplices and per-coordinate degree on supported hexahedra. Example in Section 9.5.1.

`DiscontinuousLagrange`. Use as element for cell-local polynomials. Order zero gives a cellwise constant space. Example in Section 9.4.2.

`Nedelec`. Use as element for first-family H(curl) edge elements. The positive public order starts at one. Example in Section 6.17.7.

`ConsistentMassStability`. Compute a compatible generalized spectral bound from the free stiffness and consistent mass blocks. Configure dense threshold and eigensolver tolerance. Example in Section 10.2.10.

`SUPG`. Select streamline-upwind Petrov–Galerkin stabilization with tau and a supported strong residual. Example in Section 10.2.8.

`GLS`. Select Galerkin least-squares stabilization with a scale tau and a compatible strong residual. Example in Section 10.2.8.

`CIP`. Select a continuous interior penalty on jumps of physical first derivatives. Example in Section 10.2.8.

`DGPolynomialLimiter`. Apply a slope reduction and optional convex admissibility scaling on a compatible DG stage. Example in Section 10.2.9.

`StateLimiterPort`. Implement the limiter contract consumed by a bound conservative stage. Apply the change on the stage carrier and check admissibility and preserved means. Example in Section 10.2.9.

`ScalarInterval`. Constrain the named scalar field to an interval by mean-preserving scaling in a supported limiter. Example in Section 10.2.9.

`DGJumpPenalty`. Use a jump-based interior dissipation contribution with an explicit nonnegative coefficient. Example in Section 10.2.8.

`SymmetricInteriorPenalty`. Select symmetric interior-penalty diffusion for DG with an explicit penalty. Example in Section 10.2.8.

`InvariantDomainGraphViscosity`. Choose a continuous-graph viscosity treatment using a selected mass operator and invariant domain. Example in Section 10.2.24.

`RowSumLumping`. Supply row-sum mass lumping to a compatible nodal or graph-viscosity method. Example in Section 10.2.4.

`OrderedFields`. Pass the lower and upper field names and minimum separation as an invariant domain for a compatible limiter. The waterbag recipes illustrate the required ordering. Example in Section 10.2.9.

`TaylorHood`. Select a mixed velocity–pressure space with pressure one polynomial degree lower than velocity. Example in Section 6.11.7.

`GradDiv`. Select a penalty on the divergence of the velocity in a compatible mixed momentum assembly. Example in Section 10.2.12.

`MixedMomentumContribution`. Provide explicit mass, momentum, pressure or load pieces for the mixed momentum assembly. Example in Section 10.2.12.

`Nitsche`. Enforce compatible diffusion or inflow traces weakly. Penalty and symmetry select the boundary form. Example in Section 6.9.3.

`RaviartThomas`. Use as the H(div) flux space with polynomial subdegree k. RT0 has order zero. Example in Section 9.4.2.

`MixedDiffusionElement`. Pair an H(div) flux space with a discontinuous scalar element for a mixed diffusion solve. Example in Section 9.4.2.

`BrezziDouglasMarini`. Use as the H(div) flux space with complete vector polynomials of order at least one. Example in Section 9.4.2.

`GalerkinDiffusion`. Select the Galerkin diffusion contribution where the physical residual exposes that port. Example in Section 6.11.8.

`VolumeCIP`. Supply an explicit diffusivity for continuous interior-penalty contributions on supported volume FE interior facets. Example in Section 10.2.11.

`InterpolatedFluxDivergence`. Select this convection contribution for a continuous formulation that interpolates the physical flux before taking its divergence. Example in Section 10.2.11.

`PSPG`. Set the stabilization time scale for the supported pressure-stabilized mixed residual. Example in Section 10.2.12.

`MixedContinuityContribution`. Provide explicit mass, velocity, pressure or load pieces for the mixed continuity assembly. Example in Section 10.2.12.

`EqualOrderVelocityPressure`. Select equal-degree velocity and pressure spaces with a compatible stabilized formulation. Example in Section 10.2.12.

`HydrostaticGraphViscosity`. Combine graph viscosity with hydrostatic equilibrium and a shallow-water invariant domain. Example in Section 10.2.4.

`GalerkinAdvection`. Select the Galerkin advection contribution in an appropriate finite-element weak form. Example in Section 6.11.8.

`DissipativeMassStability`. Resolve a spectral interval or an M-norm contractivity bound for the supported dissipative operator. Example in Section 10.2.10.

13.6 linear selection index

See Section 9.5 for a worked comparison in this layer.

`GMRES`. Select restarted or unrestarted GMRES with tolerances, scalar type, and an optional compatible preconditioner. Example in Section 9.6.1.

`RightGMRES`. Use fixed right preconditioning and check convergence on the physical residual. Example in Section 9.5.1.

`FlexibleGMRES`. Use flexible right preconditioning, retaining the actual preconditioned search vectors. Example in Section 9.5.1.

`ConjugateGradient`. Solve a compatible positive-definite symmetric or Hermitian linear system. Check the operator and preconditioner guarantees. Example in Section 9.5.1.

`SparseDirect`. Select assembled sparse factorization for the declared scalar type and available provider. Example in Section 9.7.2.

`ComplexSparseDirect`. Select the complex sparse direct route, for example for frequency-domain Maxwell phasors. Example in Section 6.7.5.

`ILU`. Use incomplete sparse factorization with selectable fill and drop tolerance. Example in Section 9.5.1.

`AlgebraicMultigrid`. Use a compatible multigrid preconditioner with optional near-nullspace information. Example in Section 10.2.13.

`BlockPreconditioner`. Use a diagonal or triangular sweep of inner solvers over an explicit or provider-owned block partition. Example in Section 10.2.15.

`SchurPreconditioner`. Use a leading-block solve and Schur-complement solve, with an optional approximation. Example in Section 10.2.15.

13.7 nonlinear selection index

See Section 9.6 for a worked comparison in this layer.

`Newton`. Solve a nonlinear residual using a selected Jacobian route and inner linear solver. Configure tolerances and relaxation. Example in Section 9.7.2.

`DampedNewton`. Select this declaration in the nonlinear solver lab, where it uses an Armijo backtracking step. Example in Section 9.6.1.

`TrustRegionNewton`. Select this declaration in the nonlinear solver lab and supply a trust radius in solver coordinates. Example in Section 9.6.1.

`QuasiNewton`. Select this declaration in the nonlinear solver lab to use inverse Broyden updates after an initial solve. Example in Section 9.6.1.

`FixedPoint`. Iterate a compatible fixed-point map with relaxation, tolerances and a maximum iteration count. Example in Section 9.9.1.

`JFNK`. Select Newton–Krylov with a supported Jacobian action and explicit compatible Krylov solver. Example in Section 9.6.1.

`JacobianFreeNewtonKrylov`. Use the long name for `JFNK`; the nonlinear solver lab constructs that same declaration with `GMRES`. Example in Section 10.2.14.

13.8 time selection index

See Section 9.7 for a worked comparison in this layer.

`TimeEventDefinition`. Provide an event function of accepted semantic state, a direction and terminal behavior. For a field-based terminal event the time lab uses `FieldThreshold`. Example in Section 10.1.3.

`TimeMethod`. Select a registered time algorithm by method_id, then supply the stage solver required by its binding. Example in Section 9.7.1.

`ClassicalRungeKutta4`. Select the four-stage explicit fourth-order RK method. The time step must resolve the spatial stability bound. Example in Section 9.9.1.

`ForwardEuler`. Select first-order explicit Euler with separately declared step limits. Example in Section 9.7.1.

`BackwardEuler`. Select first-order implicit Euler and supply a compatible nonlinear stage solver when required. Example in Section 9.7.1.

`BDF`. Select backward differentiation order, startup policy and maximum step ratio. Keep its history for continuation. Example in Section 9.7.2.

`AdaptiveTimeStep`. Pass as run(time_step=...) to choose the trial step range, error tolerances, safety and rejection limits. Example in Section 9.7.2.

`TimeStepLimits`. Pass through Numerics.step_limits to bound the maximum step, stability safety factor and optional CFL. Example in Section 10.1.3.

`TimeExecutionBinding`. Read the bound method, nominal order, stage configuration, startup policy and stage-projection capabilities. Example in Section 10.1.3.

13.9 field_transfer selection index

See Section 9.8 for a worked comparison in this layer.

`GlobalBroadcastTransfer`. Send an owned global scalar or vector to a compatible spatial input as a uniform value. Example in Section 10.2.19.

`SpatialAverageTransfer`. Integrate a native spatial field and divide by its physical measure using the selected quadrature. Example in Section 10.2.19.

`GlobalValueTransfer`. Send a global value to another compatible global input without a spatial sampling step. Example in Section 9.9.1.

`FiniteElementEvaluation`. Evaluate an FE field on the receiver's requested physical points, such as mechanical quadrature. Example in Section 5.13.1.

`CellToQuadratureProjection`. Map native cell data to the receiving quadrature support through the bound evaluator. Example in Section 10.2.19.

`FieldInterpolation`. Interpolate a field to a compatible receiving support using the selected transfer factory. Example in Section 10.2.19.

`ElectricFieldToFaces`. Provide the electric field on target faces, with an explicit periodic potential trace when needed. Example in Section 10.2.20.

`PeriodicPotentialTrace`. Set the signed potential increase from the first periodic face to the second. Example in Section 10.2.20.

`ConservativeVolumeTransfer`. Integrate an FE scalar density over physical FV overlaps. Configure coverage and quadrature tolerances for the receiving cells. Example in Section 10.2.21.

13.10 coupled_systems selection index

See Section 9.9 for a worked comparison in this layer.

`AlgebraicClosure`. Resolve instantaneous field feedback with a bound nonlinear solver. Use source units for initial guesses. Example in Section 10.2.1.

`MonolithicCoupling`. Advance unique transient blocks with one time algorithm, explicit stage solvers where needed, and shared stability. Example in Section 9.9.1.

`PartitionedCoupling`. Replay one trial step per transient partition while iterating exchanged fields. Nonconvergence rejects the step. Example in Section 9.9.1.

`SequentialCoupling`. Give an explicit sweep order containing each declared transient or stationary model exactly once. Example in Section 10.2.2.

`QuasiStationaryCoupling`. Iterate nonoverlapping transient and stationary blocks at a shared trial time. Example in Section 10.2.27.

`PISO`. Configure a supported FV pressure-correction solve with momentum and pressure solvers, correctors and relaxation. Example in Section 10.2.13.

`CoupledVelocityPressure`. Solve velocity, face flux and pressure together with the selected inner solvers. Example in Section 10.2.13.

APPENDIX A

Physics API reference

Each section gives the import package. For example, use `from onesim.physics.heat_transfer import HeatConduction`. The signatures show exposed arguments and defaults. Angle brackets identify a default supplied by a factory or a callable object. Type names are shortened for readability. A declared field is accessed through an explicit model property or `model.field(name)`. Some fields are created per species or component and are available through instance methods. Inspect `model.fields()` on the configured model. All sixteen domain package exports are included.

A.1 onesim.physics

`acoustics` namespace

onesim.physics.acoustics public components; compatibility aliases are not canonical exports.

`audit_equation_execution_binding` function

```
audit_equation_execution_binding(provider: object, expected: Sequence[EquationExecutionContract
    ], *, require_acknowledgement: bool = False) -> str
```

Validate an optional provider acknowledgement of residual identities.

`audit_term_execution_binding` function

```
audit_term_execution_binding(provider: object, expected: Sequence[TermExecutionContract], *,
    require_acknowledgement: bool = False) -> str
```

Audit provider acknowledgement of declared term identities.

`bind_explicit_spatial_runtime` function

```
bind_explicit_spatial_runtime(system: object, model: object, *, context: Mapping[str, object]
    | None = None) -> object
```

Return the explicitly declared common spatial operator for a system.

`bind_system` function

```
bind_system(system: object, model: object, *, runtime_registry: MethodologyRuntimeRegistry |
    None = None) -> object
```

Bind one system through an explicit adapter or generic provider port.

`biophysics` namespace

onesim.physics.biophysics public components; compatibility aliases are not canonical exports.

`can_lower_system_conditions` function

```
can_lower_system_conditions(system: object) -> bool
```

Return whether a system condition is safe for the common Study IR.

`chemistry` namespace

onesim.physics.chemistry public components; compatibility aliases are not canonical exports.

`compile_equation` function

```
compile_equation(equation: object) -> EquationExecutionKernel
```

Compile an explicit `EquationSpec.formulation.residual_form`.

`CustomPhysicsModelService` class

```
CustomPhysicsModelService()
```

Single validation path for all custom equation and interface contracts.

Selected methods.

```
validate(*, equations: tuple[Mapping[str, Any], ...], packages: tuple[Mapping[str, Any], ...],
    variables: tuple[Mapping[str, Any], ...], domain_ids: tuple[str, ...]) ->
    CustomPhysicsValidation
```

`DEFAULT_PHYSICS_TEMPLATE_REGISTRY` value

Read the default physics template registry when selecting reusable declaration templates by ID.

`derive_physics_ir` function

```
derive_physics_ir(model: PhysicsModel) -> PhysicsIR
```

Normalize equations and terms without selecting a numerical provider.

`differentiate_equation` function

```
differentiate_equation(equation: object, variable: str) -> EquationExecutionKernel
```

Compile one symbolic Jacobian entry from an equation declaration.

`electromagnetics` namespace

onesim.physics.electromagnetics public components; compatibility aliases are not canonical exports.

`electrostatics` namespace

onesim.physics.electrostatics public components; compatibility aliases are not canonical exports.

`EquationExecutionError` class

Raised when a declared residual cannot be safely compiled.

`EquationExecutionKernel` class

```
EquationExecutionKernel(equation_id: str, source: str, normalized: str, variable_names:
    tuple[str, ...], latex: str, python_code: str, _expression: object, _evaluator:
    Callable[..., object]) -> None
```

Compiled scalar residual with symbolic derivative and code views.

Selected methods.

```
evaluate(values: Mapping[str, object]) -> object
```

Evaluate the compiled residual against named scalar values.

```
derivative(variable: str) -> "'EquationExecutionKernel'"
```

Return the exact symbolic derivative with respect to one variable.

```
real_coordinate_components(coordinates: Mapping[str, tuple[str, ...]])
```

Compile exact real/imaginary residuals in explicitly named coordinates.

```
jacobian(variables: tuple[str, ...] | None = None) -> "tuple['EquationExecutionKernel', ...]"
```

Return deterministic scalar Jacobian entries in variable order.

```
to_mapping() -> dict[str, object]
```

Expose only provider-neutral, reproducible compilation metadata.

`ExplicitSystemBoundary` class

```
ExplicitSystemBoundary(condition_id: str, field_id: str, boundary_ids: tuple[str, ...], kind:
    str, expression: str, unit: str, pair_condition_id: str | None = None, dependencies:
    tuple[str, ...] = (), metadata: Mapping[str, str] = (), parameters: Mapping[str, float] =
    (), domain_ids: tuple[str, ...] = ()) -> None
```

Builder-compatible boundary declaration produced from `System.bc`.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Expose the same stable shape as :class:BoundaryConditionIR.

`ExplicitSystemInitial` class

```
ExplicitSystemInitial(initial_id: str, field_id: str, domain_ids: tuple[str, ...], expression:
    str, unit: str, value_kind: str = 'expression', dependencies: tuple[str, ...] = (),
    metadata: Mapping[str, str] = (), parameters: Mapping[str, float] = ()) -> None
```

Builder-compatible initial declaration produced from `System.bc`.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Expose the same stable shape as :class:InitialConditionIR.

`export_physics_package` function

```
export_physics_package(payload: Mapping[str, Any]) -> dict[str, object]
```

Serialize the declared physics package for portability using the package contract.

`generate_equation_code` function

```
generate_equation_code(equation: object) -> str
```

Generate inspectable Python expression code for a safe residual.

`geotechnics` namespace

onesim.physics.geotechnics public components; compatibility aliases are not canonical exports.

`heat_transfer` namespace

onesim.physics.heat_transfer public components; compatibility aliases are not canonical exports.

`hydrodynamics` namespace

onesim.physics.hydrodynamics public components; compatibility aliases are not canonical exports.

`import_physics_package` function

```
import_physics_package(payload: Mapping[str, Any]) -> dict[str, object]
```

Load a serialized physics package using the package contract.

`magnetohydrodynamics` namespace

onesim.physics.magnetohydrodynamics public components; compatibility aliases are not canonical exports.

`magnetostatics` namespace

onesim.physics.magnetostatics public components; compatibility aliases are not canonical exports.

`MethodologyDescriptor` class

```
MethodologyDescriptor(name: str, package: str, capabilities: frozenset[str] = frozenset(),
    requirements: frozenset[str] = frozenset(), version: str = '1') -> None
```

Stable, provider-free descriptor shared by all scientific domains.

`MethodologyRegistry` class

```
MethodologyRegistry(descriptors: Iterable[MethodologyDescriptor] = ()) -> None
```

Register, examine, and match methodology descriptors without special cases.

Selected methods.

```
register(descriptor: MethodologyDescriptor, *, replace: bool = False) -> None
```

```
get(name: str) -> MethodologyDescriptor
```

```
names() -> tuple[str, ...]
```

```
match(capabilities: Iterable[str], requirements: Iterable[str] = ()) ->
    tuple[MethodologyDescriptor, ...]
```

`multibody` namespace

onesim.physics.multibody public components; compatibility aliases are not canonical exports.

`multiphysics` namespace

onesim.physics.multiphysics public components; compatibility aliases are not canonical exports.

`MultiphysicsGraphService` class

```
MultiphysicsGraphService()
```

The single pre-solve planner for transfers, frames, and moving domains.

Selected methods.

```
validate(*, physics: tuple[PhysicsDescriptor, ...], variables: tuple[VariableDescriptor, ...],
    domain_ids: tuple[str, ...], assignments: tuple[DomainAssignment, ...], materials:
    tuple[MaterialDefinition, ...], couplings: tuple[CouplingSpec, ...], reference_frames:
    tuple[dict[str, Any], ...] = (), moving_domains: tuple[dict[str, Any], ...] = (),
    mesh_reference: dict[str, Any] | None = None) -> MultiphysicsGraphValidation
```

`optics` namespace

onesim.physics.optics public components; compatibility aliases are not canonical exports.

PhysicsTemplateDescriptor class

```
PhysicsTemplateDescriptor(identity: PhysicsTemplateIdentity, package: str, capabilities:
    frozenset[str]) -> None
```

Describe a registered physics template and its declared identity and metadata.

PhysicsTemplateRegistry class

```
PhysicsTemplateRegistry(descriptors: tuple[PhysicsTemplateDescriptor, ...] =
    (PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='biophysics',
    variant='bioelectrical_systems'), package='bioelectrical_systems',
    capabilities=frozenset({'biophysics', 'bioelectrical_systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='biophysics',
    variant='circulatory_systems'), package='circulatory_systems',
    capabilities=frozenset({'biophysics', 'circulatory_systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='biophysics',
    variant='organ_models'), package='organ_models', capabilities=frozenset({'biophysics',
    'organ_models'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='bioph
    ysics', variant='tissue_mechanics'), package='tissue_mechanics',
    capabilities=frozenset({'biophysics', 'tissue_mechanics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='chemical',
    variant='batteries'), package='batteries', capabilities=frozenset({'chemical',
    'batteries'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='chemical
    ', variant='corrosion'), package='corrosion', capabilities=frozenset({'chemical',
    'corrosion'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='chemical
    ', variant='electrochemistry'), package='models', capabilities=frozenset({'chemical',
    'electrochemistry'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='c
    hemical', variant='fuel_cells'), package='fuel_cells', capabilities=frozenset({'chemical',
    'fuel_cells'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='chemica
    l', variant='mass_transport'), package='mass_transport',
     capabilities=frozenset({'chemical', 'mass_transport'})),
     PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='chemical',
    variant='reaction_systems'), package='reaction_systems',
     capabilities=frozenset({'chemical', 'reaction_systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electromagnetics',
    variant='circuits'), package='models', capabilities=frozenset({'electromagnetics',
    'circuits'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electroma
    gnetics', variant='electric_fields'), package='models',
     capabilities=frozenset({'electric_fields', 'electromagnetics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electromagnetics',
    variant='electric_machines'), package='electric_machines',
     capabilities=frozenset({'electromagnetics', 'electric_machines'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electromagnetics',
    variant='high_frequency_em'), package='high_frequency_em',
     capabilities=frozenset({'electromagnetics', 'high_frequency_em'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electromagnetics',
    variant='low_frequency_em'), package='low_frequency_em',
     capabilities=frozenset({'low_frequency_em', 'electromagnetics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='electromagnetics',
    variant='magnetic_fields'), package='models', capabilities=frozenset({'magnetic_fields',
    'electromagnetics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='e
    lectromagnetics', variant='power_electronics'), package='power_electronics',
    capabilities=frozenset({'electromagnetics', 'power_electronics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics',
    variant='compressible_flow'), package='compressible_flow',
     capabilities=frozenset({'compressible_flow', 'fluidics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics',
    variant='free_surface_flow'), package='free_surface_flow',
     capabilities=frozenset({'free_surface_flow', 'fluidics'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics',
    variant='incompressible_flow'), package='incompressible_flow',
    capabilities=frozenset({'pressure_velocity_coupling', 'viscous_diffusion',
    'body_force_source', 'fluidics', 'incompressibility_constraint', 'incompressible_flow'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics',
    variant='multiphase_flow'), package='multiphase_flow',
```

```
 capabilities=frozenset({'multiphase_flow', 'fluidics'})),
 PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics',
variant='porous_flow'), package='porous_flow', capabilities=frozenset({'porous_flow',
'fluidics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='fluidics'
, variant='reactive_flow'), package='reactive_flow', capabilities=frozenset({'fluidics',
'reactive_flow'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='geot
echnics', variant='groundwater'), package='groundwater',
 capabilities=frozenset({'groundwater', 'geotechnics'})),
 PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='geotechnics',
variant='poromechanical_systems'), package='poromechanical_systems',
capabilities=frozenset({'poromechanical_systems', 'geotechnics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='geotechnics',
variant='reservoir'), package='reservoir', capabilities=frozenset({'reservoir',
'geotechnics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='geotec
hnics', variant='rock_mechanics'), package='rock_mechanics',
capabilities=frozenset({'rock_mechanics', 'geotechnics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='geotechnics',
variant='soil_mechanics'), package='soil_mechanics',
 capabilities=frozenset({'geotechnics', 'soil_mechanics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multibody',
variant='drivetrains'), package='models', capabilities=frozenset({'multibody',
'drivetrains'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multib
ody', variant='joint_systems'), package='joint_systems',
 capabilities=frozenset({'multibody', 'joint_systems'})),
 PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multibody',
variant='kinematics'), package='models', capabilities=frozenset({'kinematics',
'multibody'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multibod
y', variant='mechanisms'), package='models', capabilities=frozenset({'multibody',
'mechanisms'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multibo
dy', variant='rigid_body_dynamics'), package='rigid_body_dynamics',
capabilities=frozenset({'multibody', 'rigid_body_dynamics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multiphysics',
variant='electro_thermal'), package='electro_thermal',
 capabilities=frozenset({'electro_thermal', 'multiphysics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multiphysics',
variant='fluid_structure'), package='models', capabilities=frozenset({'fluid_structure',
'multiphysics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multi
physics', variant='general_coupled_subsystems'), package='general_coupled_subsystems',
capabilities=frozenset({'multiphysics', 'general_coupled_subsystems'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multiphysics',
variant='magneto_mechanics'), package='magneto_mechanics',
 capabilities=frozenset({'magneto_mechanics', 'multiphysics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multiphysics',
variant='reactive_coupling'), package='reactive_coupling',
 capabilities=frozenset({'reactive_coupling', 'multiphysics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='multiphysics',
variant='thermo_mechanics'), package='thermo_mechanics',
 capabilities=frozenset({'thermo_mechanics', 'multiphysics'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='plasma',
variant='discharges'), package='discharges', capabilities=frozenset({'plasma',
'discharges'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='plasma'
, variant='low_temperature_plasma'), package='low_temperature_plasma',
capabilities=frozenset({'plasma', 'low_temperature_plasma'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='plasma',
variant='plasma_field_models'), package='plasma_field_models',
capabilities=frozenset({'plasma', 'plasma_field_models'})),
PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='plasma',
variant='plasma_wall_interaction'), package='plasma_wall_interaction',
```

```
    capabilities=frozenset({'plasma', 'plasma_wall_interaction'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structural',
    variant='contact'), package='models', capabilities=frozenset({'contact', 'structural'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structural',
    variant='dynamics'), package='models', capabilities=frozenset({'structural',
    'dynamics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structura
    l', variant='fatigue'), package='models', capabilities=frozenset({'structural',
    'fatigue'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structural
    ', variant='fracture_damage'), package='fracture_damage',
     capabilities=frozenset({'structural', 'fracture_damage'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structural',
    variant='nonlinearity'), package='nonlinearity', capabilities=frozenset({'nonlinearity',
    'structural'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structu
    ral', variant='statics'), package='statics', capabilities=frozenset({'structural',
    'statics'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='structural
    ', variant='vibration'), package='vibration', capabilities=frozenset({'structural',
    'vibration'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems'
    , variant='control_signal_flow'), package='control_signal_flow',
    capabilities=frozenset({'control_signal_flow', 'systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems',
    variant='hydraulic_systems'), package='hydraulic_systems',
     capabilities=frozenset({'hydraulic_systems', 'systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems',
    variant='pneumatic_systems'), package='pneumatic_systems',
     capabilities=frozenset({'pneumatic_systems', 'systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems',
    variant='process_models'), package='process_models',
     capabilities=frozenset({'process_models', 'systems'})),
     PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems',
    variant='thermal_networks'), package='thermal_networks',
     capabilities=frozenset({'thermal_networks', 'systems'})),
     PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='systems',
    variant='zero_d_one_d_models'), package='zero_d_one_d_models',
    capabilities=frozenset({'zero_d_one_d_models', 'systems'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='thermal',
    variant='heat_conduction'), package='heat_conduction', capabilities=frozenset({'thermal',
    'volumetric_source', 'heat_conduction', 'conductive_diffusion'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='thermal',
    variant='heat_transport'), package='heat_transport', capabilities=frozenset({'thermal',
    'heat_transport'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='the
    rmal', variant='phase_change'), package='models', capabilities=frozenset({'phase_change',
    'thermal'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='thermal',
    variant='radiation'), package='models', capabilities=frozenset({'radiation', 'thermal'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='thermal',
    variant='thermal_system_models'), package='thermal_system_models',
    capabilities=frozenset({'thermal', 'thermal_system_models'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
    variant='acoustics'), package='acoustics', capabilities=frozenset({'acoustics',
    'waves'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
    variant='optics'), package='optics', capabilities=frozenset({'optics', 'waves'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
    variant='photonics'), package='photonics', capabilities=frozenset({'photonics',
    'waves'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
    variant='structural_acoustics'), package='structural_acoustics',
    capabilities=frozenset({'structural_acoustics', 'waves'})),
    PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
    variant='ultrasound'), package='ultrasound', capabilities=frozenset({'ultrasound',
    'waves'})), PhysicsTemplateDescriptor(identity=PhysicsTemplateIdentity(domain='waves',
```

```
    variant='vibroacoustics'), package='vibroacoustics',
     capabilities=frozenset({'vibroacoustics', 'waves'}))))) -> None
```

Read-only selection service for physics templates.

Selected methods.

```
all() -> tuple[PhysicsTemplateDescriptor, ...]
```

```
resolve(domain: str, variant: str) -> PhysicsTemplateDescriptor
```

```
supports(domain: str, variant: str, required: frozenset[str]) -> bool
```

plasma namespace

onesim.physics.plasma public components; compatibility aliases are not canonical exports.

ProblemIRBuilder class

```
ProblemIRBuilder()
```

Translate explicit equation/field/term declarations into `ProblemIR`.

Selected methods.

```
build(equations: object, *, problem_id: str, study_kind: str = 'stationary',
    domain_dimensions: Mapping[str, int] | None = None, coordinate_systems: Mapping[str, str]
    | None = None, mesh_region_ids: Mapping[str, str] | None = None, parameters:
    Sequence[ParameterIR] = (), boundaries: Sequence[object] = (), initial_conditions:
    Sequence[object] = (), sources: Sequence[object] = (), constraints: Sequence[object] = (),
    events: Sequence[object] = (), interfaces: Sequence[object] = (), materials:
    Sequence[object] = (), chemistry: object | Sequence[object] | None = None,
    equation_owners: Mapping[str, str] | None = None, dependency_owners: Mapping[tuple[str,
    str], str] | None = None, condition_owners: Mapping[tuple[str, str], str] | None = None)
    -> ProblemIR
```

ProblemIRBuildError class

Raised when a public declaration is not semantically complete.

REFERENCE_PHYSICS_INTERFACES value

Reference interface catalog for physics capability and contract inspection.

solid_mechanics namespace

onesim.physics.solid_mechanics public components; compatibility aliases are not canonical exports.

SystemConditionTranslation class

```
SystemConditionTranslation(boundaries: tuple[ExplicitSystemBoundary, ...] = (), initials:
    tuple[ExplicitSystemInitial, ...] = (), *, condition_owners: Mapping[tuple[str, str], str]
    = <factory>) -> None
```

All explicitly lowerable conditions for a set of systems.

SystemConditionTranslationError class

Raised when a compact system condition lacks executable semantics.

systems namespace

onesim.physics.systems public components; compatibility aliases are not canonical exports.

translate_system_conditions function

```
translate_system_conditions(systems: Sequence[object], *, selection_map: Mapping[str,
    Sequence[str]] | None = None) -> SystemConditionTranslation
```

Translate only conditions carrying complete portable semantics.

A.2 onesim.physics.acoustics

AcousticFeatureSpec class

```
AcousticFeatureSpec(feature_id: str, name: str, node_kind: AcousticNodeKind, model:
    AcousticModelKind, domain_ids: tuple[str, ...], boundary_ids: tuple[str, ...], parameters:
    tuple[tuple[str, float], ...]) -> None
```

Declare a named feature with its acoustic model kind, support IDs and parameters.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'AcousticFeatureSpec'"
```

```
to_mapping() -> dict[str, object]
```

AcousticModelKind enumeration

```
AcousticModelKind(*values)
```

Select the acoustic model family in a system or feature specification.

AcousticNodeKind enumeration

```
AcousticNodeKind(*values)
```

Select the role of a feature in an acoustic system specification.

AcousticsConfiguration class

```
AcousticsConfiguration(provider: str = 'waves.acoustics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'pressure_velocity_finite_difference
    ', time_integration: str = 'classical_rk4', solver: str = 'linear_acoustic_wave',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select Core numerical contracts and acoustic parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate configuration to the shared Core boundary.

AcousticsProvider class

```
AcousticsProvider(*, cell_count: int = 64, density: float = 1.0, sound_speed: float = 1.0,
    velocity_damping: float = 0.0, pressure_source: FieldSource = 0.0, velocity_source:
    FieldSource = 0.0, initial_pressure: np.ndarray | None = None, initial_velocity:
    np.ndarray | None = None, final_time: float = 0.0, time_step: float = 0.0001,
    time_integration: str = 'classical_rk4', bounds: tuple[float, float] | None = None,
    boundary_bounds: tuple[float, float] | None = None, periodic: bool = True, **_: object) ->
    None
```

Integrate linear pressure–velocity acoustics through Core numerics.

Selected methods.

```
spatial_factory_metadata_for(declaration, *, backend_id)
```

```
create_spatial_operator(declaration, context)
```

```
execute(problem: object | None = None) -> AcousticsResult
```

Advance pressure and particle velocity with the shared time integrator.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.acoustics', kernel_choice: str =
    'auto') -> "'AcousticsSpatialOperator'"
```

Expose the validated acoustics RHS through the common Study port.

`AcousticsPublicApi` class

```
AcousticsPublicApi(*, provider_registry=None) -> None
```

Validate and execute acoustic cases through the waves registry.

Selected methods.

```
validate(c: AcousticsConfiguration) -> tuple[str, ...]
```

```
run(c: AcousticsConfiguration)
```

Validate and execute the selected acoustic provider.

`AcousticsResult` class

```
AcousticsResult(coordinates: np.ndarray, state: AcousticsState, time: float, step_count: int,
    acoustic_energy: float, pressure_residual: float, velocity_residual: float) -> None
```

Final acoustic fields, energy and discrete wave-equation residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied fields and wave-balance diagnostics.

A direct acoustic rate calculation appears in Section 6.4.9.

`AcousticsSpatialOperator` class

```
AcousticsSpatialOperator(provider: AcousticsProvider, *, method_id: str =
    'spatial-method:fdm.acoustics', kernel_choice: str = 'auto') -> None
```

Common-runtime adapter for the two scalar acoustic fields.

Selected methods.

```
bind_initial_time(time)
```

```
state_field_dof_indices(state)
```

```
state_field_layout(problem, state)
```

```
fresh_runtime_operator()
```

```
residual(state: object, context: object)
```

Evaluate sources at the actual stage time supplied by the runner.

```
real_jacobian(state: object, context: object)
```

```
stable_step(state: object, context: object) -> float | None
```

```
courant_rate(state: object, context: object) -> float
```

Acoustic characteristic c/dx, independent of time-solver safety.

```
project_state(state: object, context: object) -> np.ndarray
```

```
bind_conditions(*, problem: object, initial_state: object | None) -> object
```

Bind pressure/velocity conditions through the common field kernel.

```
bind_field_conditions(*, problem: object, boundary_sides: object, boundary_value_resolver:
    object | None = None) -> object
```

Apply physical face values to the acoustic edge derivatives.

```
reconstruction(state: object, context: object)
```

```
result_carrier_ids(state: object, context: object)
```

```
contribute(request: object)
```

`AcousticsState` class

```
AcousticsState(pressure: np.ndarray, velocity: np.ndarray) -> None
```

Acoustic pressure and particle-velocity fields on a Core 1D mesh.

`AcousticSystemService` class

```
AcousticSystemService(*, provider_registry=None) -> None
```

Validate one acoustic Physics tree and execute registered providers.

Selected methods.

```
validate(*, systems: tuple[Mapping[str, object], ...], physics_ids: tuple[str, ...],
    domain_ids: tuple[str, ...], boundary_ids: tuple[str, ...], mesh_reference: Mapping[str,
    object] | None) -> AcousticSystemValidation
```

`AcousticSystemSpec` class

```
AcousticSystemSpec(system_id: str, name: str, physics_id: str, domain_ids: tuple[str, ...],
    dimension: int, features: tuple[AcousticFeatureSpec, ...]) -> None
```

Compose an acoustic system ID, geometry domain IDs, dimension and feature declarations.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'AcousticSystemSpec'"
```

```
to_mapping() -> dict[str, object]
```

`AcousticSystemValidation` class

```
AcousticSystemValidation(diagnostics: tuple[str, ...], normalized_systems: tuple[dict[str,
    object], ...], provider_id: str = 'onesimulation.internal.acoustics') -> None
```

Validation diagnostics and executable normalized acoustic plans.

Selected methods.

```
to_mapping() -> dict[str, object]
```

`adapt_acoustics_result` function

```
adapt_acoustics_result(result: AcousticsResult, *, result_id: str = 'waves.acoustics.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish 1D acoustic pressure/velocity fields and wave residuals.

`adapt_structural_acoustics_result` function

```
adapt_structural_acoustics_result(result: StructuralAcousticsResult, *, result_id: str =
    'waves.structural_acoustics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish interface motion and delegated acoustic fields.

`adapt_ultrasound_result` function

```
adapt_ultrasound_result(result: UltrasoundResult, *, result_id: str =
    'waves.ultrasound.result', plan_reference: ArtifactReference | None = None, diagnostics:
    Sequence[str] = ()) -> SimulationResultSet
```

Publish acoustic intensity and absorbed-power density fields.

`adapt_vibroacoustics_result` function

```
adapt_vibroacoustics_result(result: VibroacousticsResult, *, result_id: str =
    'waves.vibroacoustics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish modal vibroacoustic response and delegated acoustic fields.

`LinearAcoustics` class

```
LinearAcoustics(*, domain: object, model_id: str = <factory>, density: float, sound_speed:
    float, velocity_damping: float = 0.0, pressure_source: object = 0.0, force_density: object
    = 0.0) -> None
```

Evolve linear pressure and particle velocity on a line. Initialize both fields and supply physical boundary traces or a periodic pair.

Declared fields. `pressure` (Pa, differential), `velocity` (m/s, differential), `compression_source` (Pa/s, input), `body_force` (N/m^3, input).

Selected methods.

```
reflect_boundary_state(values, *, normal)
```

```
reflect_boundary_state_jacobian(values, *, normal)
```

`PressureDrivenMode` class

```
PressureDrivenMode(*, domain: object, model_id: str = <factory>, mass: float, stiffness:
    float, area: float, damping: float = 0.0, pressure_loading: object = 0.0) -> None
```

Drive one mechanical acoustic mode by pressure times effective area. Initialize displacement and velocity.

Declared fields. `displacement` (m, differential), `velocity` (m/s, differential), `energy` (J, derived), `pressure` (Pa, input).

`RadiatingMode` class

```
RadiatingMode(*, domain: object, model_id: str = <factory>, mass: float, stiffness: float,
    area: float, damping: float = 0.0, pressure_loading: object = 0.0, radiation_resistance:
    float) -> None
```

Constant-resistance approximation; no radiation reactance or far field.

Declared fields. `displacement` (m, differential), `velocity` (m/s, differential), `energy` (J, derived), `pressure` (Pa, input), `radiated_power` (W, derived).

`StructuralAcousticsConfiguration` class

```
StructuralAcousticsConfiguration(provider: str = 'waves.structural_acoustics', mesh: str =
    'delegated_acoustics_plus_lumped_mode', discretization: str =
    'one_way_acoustic_interface', time_integration: str = 'classical_rk4', solver: str =
    'acoustics_provider_composition', parameters: Mapping[str, object] = <factory>) -> None
```

Select acoustic delegation and modal structural parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to shared Core numerical configuration.

`StructuralAcousticsProvider` class

```
StructuralAcousticsProvider(*, area: float = 1.0, modal_mass: float = 1.0, modal_stiffness:
    float = 1.0, modal_damping: float = 0.0, initial_displacement: float = 0.0,
    initial_velocity: float = 0.0, final_time: float = 0.0, time_step: float = 0.001,
    time_integration: str = 'classical_rk4', **acoustics_parameters: object) -> None
```

Drive a structural interface mode from the existing acoustic solver output.

Selected methods.

```
execute(problem: object | None = None) -> StructuralAcousticsResult
```

Integrate the uncoupled acoustic field and interface mode together.

`StructuralAcousticsPublicApi` class

```
StructuralAcousticsPublicApi(*, provider_registry=None) -> None
```

Validate and execute structural acoustics.

Selected methods.

```
validate(c: StructuralAcousticsConfiguration) -> tuple[str, ...]
```

```
run(c: StructuralAcousticsConfiguration)
```

Validate and execute provider.

`StructuralAcousticsResult` class

```
StructuralAcousticsResult(state: StructuralAcousticsState, acoustics_result: AcousticsResult,
    area: float, time: float, steps: int, residual: float) -> None
```

Delegated acoustic result and pressure-driven structural interface mode.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return interface state, force and delegated acoustic solution.

`StructuralAcousticsState` class

```
StructuralAcousticsState(displacement: float, velocity: float) -> None
```

Interface modal displacement and velocity.

`UltrasonicAcoustics` class

```
UltrasonicAcoustics(*, domain: object, model_id: str = <factory>, density: float, sound_speed:
    float, velocity_damping: float = 0.0, pressure_source: object = 0.0, force_density: object
    = 0.0) -> None
```

Select a linear damped wave model with the acoustic pressure and velocity fields.

Declared fields. `pressure` (Pa, differential), `velocity` (m/s, differential), `compression_source` (Pa/s, input), `body_force` (N/m^3, input), `intensity` (W/m^2, derived), `dissipation` (W/m^3, derived).

`UltrasoundConfiguration` class

```
UltrasoundConfiguration(provider: str = 'waves.ultrasound', mesh: str =
    'delegated_acoustics.1d', discretization: str = 'acoustic_intensity_absorption',
    time_integration: str = 'classical_rk4', solver: str = 'acoustics_provider_composition',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select delegated acoustic contracts and ultrasound absorption.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to common Core configuration.

`UltrasoundProvider` class

```
UltrasoundProvider(*, absorption_coefficient: float = 0.0, **acoustic_parameters: object) ->
    None
```

Delegate wave propagation to Acoustics and calculate ultrasound observables.

Selected methods.

```
execute(problem: object | None = None) -> UltrasoundResult
```

Delegate acoustic solve and evaluate ultrasound intensity/absorption.

`UltrasoundPublicApi` class

```
UltrasoundPublicApi(*, provider_registry=None) -> None
```

Validate and execute ultrasound through waves registry.

Selected methods.

```
validate(c: UltrasoundConfiguration) -> tuple[str, ...]
```

```
run(c: UltrasoundConfiguration)
```

Validate and execute provider.

`UltrasoundResult` class

```
UltrasoundResult(state: UltrasoundState, acoustics: AcousticsResult, absorption_coefficient:
    float) -> None
```

Delegated acoustic state plus intensity and absorption diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return intensity/absorption fields and delegated acoustic data.

`UltrasoundState` class

```
UltrasoundState(absorbed_power_density: np.ndarray) -> None
```

Non-negative absorbed-power density derived from acoustic intensity.

`VibroacousticsConfiguration` class

```
VibroacousticsConfiguration(provider: str = 'waves.vibroacoustics', mesh: str =
    'delegated_structural_acoustics', discretization: str = 'modal_radiation_impedance',
    time_integration: str = 'classical_rk4', solver: str = 'structural_acoustics_provider_compo
    sition', parameters: Mapping[str, object] = <factory>) -> None
```

Select structural-acoustic delegation and radiation impedance.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to common Core configuration.

`VibroacousticsProvider` class

```
VibroacousticsProvider(*, radiation_resistance: float = 0.0,
     **structural_acoustics_parameters: object) -> None
```

Delegate coupled vibration/acoustics and evaluate radiation impedance output.

Selected methods.

```
execute(problem: object | None = None) -> VibroacousticsResult
```

Delegate interface response and calculate energy/radiated power.

`VibroacousticsPublicApi` class

```
VibroacousticsPublicApi(*, provider_registry=None) -> None
```

Validate and execute vibroacoustics through waves registry.

Selected methods.

```
validate(c: VibroacousticsConfiguration) -> tuple[str, ...]
```

```
run(c: VibroacousticsConfiguration)
```

Validate and execute provider.

`VibroacousticsResult` class

```
VibroacousticsResult(state: VibroacousticsState, structural_acoustics:
    StructuralAcousticsResult, modal_mass: float, modal_stiffness: float,
    radiation_resistance: float) -> None
```

Delegated interface response with mechanical and radiated acoustic power.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return vibroacoustic modal observables and delegated coupling data.

`VibroacousticsState` class

```
VibroacousticsState(displacement: float, velocity: float) -> None
```

Modal interface displacement and velocity inherited from structural acoustics.

A.3 onesim.physics.biophysics

`adapt_bioelectrical_result` function

```
adapt_bioelectrical_result(result: BioelectricalResult, *, result_id: str =
    'biophysics.bioelectrical_systems.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish membrane voltage, gates, ion currents, and current balance.

`adapt_bioheat_result` function

```
adapt_bioheat_result(result: BioheatResult, *, result_id: str = 'biophysics.bioheat.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish 1D Pennes temperature and thermal balance diagnostics.

`adapt_biophysics_result` function

```
adapt_biophysics_result(result: object, *, provider_id: str | None = None, result_id: str =
    'biophysics.result', plan_reference: ArtifactReference | None = None, diagnostics:
    Sequence[str] = ()) -> SimulationResultSet
```

Adapt any built-in biophysics result through one explicit entry point.

`adapt_circulatory_result` function

```
adapt_circulatory_result(result: CirculatoryResult, *, result_id: str =
    'biophysics.circulatory_systems.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish circulatory node pressures, segment flows, and balances.

`adapt_organ_result` function

```
adapt_organ_result(result: OrganResult, *, result_id: str = 'biophysics.organ_models.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish arbitrary organ/species concentrations and balances.

`adapt_tissue_result` function

```
adapt_tissue_result(result: TissueResult, *, result_id: str =
    'biophysics.tissue_mechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish tissue nodal kinematics, element forces, and mechanics balances.

`AxialTissue` class

```
AxialTissue(*, domain: object, model_id: str = <factory>, mass: float, element: TissueElement,
    force: object = 0.0) -> None
```

Integrate axial mechanical motion of a mass attached to a tissue element.

Declared fields. `displacement` (m, differential), `velocity` (m/s, differential), `strain` (1, derived), `element_force` (N, derived), `load` (N, input).

`BioelectricalConfiguration` class

```
BioelectricalConfiguration(provider: str = 'biophysics.bioelectrical_systems', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'explicit_block', parameters:
    Mapping[str, object] = <factory>) -> None
```

Select bioelectrical physics separately from shared numerics.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

`BioelectricalPublicApi` class

```
BioelectricalPublicApi(registry=None, *, provider_registry=None) -> None
```

Execute bioelectrical problems through the shared application boundary.

Selected methods.

```
validate(problem: InitialValueProblem, configuration: BioelectricalConfiguration) ->
    tuple[str, ...]
```

```
run(problem: InitialValueProblem, configuration: BioelectricalConfiguration)
```

`BioheatProvider` class

```
BioheatProvider(*, cell_count: int = 64, density: float = 1000.0, heat_capacity: float =
    3600.0, conductivity: float = 0.5, perfusion_capacity_rate: float = 0.0,
    blood_temperature: float = 310.15, metabolic_heat: FieldSource = 0.0, external_heat:
    FieldSource = 0.0, initial_temperature: np.ndarray | None = None, time_step: float =
    0.001, final_time: float = 0.0, time_integration: str = 'classical_rk4', **_: object) ->
    None
```

Integrate the Pennes tissue bioheat equation.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.bioheat') ->
    CallableSpatialOperator
```

Expose the Pennes RHS through the common transient Study.

```
execute(problem: object | None = None) -> BioheatResult
```

```
execute_simulation(request: SimulationExecutionRequest) -> BioheatResult
```

Execute the versioned problem-first request contract.

`BioheatResult` class

```
BioheatResult(coordinates: np.ndarray, temperature: np.ndarray, time: float, step_count: int,
    thermal_energy: float, perfusion_heat: float, balance_residual: float) -> None
```

Temperature field, heat inventory, and equation residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

`BiophysicsProviderRegistry` class

```
BiophysicsProviderRegistry() -> None
```

Register providers by stable physics identifier.

Selected methods.

```
register(provider_id: str, factory: Callable[..., object], *, catalog_role: str =
    'unclassified') -> None
```

```
catalog_role(provider_id: str) -> str
```

```
create(provider_id: str, **parameters: object) -> object
```

```
ids() -> tuple[str, ...]
```

`CirculatoryConfiguration` class

```
CirculatoryConfiguration(provider: str = 'biophysics.circulatory_systems', mesh: str =
    'core.lumped.0d', discretization: str = 'lumped_parameter', time_integration: str =
    'classical_rk4', solver: str = 'explicit_block', parameters: Mapping[str, object] =
    <factory>) -> None
```

Numerical selection for a circulatory network.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

`CirculatoryPublicApi` class

```
CirculatoryPublicApi(*, provider_registry=None) -> None
```

Execute circulatory problems through the shared application boundary.

Selected methods.

```
validate(problem: InitialValueProblem, configuration: CirculatoryConfiguration) -> tuple[str,
    ...]
```

```
run(problem: InitialValueProblem, configuration: CirculatoryConfiguration)
```

`CompliantVessels` class

```
CompliantVessels(*, domain: object, model_id: str = <factory>, upstream_compliance: float,
    downstream_compliance: float, vessel: VesselSegment, pump_flow: object = 0.0) -> None
```

Integrate two pressure compartments coupled by a resistive vessel.

Declared fields. `upstream_pressure` (Pa, differential), `downstream_pressure` (Pa, differential), `flow` (m^3/s, derived), `pump` (m^3/s, input).

`DEFAULT_BIOPHYSICS_PROVIDER_REGISTRY` value

Default registry for the domain provider route. Inspect the registered IDs when choosing a provider.

`OrganClearance` class

```
OrganClearance(*, domain: object, model_id: str = <factory>, volume: float, clearance: float,
    infusion_rate: object = 0.0) -> None
```

Integrate a well-mixed amount with elimination rate proportional to concentration and an optional infusion input.

Declared fields. `amount` (mol, differential), `concentration` (mol/m^3, derived), `infusion` (mol/s, input), `elimination_rate` (mol/s, derived).

`OrganConfiguration` class

```
OrganConfiguration(provider: str = 'biophysics.organ_models', mesh: str = 'core.lumped.0d',
    discretization: str = 'lumped_parameter', time_integration: str = 'classical_rk4', solver:
    str = 'explicit_block', parameters: Mapping[str, object] = <factory>) -> None
```

Numerical selection for generic organ compartments.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

`OrganPublicApi` class

```
OrganPublicApi(*, provider_registry=None) -> None
```

Execute organ transport with `C=A/V` and conservation diagnostics.

Selected methods.

```
validate(problem: InitialValueProblem, configuration: OrganConfiguration) -> tuple[str, ...]
```

```
run(problem: InitialValueProblem, configuration: OrganConfiguration)
```

`PassiveMembrane` class

```
PassiveMembrane(*, domain: object, model_id: str = <factory>, capacitance: float, conductance:
    float, reversal_potential: float, current_density: object = 0.0) -> None
```

Integrate voltage across a passive capacitive membrane with a leak conductance, reversal potential and injected current density.

Declared fields. `voltage` (V, differential), `leak_current` (A/m^2, derived), `current` (A/m^2, input).

`TissueConfiguration` class

```
TissueConfiguration(provider: str = 'biophysics.tissue_mechanics', mesh: str =
    'core.lumped.1d', discretization: str = 'lumped_finite_element', time_integration: str =
    'classical_rk4', solver: str = 'explicit_block', parameters: Mapping[str, object] =
    <factory>) -> None
```

Numerical selection for a tissue mechanics model.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

`TissueElement` class

```
TissueElement(element_id: str, first: int, second: int, stiffness: float, rest_length: float,
    damping: float = 0.0) -> None
```

Axial elastic/viscoelastic connection between two tissue nodes.

`TissuePublicApi` class

```
TissuePublicApi(*, provider_registry=None) -> None
```

Execute tissue mechanics with shared numerical contracts.

Selected methods.

```
validate(problem: InitialValueProblem, configuration: TissueConfiguration) -> tuple[str, ...]
```

```
run(problem: InitialValueProblem, configuration: TissueConfiguration)
```

`VesselSegment` class

```
VesselSegment(segment_id: str, upstream: str, downstream: str, resistance: float, one_way:
    bool = False) -> None
```

Resistive connection between two compliant pressure compartments.

A.4 onesim.physics.chemistry

`BatteryChemistry` class

```
BatteryChemistry(chemistry_id: str, open_circuit_voltage: Callable[[float, float], float],
    internal_resistance: Callable[[float, float], float], open_circuit_voltage_gradient:
    Callable[[float, float], object] | None = None, internal_resistance_gradient:
    Callable[[float, float], object] | None = None) -> None
```

Open-circuit and resistance closures for an arbitrary cell chemistry.

Selected methods.

```
to_mapping()
```

State-dependent constitutive laws are ports, not coordinate callbacks.

`Chemistry` class

```
Chemistry(*, network: object, kinetics: ChemistryKinetics | None = None, source: str | None =
    None) -> None
```

Bundle a reaction network, kinetic collaborator and provenance for a model that consumes chemistry.

Selected methods.

```
from_database(path, *, catalog, variant=None, version='latest', species=None,
    gas_temperature=None, kinetics=None)
```

Load an existing SQLite catalog through its loss-aware common IR adapter.

```
from_file(path, *, loader=None, kinetics=None)
```

```
bind_kinetics()
```

```
evaluate(*, densities, mean_electron_energy, gas, time, points=None)
```

```
to_mapping()
```

`ChemistryEvaluation` class

```
ChemistryEvaluation(*, reaction_rates: Mapping, energy_sources: Mapping = <factory>,
    rate_unit: str = '1/m^3/s', energy_unit: str = 'W/m^3') -> None
```

Reaction progress in m^-3 s^-1 and signed energy production in W m^-3.

`ChemistryJacobian` class

```
ChemistryJacobian(*, reaction_density_derivatives: object, reaction_energy_derivatives:
    object, energy_density_derivatives: object = <factory>, energy_energy_derivatives: object
    = <factory>) -> None
```

Carry the density and energy derivatives of reaction progress and energy production.

`ChemistryKinetics` protocol

```
ChemistryKinetics(*args, **kwargs)
```

Kinetic evaluation contract for the supplied reaction network. Use its rate and derivative interface.

Selected methods.

```
evaluate(*, network, densities, mean_electron_energy, gas, time)
```

`ElectrochemicalReaction` class

```
ElectrochemicalReaction(reaction_id: str, stoichiometry: Mapping[str, float],
    exchange_current_density: float, equilibrium_potential: Callable[[Mapping[str, float],
    float], float], electron_number: float = 1.0, transfer_coefficient: float = 0.5,
    equilibrium_concentration_gradient: Callable | None = None,
    equilibrium_temperature_derivative: Callable | None = None) -> None
```

Butler-Volmer reaction with arbitrary species stoichiometry.

Selected methods.

```
to_mapping()
```

Keep the concentration/temperature law's explicit runtime identity.

`LumpedBattery` class

```
LumpedBattery(*, domain: object, model_id: str = <factory>, chemistry: BatteryChemistry,
    charge_capacity: float, current: object, thermal_capacity: float, heat_transfer: float,
    ambient_temperature: float | None = 298.15) -> None
```

Advance charge and temperature balances with applied current and a chemistry closure.

Declared fields. `load_current` (A, input), `ambient` (K, input), `state_of_charge` (1, differential), `temperature` (K, differential), `terminal_voltage` (V, derived), `ohmic_heat` (W, derived).

`LumpedElectrode` class

```
LumpedElectrode(*, domain: object, model_id: str = <factory>, reactions:
    tuple[ElectrochemicalReaction, ...], area: float, volume: float, double_layer_capacitance:
    float, applied_current_density: object = 0.0, temperature: float | None = 298.15) -> None
```

Advance a global electrode state with double-layer capacitance and explicit electrochemical reaction declarations.

Selected methods.

```
concentration(species)
```

`ReactionIR` class

```
ReactionIR(reaction_id: str, reactants: tuple[tuple[str, int], ...], products:
    tuple[tuple[str, int], ...], rate_law: Mapping[str, object], reaction_type: str =
    'volume', energy_channels: tuple[Mapping[str, object], ...] = (), validity_range:
    tuple[float, float] | None = None, validity_coordinate: str = '', surface: Mapping[str,
    object] | None = None, provenance: str = '') -> None
```

One reaction channel with explicit stoichiometry and rate metadata.

Selected methods.

```
to_dict() -> dict[str, object]
```

```
from_dict(payload: Mapping[str, object]) -> "'ReactionIR'"
```

`ReactionKinetics` class

```
ReactionKinetics(*, domain: object, model_id: str = <factory>, network: ReactionNetworkIR,
    temperature: float | None = 298.15, species_sources: Mapping = <factory>) -> None
```

Advance the species of an explicit homogeneous reaction network with a thermal input and signed species sources.

Selected methods.

```
concentration(species)
```

```
source(species)
```

```
production(species)
```

`ReactionNetworkIR` class

```
ReactionNetworkIR(network_id: str, version: str, species: tuple[SpeciesIR, ...], reactions:
    tuple[ReactionIR, ...], variant: str = '', closure: str = 'strict', provenance: str = '',
    metadata: Mapping[str, object] = <factory>, strict_balance: bool = False) -> None
```

Versioned, validated reaction network independent of a chemistry provider.

Selected methods.

```
balance_diagnostics() -> tuple[str, ...]
```

```
to_dict() -> dict[str, object]
```

```
from_dict(payload: Mapping[str, object]) -> "'ReactionNetworkIR'"
```

`SpeciesIR` class

```
SpeciesIR(species_id: str, charge_number: int, mass_kg: float, formula: str = '', role: str =
    '', state: str = '', composition: tuple[tuple[str, int], ...] = (), aliases: tuple[str,
    ...] = (), transport: Mapping[str, object] = <factory>, provenance: str = '') -> None
```

Species metadata shared by gas, liquid, surface and bio chemistry.

Selected methods.

```
to_dict() -> dict[str, object]
```

```
from_dict(payload: Mapping[str, object]) -> "'SpeciesIR'"
```

`SpeciesTransport` class

```
SpeciesTransport(*, domain: object, model_id: str = <factory>, species:
    tuple[TransportSpecies, ...], velocity: float, sources: object = None) -> None
```

Advance a concentration for every declared species on a line with advection, diffusion, decay and prescribed sources.

Selected methods.

```
concentration(species)
```

```
supply(species)
```

`TransportSpecies` class

```
TransportSpecies(species_id: str, diffusivity: float | None = None, decay_rate: float = 0.0,
    diffusivity_property_id: str | None = None) -> None
```

Transport parameters for one chemical species.

Selected methods.

```
to_portable_mapping() -> dict[str, object]
```

Return the complete, versioned transport declaration for a study plan.

```
from_portable_mapping(payload: Mapping[str, object]) -> "'TransportSpecies'"
```

Rehydrate only this exact transport schema, without silent options.

A.5 onesim.physics.electromagnetics

`AveragedBuckConverter` class

```
AveragedBuckConverter(*, domain: object, model_id: str = <factory>, input_voltage: object,
    duty_cycle: object, inductance: float, capacitance: float, load_resistance: float,
    series_resistance: float = 0.0) -> None
```

Integrate inductor current and output voltage with an averaged duty-cycle drive and resistive load.

Declared fields. `supply` (V, input), `duty` (1, input), `inductor_current` (A, differential), `output_voltage` (V, differential), `input_current` (A, derived), `input_power` (W, derived), `output_power` (W, derived), `conduction_loss` (W, derived), `voltage_residual` (V, derived), `current_residual` (A, derived), `duty_cycle` (1, derived).

`CircularWaveguideMode` class

```
CircularWaveguideMode(name: str, polarization_angle: float = 0.0) -> None
```

Select a named circular waveguide mode and its polarization angle.

`cold_electron_permittivity_property` function

```
cold_electron_permittivity_property(property_id, *, electron_density='electron_density',
    collision_frequency='collision_frequency', magnetic_components=('Bx', 'By', 'Bz'),
    angular_frequency='omega', phasor_convention)
```

Declare a lab-frame cold-electron epsilon tensor with explicit phasors.

`DCMotor` class

```
DCMotor(*, domain: object, model_id: str = <factory>, resistance: float, inductance: float,
    motor_constant: float, inertia: float, viscous_friction: float = 0.0, applied_voltage:
    object = 0.0, load_torque: object = 0.0) -> None
```

Integrate electrical current and mechanical rotation under applied voltage and load torque.

Declared fields. `voltage` (V, input), `torque` (N*m, input), `current` (A, differential), `angular_velocity` (rad/s, differential), `angle` (rad, differential), `back_emf` (V, derived), `electromagnetic_torque` (N*m, derived), `electrical_power` (W, derived), `mechanical_power` (W, derived).

`FrequencyDomainMaxwell` class

```
FrequencyDomainMaxwell(*, domain: object, model_id: str = <factory>, frequency: float,
    material: object, phasor_convention: str, amplitude_convention: str, impressed_current:
    object = (0.0, 0.0, 0.0)) -> None
```

Solve for complex electric phasors and derive magnetic flux density and absorbed power at one frequency.

Declared fields. `electric_field` (V/m, algebraic, x/y/z), `magnetic_flux_density` (T, derived, x/y/z), `absorbed_power_density` (W/m^3, derived), `impressed_current_density` (A/m^2, input, x/y/z).

Selected methods.

```
components()
```

`ImpedanceBoundary` class

```
ImpedanceBoundary(*, boundary: object, physics: object, admittance: object,
    incident_electric_field: object = (0.0, 0.0, 0.0)) -> None
```

Local tangential admittance in S, with an incident electric field in V/m.

Selected methods.

```
bind_trace(*, model, mesh, space, facets)
```

`MagneticDiffusion` class

```
MagneticDiffusion(*, domain: object, model_id: str = <factory>, permeability: float,
    conductivity: float) -> None
```

Advance magnetic induction with diffusivity equal to the reciprocal of permeability times conductivity.

Declared fields. `magnetic_induction` (T, differential), `magnetic_energy_density` (J/m^3, derived).

`operators` namespace

Electromagnetics / operators.

`PerfectElectricConductor` class

```
PerfectElectricConductor(*, boundary: object, physics: object) -> None
```

Apply the PEC tangential electric-field condition on the selected boundary.

`SeriesRLC` class

```
SeriesRLC(*, domain: object, model_id: str = <factory>, resistance: float, inductance: float,
    capacitance: float, source_voltage: object = 0.0) -> None
```

Integrate charge and current for the series circuit with an applied supply voltage.

Declared fields. `supply` (V, input), `charge` (C, differential), `current` (A, differential), `capacitor_voltage` (V, derived), `stored_energy` (J, derived), `source_voltage` (V, derived), `resistor_voltage` (V, derived), `inductor_voltage` (V, derived), `voltage_residual` (V, derived).

`WaveguideModeBasis` class

```
WaveguideModeBasis(modes: tuple, amplitudes: tuple) -> None
```

Explicit propagating port basis and relative complex incident amplitudes.

Selected methods.

```
bind(*, incident_power, **arguments)
```

`WaveguidePort` class

```
WaveguidePort(*, boundary: object, physics: object, mode: object, incident_power: float) ->
    None
```

Attach a mode and incident power to the Maxwell model on the selected boundary.

A.6 onesim.physics.electrostatics

`FixedPotential` class

```
FixedPotential(*, boundary: object, physics: object, value: object) -> None
```

Apply an electrostatic potential trace in volts to the named boundary.

`NormalElectricDisplacement` class

```
NormalElectricDisplacement(*, boundary: object, physics: object, value: object) -> None
```

Prescribed outward D dot n, positive charge flux in C/m^2.

Selected methods.

```
electric_displacement(*, points, time)
```

`PoissonEquation` class

```
PoissonEquation(*, domain, permittivity, charge_density=None, charge_density_source=None,
    model_id=None)
```

Solve electrostatic potential from charge and permittivity, and evaluate electric field from the negative potential gradient.

Declared fields. `potential` (V, algebraic), `electric_field` (V/m, derived, x/y/z), `charge_density` (C/m^3, input).

`ZeroNormalElectricDisplacement` class

```
ZeroNormalElectricDisplacement(*, boundary: object, physics: object) -> None
```

Prescribe zero outward normal displacement on a Poisson boundary.

Selected methods.

```
electric_displacement(*, points, time)
```

A.7 onesim.physics.geotechnics

`adapt_geotechnics_result` function

```
adapt_geotechnics_result(result: object, *, provider_id: str | None = None, result_id: str =
    'geotechnics.result', plan_reference: ArtifactReference | None = None, diagnostics:
    Sequence[str] = ()) -> SimulationResultSet
```

Adapt any built-in geotechnics result through one explicit entry point.

adapt_groundwater_result function

```
adapt_groundwater_result(result: GroundwaterResult, *, result_id: str =
    'geotechnics.groundwater.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish hydraulic head, Darcy flux, and the storage balance residual.

adapt_poromechanical_result function

```
adapt_poromechanical_result(result: PoromechanicalResult, *, result_id: str =
    'geotechnics.poromechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish coupled displacement/pressure fields and Biot diagnostics.

adapt_reservoir_result function

```
adapt_reservoir_result(result: ReservoirResult, *, result_id: str =
    'geotechnics.reservoir.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish two-phase pressure, saturation, phase fluxes, and balances.

adapt_rock_mechanics_result function

```
adapt_rock_mechanics_result(result: RockMechanicsResult, *, result_id: str =
    'geotechnics.rock_mechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish rock-bar kinematics, stresses, plasticity, and failure data.

adapt_soil_mechanics_result function

```
adapt_soil_mechanics_result(result: SoilMechanicsResult, *, result_id: str =
    'geotechnics.soil_mechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish consolidation fields, stresses, flux, and balance diagnostics.

BiotConsolidation class

```
BiotConsolidation(*, domain: object, model_id: str = <factory>, youngs_modulus: float,
    poisson_ratio: float, biot_coefficient: float, specific_storage: float, permeability:
    float, fluid_viscosity: float, fluid_supply: object = 0.0, body_force: object = (0.0, 0.0,
    0.0)) -> None
```

Couple pore pressure and elastic displacement through fluid content and volumetric strain.

Declared fields. `fluid_content` (1, differential), `displacement` (m, algebraic, x/y/z), `pore_pressure` (Pa, algebraic), `darcy_flux` (m/s, derived, x/y/z), `volumetric_strain` (1, derived), `water_supply` (1/s, input), `force_density` (N/m^3, input, x/y/z).

BuckleyLeverettFlow class

```
BuckleyLeverettFlow(*, domain: object, model_id: str = <factory>, porosity: float,
    darcy_velocity: float, fractional_flow: object, water_source: object = 0.0) -> None
```

Advance water saturation from fractional flow with prescribed porosity, Darcy velocity and water source.

Declared fields. `water_saturation` (1, differential), `water_flux` (m/s, derived), `oil_flux` (m/s, derived), `water_exchange` (1/s, input).

`DEFAULT_GEOTECHNICS_PROVIDER_REGISTRY` value

Default registry for the domain provider route. Inspect the registered IDs when choosing a provider.

`DrainedBoundary` class

```
DrainedBoundary(*, boundary: object, physics: object, value: object = 0.0, factory: object =
    None) -> None
```

Set pore pressure on the draining boundary. The default value is zero.

`GeotechnicsProviderRegistry` class

```
GeotechnicsProviderRegistry() -> None
```

Register geotechnical providers by stable identifier.

Selected methods.

```
register(provider_id: str, factory: Callable[..., object]) -> None
```

```
create(provider_id: str, **parameters: object) -> object
```

```
ids() -> tuple[str, ...]
```

`GroundwaterConfiguration` class

```
GroundwaterConfiguration(provider: str = 'geotechnics.groundwater', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'explicit_diffusion', parameters:
    Mapping[str, object] = <factory>) -> None
```

Numerical selection for groundwater simulation.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

`GroundwaterFlow` class

```
GroundwaterFlow(*, domain: object, model_id: str = <factory>, conductivity: float,
    specific_storage: float, recharge: object = 0.0) -> None
```

Advance hydraulic head and compute Darcy flux with conductivity, storage and recharge.

Declared fields. `head` (m, differential), `darcy_flux` (m/s, derived), `water_supply` (1/s, input).

`GroundwaterProvider` class

```
GroundwaterProvider(*, cell_count: int = 64, conductivity: float = 1.0, storage: float = 1.0,
    recharge: float | Callable[[np.ndarray, float], np.ndarray] = 0.0, initial_head:
    np.ndarray | None = None, final_time: float = 0.0, time_step: float = 0.0001,
    time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate a transient confined-aquifer hydraulic-head equation.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.groundwater', coordinates:
    np.ndarray | None = None, gradient_operator: object | None = None, diffusion_operator:
    object | None = None) -> CallableSpatialOperator
```

Expose the groundwater RHS through the common transient Study.

```
execute(problem: object | None = None) -> "'GroundwaterResult'"
```

Advance hydraulic head and return Darcy flux/storage diagnostics.

`GroundwaterPublicApi` class

```
GroundwaterPublicApi(*, provider_registry=None) -> None
```

Execute groundwater problems through shared Core numerical contracts.

Selected methods.

```
validate(configuration: GroundwaterConfiguration) -> tuple[str, ...]
```

```
run(configuration: GroundwaterConfiguration)
```

`GroundwaterResult` class

```
GroundwaterResult(coordinates: np.ndarray, head: np.ndarray, darcy_flux: np.ndarray, time:
    float, step_count: int, balance_residual: float) -> None
```

Groundwater head, Darcy flux, and storage residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return a serializable groundwater result mapping.

`ImpermeableBoundary` class

```
ImpermeableBoundary(*, boundary: object, physics: object, flux: object = 0.0, jacobian: object
    = None, wave_speed: object = 0.0, factory: object = None) -> None
```

Represent the no-discharge case at the default flux, with ports for a supplied boundary response.

`PoreFluidFlux` class

```
PoreFluidFlux(*, boundary: object, physics: object, flux: object, jacobian: object = None,
    wave_speed: object = 0.0, factory: object = None) -> None
```

Outward Darcy discharge in m/s, with its exact pressure response if dependent.

Selected methods.

```
semantic_conditions(condition_id)
```

```
bind_scalar_balance(*, model, mesh)
```

```
bind_finite_volume(*, model, mesh)
```

`PoromechanicalResult` class

```
PoromechanicalResult(coordinates: np.ndarray, state: PoromechanicalState, strain: np.ndarray,
    darcy_flux: np.ndarray, time: float, step_count: int, elastic_energy: float,
    fluid_storage_energy: float, coupling_residual: float) -> None
```

Final coupled fields and physically relevant Biot diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied arrays and scalar diagnostics for serialization.

`PoromechanicalState` class

```
PoromechanicalState(displacement: np.ndarray, pore_pressure: np.ndarray) -> None
```

Coupled one-dimensional solid displacement and pore-pressure state.

`PoromechanicsConfiguration` class

```
PoromechanicsConfiguration(provider: str = 'geotechnics.poromechanics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'coupled_biot_diffusion',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared mesh, discretization, solver, and time integration contracts.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate the public selection into the shared Core configuration.

`PoromechanicsProvider` class

```
PoromechanicsProvider(*, cell_count: int = 64, elastic_modulus: float = 1.0, biot_coefficient:
    float = 0.8, permeability: float = 1.0, fluid_viscosity: float = 1.0, storage: float =
    1.0, solid_mobility: float = 1.0, body_force: FieldSource = 0.0, fluid_source: FieldSource
    = 0.0, initial_displacement: np.ndarray | None = None, initial_pore_pressure: np.ndarray |
    None = None, final_time: float = 0.0, time_step: float = 0.0001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate the one-dimensional quasi-static Biot system.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.poromechanics') ->
    CallableSpatialOperator
```

Expose the coupled Biot fields through the common Study runner.

```
execute(problem: object | None = None) -> PoromechanicalResult
```

Advance `(u, p)` and return strains, fluxes, energies, and residual.

`PoromechanicsPublicApi` class

```
PoromechanicsPublicApi(*, provider_registry=None) -> None
```

Execute poromechanical problems through the geotechnics registry.

Selected methods.

```
validate(configuration: PoromechanicsConfiguration) -> tuple[str, ...]
```

Return all public-boundary configuration errors.

```
run(configuration: PoromechanicsConfiguration)
```

Validate and execute the configured provider.

`PrescribedPorePressure` class

```
PrescribedPorePressure(*, boundary: object, physics: object, value: object, factory: object =
    None) -> None
```

Pore pressure in Pa; callbacks return one pressure at time/world position.

Selected methods.

```
semantic_conditions(condition_id)
```

```
bind_scalar_balance(*, model, mesh)
```

```
bind_finite_volume(*, model, mesh)
```

`QuadraticCoreyFractionalFlow` class

```
QuadraticCoreyFractionalFlow(*, viscosity_ratio: float = 1.0) -> None
```

Zero residual saturations, krw=S^2, kro=(1-S)^2; ratio=mu_w/mu_o.

Selected methods.

```
fraction(saturation)
```

```
fraction_derivative(saturation)
```

`ReservoirConfiguration` class

```
ReservoirConfiguration(provider: str = 'geotechnics.reservoir', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'two_phase_darcy_fractional_flow',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and reservoir provider parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this API object to the shared Core configuration.

`ReservoirProvider` class

```
ReservoirProvider(*, cell_count: int = 64, permeability: float = 1.0, porosity: float = 0.2,
    total_compressibility: float = 1.0, water_viscosity: float = 1.0, oil_viscosity: float =
    2.0, residual_water_saturation: float = 0.1, residual_oil_saturation: float = 0.1,
    water_corey_exponent: float = 2.0, oil_corey_exponent: float = 2.0, pressure_source:
    FieldSource = 0.0, water_source: FieldSource = 0.0, initial_pressure: np.ndarray | None =
    None, initial_water_saturation: np.ndarray | None = None, final_time: float = 0.0,
    time_step: float = 0.0001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate an immiscible, incompressible two-phase Darcy reservoir model.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.reservoir') ->
    CallableSpatialOperator
```

Expose pressure/saturation transport through the common Study.

```
execute(problem: object | None = None) -> ReservoirResult
```

Advance pressure and saturation and return phase diagnostics.

`ReservoirPublicApi` class

```
ReservoirPublicApi(*, provider_registry=None) -> None
```

Execute reservoir cases through the geotechnics provider registry.

Selected methods.

```
validate(configuration: ReservoirConfiguration) -> tuple[str, ...]
```

Return public-boundary validation errors.

```
run(configuration: ReservoirConfiguration)
```

Validate and execute the selected reservoir provider.

`ReservoirResult` class

```
ReservoirResult(coordinates: np.ndarray, state: ReservoirState, effective_water_saturation:
    np.ndarray, total_flux: np.ndarray, water_flux: np.ndarray, oil_flux: np.ndarray, time:
    float, step_count: int, pressure_balance_residual: float, saturation_balance_residual:
    float) -> None
```

Final reservoir fields, phase fluxes, and conservation diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied arrays and scalar diagnostics for serialization.

`ReservoirState` class

```
ReservoirState(pressure: np.ndarray, water_saturation: np.ndarray) -> None
```

Pressure and wetting-phase saturation on one shared grid.

`RockMechanicsConfiguration` class

```
RockMechanicsConfiguration(provider: str = 'geotechnics.rock_mechanics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'quasi_static_elastoplastic_relaxati
    on', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and rock constitutive parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

RockMechanicsProvider class

```
RockMechanicsProvider(*, cell_count: int = 64, youngs_modulus: float = 10.0, poisson_ratio:
    float = 0.25, cohesion: float = 1.0, friction_angle: float = 0.0, confining_pressure:
    FieldSource = 0.0, hardening_modulus: float = 0.0, shear_yield_stress: float | None =
    None, mobility: float = 1.0, body_force: FieldSource = 0.0, displacement_rate: FieldSource
    = 0.0, initial_displacement: np.ndarray | None = None, initial_equivalent_plastic_strain:
    np.ndarray | None = None, final_time: float = 0.0, time_step: float = 0.0001,
    time_integration: str = 'classical_rk4', **_: object) -> None
```

Advance a small-strain rock bar with Mohr-Coulomb/J2 plasticity.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.rock_mechanics') ->
    CallableSpatialOperator
```

Expose the rock-bar equations through the common Study runner.

```
execute(problem: object | None = None) -> RockMechanicsResult
```

Advance displacement, perform constitutive updates, and report failure.

RockMechanicsPublicApi class

```
RockMechanicsPublicApi(*, provider_registry=None) -> None
```

Execute rock-mechanics cases through the geotechnics registry.

Selected methods.

```
validate(configuration: RockMechanicsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: RockMechanicsConfiguration)
```

Validate and execute the selected provider.

RockMechanicsResult class

```
RockMechanicsResult(coordinates: np.ndarray, state: RockMechanicsState, strain: np.ndarray,
    axial_stress: np.ndarray, yield_function: np.ndarray, confining_pressure: np.ndarray,
    time: float, step_count: int, elastic_energy: float, plastic_dissipation: float,
    equilibrium_residual: float) -> None
```

Final displacement, strain, stress, plastic history, and failure data.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied fields and scalar diagnostics.

RockMechanicsState class

```
RockMechanicsState(displacement: np.ndarray, equivalent_plastic_strain: np.ndarray) -> None
```

Displacement and committed scalar plastic history on one grid.

`SaturationInlet` class

```
SaturationInlet(*, boundary: object, physics: object, value: float) -> None
```

Supply inlet saturation for the model on the selected boundary.

Selected methods.

```
bind_finite_volume(*, model, mesh)
```

`SaturationOutflow` class

```
SaturationOutflow(*, boundary: object, physics: object) -> None
```

Supply the outgoing saturation trace treatment for the selected boundary.

Selected methods.

```
bind_finite_volume(*, model, mesh)
```

`SoilMechanicsConfiguration` class

```
SoilMechanicsConfiguration(provider: str = 'geotechnics.soil_mechanics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'terzaghi_consolidation',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and consolidation parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`SoilMechanicsProvider` class

```
SoilMechanicsProvider(*, cell_count: int = 64, permeability: float = 1.0, fluid_viscosity:
    float = 1.0, storage: float = 1.0, coupling_coefficient: float = 1.0,
    constrained_compressibility: float = 0.01, effective_stress_coefficient: float = 1.0,
    total_stress: FieldSource = 0.0, total_stress_rate: FieldSource = 0.0,
    initial_pore_pressure: np.ndarray | None = None, initial_settlement: np.ndarray | None =
    None, initial_void_ratio: np.ndarray | None = None, final_time: float = 0.0, time_step:
    float = 0.0001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate a saturated 1D Terzaghi-type consolidation model.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.soil_mechanics', coordinates:
    np.ndarray | None = None, gradient_operator: object | None = None, diffusion_operator:
    object | None = None) -> CallableSpatialOperator
```

Expose the consolidation fields through the common Study runner.

```
execute(problem: object | None = None) -> SoilMechanicsResult
```

Advance consolidation and return effective stress and drainage data.

`SoilMechanicsPublicApi` class

```
SoilMechanicsPublicApi(*, provider_registry=None) -> None
```

Execute soil-mechanics cases through the geotechnics registry.

Selected methods.

```
validate(configuration: SoilMechanicsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: SoilMechanicsConfiguration)
```

Validate and execute the selected provider.

`SoilMechanicsResult` class

```
SoilMechanicsResult(coordinates: np.ndarray, state: SoilMechanicsState, total_stress:
    np.ndarray, effective_stress: np.ndarray, darcy_flux: np.ndarray, time: float, step_count:
    int, pressure_balance_residual: float, settlement_balance_residual: float,
    degree_of_consolidation: float) -> None
```

Final consolidation fields and mass/effective-stress diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied fields and scalar diagnostics for serialization.

`SoilMechanicsState` class

```
SoilMechanicsState(pore_pressure: np.ndarray, settlement: np.ndarray, void_ratio: np.ndarray)
    -> None
```

Excess pore pressure, accumulated compression, and void ratio.

`TerzaghiConsolidation` class

```
TerzaghiConsolidation(*, domain: object, model_id: str = <factory>, permeability: float,
    fluid_viscosity: float, constrained_storage: float, constrained_compressibility: float,
    total_stress: float, fluid_supply: object = 0.0) -> None
```

Advance the scalar consolidation balance with constrained storage, compressibility and prescribed total stress.

Declared fields. `pore_pressure` (Pa, differential), `vertical_strain` (1, differential), `void_ratio` (1, differential), `effective_stress` (Pa, derived), `darcy_flux` (m/s, derived), `water_supply` (1/s, input).

A.8 onesim.physics.heat_transfer

`advanced_thermal` namespace

Central product integration for advanced Thermal features.

`configuration` namespace

Public thermal configuration built on the shared numerical contract.

`ConvectiveHeatFlux` class

```
ConvectiveHeatFlux(*, boundary: object, physics: object, heat_transfer_coefficient: object,
    ambient_temperature: object) -> None
```

Use an ambient temperature and transfer coefficient to prescribe convective exchange across a boundary.

`ConvectiveHeatTransfer` class

```
ConvectiveHeatTransfer(*, domain, material, velocity, heating=None, heating_source=None,
    model_id=None)
```

Solve volume heat transport with a prescribed velocity, thermal material and optional volumetric source.

Declared fields. `temperature` (K, differential), `heat_flux` (W/m^2, derived, x/y/z), `heating` (W/m^3, input), `advecting_velocity` (m/s, input, x/y/z).

`errors` namespace

Machine-readable errors at the public Thermal API boundary.

`FixedTemperature` class

```
FixedTemperature(*, boundary: object, physics: object, value: object) -> None
```

Set an absolute temperature trace on a thermal boundary.

`heat_conduction` namespace

Public execution contract for the executable heat-conduction variant.

`heat_transport` namespace

Public execution contract for Thermal heat transport.

`HeatAdvectionDiffusion` class

```
HeatAdvectionDiffusion(*, domain: object, model_id: str = <factory>, velocity: float,
    thermal_diffusivity: float, heating_rate: object = 0.0) -> None
```

Advance the scalar temperature field on a line with prescribed speed, diffusivity and heating rate.

Declared fields. `temperature` (K, differential), `thermal_supply` (K/s, input).

`HeatConduction` class

```
HeatConduction(*, domain, material, heating=None, heating_source=None, model_id=None)
```

Solve the Fourier heat balance with a material, volume domain and optional volumetric source.

Declared fields. `temperature` (K, differential), `heat_flux` (W/m^2, derived, x/y/z), `heating` (W/m^3, input).

`HeatFlux` class

```
HeatFlux(*, boundary: object, physics: object, value: object) -> None
```

Prescribed outward Fourier heat flux, positive for heat leaving the body.

Selected methods.

```
outward_heat_flux(*, points, time)
```

`PhaseChangeHeating` class

```
PhaseChangeHeating(*, domain: object, model_id: str = <factory>, mass: float, heating_power:
    float | None, phase_law: object) -> None
```

Evolve a lumped thermal state with a phase-change enthalpy law.

Declared fields. `specific_enthalpy` (J/kg, differential), `temperature` (K, derived), `liquid_fraction` (1, derived), `heat_source` (W, input).

public_api namespace

API-first thermal facade with explicit shared-numerics planning.

RadiationCooling class

```
RadiationCooling(*, domain: object, model_id: str = <factory>, heat_capacity: float,
    radiating_area: float, radiation: object, heating_power: float | None = 0.0) -> None
```

Evolve a lumped temperature under a radiative power law and optional heating.

Declared fields. `temperature` (K, differential), `heat_source` (W, input), `radiated_power` (W, derived).

RadiativeHeatFlux class

```
RadiativeHeatFlux(*, boundary: object, physics: object, emissivity: object,
    ambient_temperature: object, stefan_boltzmann: float = 5.670374419e-08) -> None
```

Grey diffuse outward radiation to a prescribed thermal enclosure.

Selected methods.

```
outward_flux(temperature, *, points, time)
```

ThermalGreyDiffuseRadiationBoundary class

```
ThermalGreyDiffuseRadiationBoundary(condition_id: str, emissivity: float, ambient_temperature:
    float, domain_ids: tuple[str, ...] = (), stefan_boltzmann: float = 5.670374419e-08,
    ad_capability_id: str = 'capability:automatic_differentiation', newton_capability_id: str
    = 'coupling.newton') -> None
```

Gray, diffuse, opaque surface radiation to an ambient enclosure.

Selected methods.

```
flux(surface_temperature: float) -> float
```

Evaluate the declared physical radiation flux for contract tests.

```
jacobian(surface_temperature: float) -> float
```

Return the analytic d(flux)/d(surface_temperature) contract value.

```
to_boundary_condition() -> ThermalBoundaryCondition
```

```
to_nonlinear_contract() -> "'ThermalRadiationNonlinearContract'"
```

```
to_optional_ir_block() -> ThermalOptionalIRBlock
```

```
to_mapping() -> dict[str, object]
```

ThermalPhaseChangeLawSet class

```
ThermalPhaseChangeLawSet(law_set_id: str, solidus_temperature: float, liquidus_temperature:
    float, latent_heat: float, solid_heat_capacity: float, liquid_heat_capacity: float,
    version: str = '1.0.0') -> None
```

Piecewise-smooth enthalpy formulation with a finite mushy interval.

Selected methods.

```
phase_fraction(temperature: float | np.ndarray) -> float | np.ndarray
```

```
enthalpy(temperature: float | np.ndarray) -> float | np.ndarray
```

```
effective_heat_capacity(temperature: float | np.ndarray) -> float | np.ndarray
```

```
temperature_from_enthalpy(enthalpy: float | np.ndarray) -> float | np.ndarray
```

Invert this monotone piecewise enthalpy law, relative to solidus.

```
temperature_enthalpy_derivative(enthalpy)
```

Branch derivative of the inverse enthalpy law, using its knot convention.

```
phase_fraction_derivative(temperature)
```

Interior mushy slope; saturated branches have zero slope.

```
laws() -> tuple[ThermalPropertyLaw, ...]
```

```
register(registry: ThermalPropertyLawRegistry) -> tuple[str, ...]
```

Register all phase-change laws in the shared material-law registry.

```
evaluate_registered(registry: ThermalPropertyLawRegistry, temperature: float) -> dict[str,
    float]
```

```
to_optional_ir_block() -> ThermalOptionalIRBlock
```

```
to_mapping() -> dict[str, object]
```

`validation_report` namespace

Structured public validation report for one Thermal configuration.

A.9 onesim.physics.hydrodynamics

`CahnHilliard` class

```
CahnHilliard(*, domain: object, model_id: str = <factory>, free_energy: object, mobility:
    object, velocity: object | None = None, source: object | None = None) -> None
```

Evolve composition and chemical potential with a specified free energy and mobility.

Declared fields. `composition` (1, differential), `chemical_potential` (Pa, algebraic), `capillary_force` (N/m^3, derived, x/y/z), `phase_flux` (m/s, derived, x/y/z), `free_energy_density` (J/m^3, derived), `advecting_velocity` (m/s, input, x/y/z), `composition_source` (1/s, input).

`CompositionGradient` class

```
CompositionGradient(*, boundary: object, physics: object, value: object = 0.0) -> None
```

Outward composition derivative grad(c).n, in 1/m.

Selected methods.

```
composition_gradient(*, points, time)
```

`CompressibleEuler` class

```
CompressibleEuler(*, domain: object, model_id: str = <factory>, equation_of_state: object,
    mass_source: object = 0.0, momentum_source: object = 0.0, energy_source: object = 0.0) ->
    None
```

Advance conservative mass, momentum and total energy with an equation of state and optional sources.

Declared fields. `density` (kg/m^3, differential), `momentum` (kg/m^2/s, differential), `total_energy` (J/m^3, differential), `velocity` (m/s, derived), `pressure` (Pa, derived), `internal_energy` (J/kg, derived), `mass_supply` (kg/m^3/s, input), `momentum_supply` (N/m^3, input), `energy_supply` (W/m^3, input).

Selected methods.

```
reflect_boundary_state(interior, *, normal)
```

```
reflect_boundary_state_jacobian(interior, *, normal)
```

`DarcyFluxBoundary` class

```
DarcyFluxBoundary(*, boundary: object, physics: object, flux: object = 0.0, factory: object |
    None = None) -> None
```

Externally prescribed outward Darcy velocity in m/s.

Selected methods.

```
semantic_conditions(condition_id)
```

```
bind_scalar_balance(*, model, mesh)
```

```
bind_finite_volume(*, model, mesh)
```

`DarcyPressureBoundary` class

```
DarcyPressureBoundary(*, boundary: object, physics: object, value: object, factory: object |
    None = None) -> None
```

Physical pressure in Pa, constant or callback(time, coordinate).

Selected methods.

```
semantic_conditions(condition_id)
```

```
bind_scalar_balance(*, model, mesh)
```

```
bind_finite_volume(*, model, mesh)
```

`DarcyPressureFlow` class

```
DarcyPressureFlow(*, domain: object, model_id: str = <factory>, permeability: float,
    viscosity: float, pressure_storage: float) -> None
```

Advance or solve porous pressure with Darcy velocity derived from permeability and viscosity.

Declared fields. `pressure` (Pa, differential), `darcy_velocity` (m/s, derived).

Selected methods.

```
scalar_balance()
```

`DoubleWellFreeEnergy` class

```
DoubleWellFreeEnergy(*, surface_tension: object, interface_width: object) -> None
```

psi(c)=beta*(c^2-1)^2/4 and gradient energy kappa*|grad(c)|^2/2.

Selected methods.

```
density(composition)
```

```
derivative(composition)
```

```
second_derivative(composition)
```

`EulerInvariantDomain` class

```
EulerInvariantDomain(density_floor: float = 1e-12, pressure_floor: float = 1e-12) -> None
```

Provide density and pressure floors for a compatible conservative invariant-domain limiter.

Selected methods.

```
bind(*, model, law)
```

`FixedVelocity` class

```
FixedVelocity(*, boundary: object, physics: object, value: object) -> None
```

Physical Cartesian velocity trace, including time-dependent motion.

`FlowTraction` class

```
FlowTraction(*, boundary: object, physics: object, value: object) -> None
```

Prescribed outward Cauchy traction sigma.n, including viscous stress.

Selected methods.

```
traction(*, points, normal, time)
```

`IdealGasEquationOfState` class

```
IdealGasEquationOfState(gamma: float = 1.4) -> None
```

Close the Euler pressure and wave-speed relations for an ideal gas.

Selected methods.

```
to_conserved(primitive: Euler1DPrimitiveState) -> Euler1DConservedState
```

```
to_primitive(state: Euler1DConservedState) -> Euler1DPrimitiveState
```

```
flux(state: Euler1DConservedState) -> Euler1DConservedState
```

```
flux_jacobian(state: Euler1DConservedState)
```

Complete conservative-coordinate derivative of the physical flux.

```
maximum_wave_speed(state: Euler1DConservedState) -> float
```

```
maximum_wave_speed_gradient(state: Euler1DConservedState)
```

Conservative-coordinate gradient, selecting zero for abs(u) at rest.

`IncompressibleNavierStokes` class

```
IncompressibleNavierStokes(*, domain: object, model_id: str = <factory>, material: object,
    body_force: object | None = None) -> None
```

Solve velocity and pressure from momentum and incompressibility with a fluid material and body force.

Declared fields. `velocity` (m/s, differential, x/y/z), `pressure` (Pa, algebraic), `body_force` (N/m^3, input, x/y/z).

`NoSlipWall` class

```
NoSlipWall(*, boundary: object, physics: object) -> None
```

Set stationary zero wall velocity for the owned flow model.

`OldroydBShear` class

```
OldroydBShear(*, domain: object, model_id: str = <factory>, shear_rate: float | None,
    stress_law: object) -> None
```

Advance homogeneous polymer stress in a prescribed shear using the selected stress law.

Declared fields. `stress_xx` (Pa, differential), `stress_xy` (Pa, differential), `stress_yy` (Pa, differential), `imposed_rate` (1/s, input).

`PhaseAdvection` class

```
PhaseAdvection(*, domain: object, model_id: str = <factory>, velocity: float, first_density:
    float, second_density: float, phase_source: object = 0.0) -> None
```

Advance a phase fraction under prescribed transport and derive mixture density.

Declared fields. `volume_fraction` (1, differential), `mixture_density` (kg/m^3, derived), `volume_supply` (1/s, input).

`PhaseFlux` class

```
PhaseFlux(*, boundary: object, physics: object, value: object = 0.0) -> None
```

Outward total phase flux (c*u-M*grad(mu)).n, in m/s.

Selected methods.

```
phase_flux(*, points, time)
```

`PhaseFractionInvariantDomain` class

```
PhaseFractionInvariantDomain() -> None
```

One convex scaling preserves the cell average and enforces both bounds.

Selected methods.

```
coordinate_bounds(*, point_count)
```

```
bind(*, model, law)
```

```
admissible(values)
```

```
source_step(values, source)
```

```
maximum_scaling(mean, samples)
```

`PressureTraction` class

```
PressureTraction(*, boundary: object, physics: object, value: object) -> None
```

Outward traction -p_ext*n, with zero imposed tangential traction.

Selected methods.

```
traction(*, points, normal, time)
```

`ReactiveMixtureTransport` class

```
ReactiveMixtureTransport(*, domain: object, model_id: str = <factory>, network:
    ReactionNetworkIR, volumetric_heat_capacity: float, kinetics: object = None,
    thermal_source: object = 0.0, species_sources: Mapping = <factory>, velocity: float) ->
    None
```

Couple species kinetics with thermal and advective transport using an explicit reaction network.

`RotatingWall` class

```
RotatingWall(*, boundary: object, physics: object, angular_velocity: tuple, rotation_origin:
    tuple = (0.0, 0.0, 0.0)) -> None
```

Prescribe rigid-wall rotation around an origin through the angular-velocity vector.

Selected methods.

```
velocity_at(points)
```

`ShallowWater` class

```
ShallowWater(*, domain: object, model_id: str = <factory>, gravity: float = 9.81,
    water_source: object = 0.0, discharge_source: object = 0.0) -> None
```

Advance water depth and discharge with gravity and optional sources.

Declared fields. `depth` (m, differential), `discharge` (m^2/s, differential), `velocity` (m/s, derived), `water_supply` (m/s, input), `momentum_supply` (m^2/s^2, input).

Selected methods.

```
reflect_boundary_state(interior, *, normal)
```

```
reflect_boundary_state_jacobian(interior, *, normal)
```

`ShallowWaterInvariantDomain` class

```
ShallowWaterInvariantDomain(depth_floor: float = 1e-12) -> None
```

Supply the positive-depth floor for compatible shallow-water limiting.

Selected methods.

```
bind(*, model, law)
```

```
admissible(values)
```

```
maximum_scaling(mean, samples)
```

```
source_step(values, source)
```

Exact forward-update limit for the positive-depth half-space.

`TopographicShallowWater` class

```
TopographicShallowWater(*, domain: object, model_id: str = <factory>, gravity: float = 9.81,
    water_source: object = 0.0, discharge_source: object = 0.0, bed_elevation: object = 0.0)
    -> None
```

Stationary bed elevation in metres; depth may vanish with zero discharge.

Declared fields. `depth` (m, differential), `discharge` (m^2/s, differential), `velocity` (m/s, derived), `water_supply` (m/s, input), `momentum_supply` (m^2/s^2, input), `bed` (m, derived), `free_surface` (m, derived).

Selected methods.

```
bed_values(coordinates)
```

```
hydrostatic_balance()
```

`TransmissiveBoundary` class

```
TransmissiveBoundary(*, boundary: object, physics: object) -> None
```

Zero-gradient exterior states for the model's differential fields.

Selected methods.

```
bind_finite_volume(*, model, mesh)
```

```
semantic_conditions(condition_id)
```

`ViscoelasticStressLaw` class

```
ViscoelasticStressLaw(relaxation_time: float, polymer_viscosity: float) -> None
```

Evaluate Oldroyd-B upper-convected stress rate and relaxation.

Selected methods.

```
evaluate(stress: tuple[tuple[float, ...], ...], velocity_gradient: tuple[tuple[float, ...],
    ...]) -> tuple[tuple[float, ...], ...]
```

```
jacobian_action(stress, velocity_gradient, *, stress_direction, gradient_direction)
```

Exact joint derivative of relaxation, source and convected deformation.

`WetDryShallowWaterDomain` class

```
WetDryShallowWaterDomain() -> None
```

Provide the shallow-water invariant-domain policy that permits a dry state.

Selected methods.

```
bind(*, model, law)
```

```
admissible(values)
```

```
source_step(values, source)
```

A.10 onesim.physics.magnetohydrodynamics

`adapt_ambipolar_diffusion_result` function

```
adapt_ambipolar_diffusion_result(result: AmbipolarDiffusionResult, *, result_id: str =
    'magnetohydrodynamics.ambipolar_diffusion.result', plan_reference: ArtifactReference |
    None = None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish ambipolar magnetic diffusion and its residual field.

`adapt_built_in_mhd_result` function

```
adapt_built_in_mhd_result(result: object, *, provider_id: str | None = None, result_id: str =
    'magnetohydrodynamics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Adapt any typed built-in MHD result through one dispatch entry point.

`adapt_cgl_mhd_result` function

```
adapt_cgl_mhd_result(result: CGLMHDResult, *, result_id: str =
    'magnetohydrodynamics.cgl_mhd.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish conservative and gyrotropic-pressure CGL fields.

`adapt_drift_diffusion_result` function

```
adapt_drift_diffusion_result(result: DriftDiffusionResult, *, result_id: str =
    'magnetohydrodynamics.drift_diffusion.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish charged-particle density, electric field, and particle flux.

`adapt_electromagnetic_plasma_result` function

```
adapt_electromagnetic_plasma_result(result: ElectromagneticPlasmaResult, *, result_id: str =
    'magnetohydrodynamics.electromagnetic_plasma.result', plan_reference: ArtifactReference |
    None = None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish transverse EM plasma fields and charge-continuity residual.

adapt_electrostatic_plasma_result function

```
adapt_electrostatic_plasma_result(result: ElectrostaticPlasmaResult, *, result_id: str =
    'magnetohydrodynamics.electrostatic_plasma.result', plan_reference: ArtifactReference |
    None = None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish electrostatic density, potential, field, and Poisson residual.

adapt_glm_mhd_result function

```
adapt_glm_mhd_result(result: GLMMHDResult, *, provider_id: str = 'glm_mhd_2d', result_id: str
    = 'magnetohydrodynamics.glm_mhd.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish GLM divergence-cleaning fields and optional conservative state.

adapt_hall_mhd_result function

```
adapt_hall_mhd_result(result: HallMHDResult, *, provider_id: str = 'hall_mhd_2d', result_id:
    str = 'magnetohydrodynamics.hall_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish Hall-MHD magnetic induction and divergence fields.

adapt_mhd_2d_result function

```
adapt_mhd_2d_result(result: MHD2DResult, *, provider_id: str = 'resistive_mhd_2d', result_id:
    str = 'magnetohydrodynamics.mhd_2d.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish the conservative 2-D MHD state and pressure.

adapt_mhd_3d_result function

```
adapt_mhd_3d_result(result: MHD3DResult, *, provider_id: str = 'resistive_mhd_3d', result_id:
    str = 'magnetohydrodynamics.mhd_3d.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish the conservative 3-D MHD state and optional diagnostics.

adapt_mhd_result function

```
adapt_mhd_result(result: object, *, provider_id: str | None = None, result_id: str =
    'magnetohydrodynamics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Public alias for :func:adapt_built_in_mhd_result.

adapt_multi_fluid_mhd_result function

```
adapt_multi_fluid_mhd_result(result: MultiFluidMHDResult, *, result_id: str =
    'magnetohydrodynamics.multi_fluid_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish arbitrary species fields from a multi-fluid result.

adapt_multiphase_mhd_result function

```
adapt_multiphase_mhd_result(result: MultiphaseMHDResult, *, result_id: str =
    'magnetohydrodynamics.multiphase_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish multiphase MHD conserved and phase-fraction fields.

adapt_porous_mhd_result function

```
adapt_porous_mhd_result(result: PorousMHDResult, *, result_id: str =
    'magnetohydrodynamics.porous_mhd.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish porous-MHD fields and Darcy–Forchheimer drag power.

adapt_quasistatic_mhd_result function

```
adapt_quasistatic_mhd_result(result: QuasistaticMHDResult, *, result_id: str =
    'magnetohydrodynamics.quasistatic_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish quasistatic electric fields and the nonlinear force diagnostics.

adapt_radiation_exchange_result function

```
adapt_radiation_exchange_result(result: RadiationExchangeResult, *, result_id: str =
    'magnetohydrodynamics.radiation_exchange.result', plan_reference: ArtifactReference | None
    = None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish matter/radiation energies and conservative exchange residual.

adapt_radiation_mhd_result function

```
adapt_radiation_mhd_result(result: RadiationMHDResult, *, result_id: str =
    'magnetohydrodynamics.radiation_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish radiation-MHD conserved, radiation, and exchange fields.

adapt_reactive_mhd_result function

```
adapt_reactive_mhd_result(result: ReactiveMHDResult, *, result_id: str =
    'magnetohydrodynamics.reactive_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish reactive MHD conserved, composition, and heat-release fields.

adapt_reactive_source_result function

```
adapt_reactive_source_result(result: ReactiveSourceResult, *, result_id: str =
    'magnetohydrodynamics.reactive_source.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish a 1-D reactive source state.

adapt_relativistic_mhd_result function

```
adapt_relativistic_mhd_result(result: RelativisticMHDResult, *, result_id: str =
    'magnetohydrodynamics.relativistic_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish special-relativistic conservative and primitive fields.

`adapt_two_fluid_mhd_result` function

```
adapt_two_fluid_mhd_result(result: TwoFluidMHDResult, *, result_id: str =
    'magnetohydrodynamics.two_fluid_mhd.result', plan_reference: ArtifactReference | None =
    None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish ion/electron states and electrostatic consistency.

`InductionlessMomentum` class

```
InductionlessMomentum(*, domain: object, model_id: str = <factory>, density: float,
    kinematic_viscosity: float, conductivity: float, magnetic_field: float, pressure_gradient:
    float, electric_field: float = 0.0) -> None
```

Solve the momentum balance in a prescribed magnetic field with induction neglected by the physical model.

Declared fields. `velocity` (m/s, differential), `current_density` (A/m^2, derived), `lorentz_force` (N/m^3, derived).

`InductionlessNoSlipWall` class

```
InductionlessNoSlipWall(*, boundary: object, physics: object, velocity: object = 0.0, factory:
    object | None = None) -> None
```

Velocity of the no-slip wall in m/s, constant or callback(time, coordinate).

Selected methods.

```
semantic_conditions(condition_id)
```

```
bind_scalar_balance(*, model, mesh)
```

```
bind_finite_volume(*, model, mesh)
```

`MagnetohydrodynamicPublicApi` class

```
MagnetohydrodynamicPublicApi(provider_factory: Callable[[str], object] | None = None,
    validator: object | None = None, *, provider_registry: object | None = None) -> None
```

Provider-neutral facade for standalone and coupled MHD execution.

Selected methods.

```
create_provider(provider_id: str, **parameters: object) -> object
```

Create an MHD provider through the public delegation boundary.

```
provider_ids() -> tuple[str, ...]
```

Return the public, stable IDs available for delegation.

```
create_common_study_operator(kind: str, *, provider_id: str, method_id: str, initial_state:
    object | None = None, electric_field: object | None = None, **options: object) -> object
```

Create a common-study operator for a public MHD delegation.

```
bind_common_study_system(system: object, model: object, *, context: Mapping[str, object] |
    None = None) -> object | None
```

Bind one system through MHD's public common-study boundary.

```
result_diagnostics() -> object
```

Create the public MHD result-diagnostics service for adapters.

```
run(problem: InitialValueProblem, configuration: NumericalConfiguration) -> object
```

Run a problem with a numerical MHD configuration.

```
run_mapping(problem: object, configuration: Mapping[str, object]) -> object
```

Run an MHD provider from a JSON-compatible configuration mapping.

`MAGNETOHYDRODYNAMICS` value

Public identity for the magnetohydrodynamic methodology.

`RadiationMatterExchange` class

```
RadiationMatterExchange(*, domain: object, model_id: str = <factory>, heat_capacity: float,
    absorption: float, light_speed: float, radiation_constant: float, matter_power: float |
    None = 0.0, radiation_power: float | None = 0.0) -> None
```

Integrate the homogeneous matter and radiation energy balances and their exchange.

Declared fields. `matter_energy` (J/m^3, differential), `radiation_energy` (J/m^3, differential), `temperature` (K, derived), `matter_source` (W/m^3, input), `radiation_source` (W/m^3, input).

`TransverseMHDWaves` class

```
TransverseMHDWaves(*, domain: object, model_id: str = <factory>, density: float,
    longitudinal_field: float, permeability: float, magnetic_diffusivity: float = 0.0,
    kinematic_viscosity: float = 0.0, hall_diffusivity: float = 0.0, body_force_y: object =
    0.0, body_force_z: object = 0.0, magnetic_source_y: object = 0.0, magnetic_source_z:
    object = 0.0) -> None
```

Evolve two transverse velocity and magnetic perturbations with optional dissipative and Hall terms.

Declared fields. `velocity_y` (m/s, differential), `velocity_z` (m/s, differential), `magnetic_y` (T, differential), `magnetic_z` (T, differential), `force_y` (N/m^3, input), `force_z` (N/m^3, input), `induction_rate_y` (T/s, input), `induction_rate_z` (T/s, input).

`TwoFluidExchange` class

```
TwoFluidExchange(*, domain: object, model_id: str = <factory>, density_one: float,
    density_two: float, drag_coefficient: float, force_one: float | None = 0.0, force_two:
    float | None = 0.0, heating_one: float | None = 0.0, heating_two: float | None = 0.0) ->
    None
```

Integrate homogeneous momentum and energy exchange of two fluids.

Declared fields. `velocity_one` (m/s, differential), `velocity_two` (m/s, differential), `energy_one` (J/m^3, differential), `energy_two` (J/m^3, differential), `force_density_one` (N/m^3, input), `force_density_two` (N/m^3, input), `heat_source_one` (W/m^3, input), `heat_source_two` (W/m^3, input).

Selected methods.

```
force_input(fluid)
```

```
heat_input(fluid)
```

A.11 onesim.physics.magnetostatics

`CoulombGauge` class

```
CoulombGauge(multiplier_reference_dof: int = 0) -> None
```

The div(A)=0 constraint; no implicit penalty parameter or mass shift.

Selected methods.

```
bind(*, space, mesh, boundary_facets)
```

```
semantic_constraint(identifier, field)
```

`MagneticFieldBoundary` class

```
MagneticFieldBoundary(*, boundary: object, physics: object, value: object) -> None
```

Prescribed tangential H. the Ampere weak boundary source is H cross n.

Selected methods.

```
bind_trace(**context)
```

`MagneticInsulation` class

```
MagneticInsulation(*, boundary: object, physics: object) -> None
```

Tangential A=0 on the outer boundary of the vector-potential problem.

Selected methods.

```
bind_potential(**context)
```

`MagneticPotentialBoundary` class

```
MagneticPotentialBoundary(*, boundary: object, physics: object, value: object) -> None
```

Prescribed tangential vector potential in Wb/m, evaluated in world coordinates.

Selected methods.

```
bind_potential(**context)
```

`MagneticReluctanceBoundary` class

```
MagneticReluctanceBoundary(*, boundary: object, physics: object, reluctance: object,
    reference_potential: object = (0.0, 0.0, 0.0), reference_field: object = (0.0, 0.0, 0.0))
    -> None
```

H cross n + beta A_t = H_ref cross n + beta A_ref,t, beta in 1/H.

Selected methods.

```
bind_trace(**context)
```

`MagneticVectorPotential` class

```
MagneticVectorPotential(*, gauge: object) -> None
```

Select the vector-potential formulation and provide its gauge declaration.

`Magnetostatics` class

```
Magnetostatics(*, domain: object, model_id: str = <factory>, permeability: object,
    formulation: object, current_sources: tuple = ()) -> None
```

Solve the static magnetic field through the declared formulation, permeability and current sources.

Declared fields. `vector_potential` (Wb/m, algebraic, x/y/z), `magnetic_flux_density` (T, derived, x/y/z), `magnetic_field_strength` (A/m, derived, x/y/z).

Selected methods.

```
intrinsic_constraints(problem_id)
```

`SolenoidCurrent` class

```
SolenoidCurrent(*, inner_radius: float, outer_radius: float, length: float, turns: int,
    current: float, center: tuple = (0.0, 0.0, 0.0), axis: tuple = (0.0, 0.0, 1.0), source_id:
    str = <factory>) -> None
```

Uniform azimuthal current density over a resolved winding cross section.

Selected methods.

```
current_density(points)
```

A.12 onesim.physics.multibody

`adapt_drivetrain_result` function

```
adapt_drivetrain_result(result: DrivetrainResult, *, result_id: str =
    'multibody.drivetrains.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish shaft states, pair coupling quantities, and power diagnostics.

`adapt_flexible_body_result` function

```
adapt_flexible_body_result(result: FlexibleBodyResult, *, result_id: str =
    'multibody.flexible_body.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish reduced modal coordinates and one field per modal shape.

`adapt_joint_system_result` function

```
adapt_joint_system_result(result: JointSystemResult, *, result_id: str =
    'multibody.joint_systems.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish generalized coordinates, joint constraints, and reactions.

`adapt_kinematics_result` function

```
adapt_kinematics_result(result: KinematicsResult, *, result_id: str =
    'multibody.kinematics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish serial-chain joint fields, geometry, Jacobian, and diagnostics.

adapt_mechanism_result function

```
adapt_mechanism_result(result: MechanismResult, *, result_id: str =
    'multibody.mechanisms.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish four-bar joint angles, point geometry, and closure quality.

adapt_multibody_result function

```
adapt_multibody_result(result: object, *, provider_id: str | None = None, result_id: str =
    'multibody.result', plan_reference: ArtifactReference | None = None, diagnostics:
    Sequence[str] = ()) -> SimulationResultSet
```

Adapt any built-in multibody result through one explicit entry point.

adapt_pendulum_result function

```
adapt_pendulum_result(result: PendulumResult, *, result_id: str = 'multibody.pendulum.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish the simple-pendulum state and energy conservation diagnostic.

adapt_rigid_body_dynamics_result function

```
adapt_rigid_body_dynamics_result(result: RigidBodyDynamicsResult, *, result_id: str =
    'multibody.rigid_body_dynamics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish six-degree-of-freedom rigid-body state and balances.

DEFAULT_MULTIBODY_PROVIDER_REGISTRY value

Default registry for the domain provider route. Inspect the registered IDs when choosing a provider.

DrivenFourBar class

```
DrivenFourBar(*, domain: object, model_id: str = <factory>, ground_length: float,
    crank_length: float, coupler_length: float, rocker_length: float, drive_speed: float |
    None, assembly_branch: str = 'open') -> None
```

Specify linkage lengths, drive speed and the intended assembly branch.

Declared fields. `input_angle` (rad, differential), `coupler_angle` (rad, derived), `rocker_angle` (rad, derived), `crank_x` (m, derived), `crank_y` (m, derived), `rocker_x` (m, derived), `rocker_y` (m, derived), `drive` (rad/s, input).

DrivetrainConfiguration class

```
DrivetrainConfiguration(provider: str = 'multibody.drivetrains', mesh: str = 'core.dof.0d',
    discretization: str = 'lumped_dof', time_integration: str = 'classical_rk4', solver: str =
    'torsional_network', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and torsional-network parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`DrivetrainProvider` class

```
DrivetrainProvider(*, inertias: Sequence[float] = (1.0, 1.0), gear_ratios: Sequence[float] |
    None = None, shaft_stiffness: Sequence[float] | None = None, shaft_damping:
    Sequence[float] | None = None, external_torque: TorqueSource = 0.0, initial_angles:
    Sequence[float] | None = None, initial_angular_velocities: Sequence[float] | None = None,
    final_time: float = 0.0, time_step: float = 0.001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate a flexible torsional gear-and-shaft network.

Selected methods.

```
execute(problem: object | None = None) -> DrivetrainResult
```

Advance the coupled shaft state and return energy/power diagnostics.

`DrivetrainPublicApi` class

```
DrivetrainPublicApi(*, provider_registry=None) -> None
```

Execute drivetrain cases through the multibody provider registry.

Selected methods.

```
validate(configuration: DrivetrainConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: DrivetrainConfiguration)
```

Validate and execute the selected provider.

`DrivetrainResult` class

```
DrivetrainResult(state: DrivetrainState, shaft_torques: np.ndarray, relative_twist:
    np.ndarray, relative_speed: np.ndarray, time: float, step_count: int, kinetic_energy:
    float, shaft_energy: float, damping_dissipation: float, power_balance_residual: float) ->
    None
```

Final shaft state, coupling torques, energies, and power residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied arrays and scalar drivetrain diagnostics.

`DrivetrainState` class

```
DrivetrainState(angles: np.ndarray, angular_velocities: np.ndarray) -> None
```

Generalized shaft angles and angular velocities.

`EulerTop` class

```
EulerTop(*, domain: object, model_id: str = <factory>, inertia: tuple[float, float, float],
    torque: tuple[float | None, float | None, float | None] = (0.0, 0.0, 0.0)) -> None
```

Integrate body-axis angular velocity with principal inertias and applied torques.

Declared fields. `omega_x` (rad/s, differential), `omega_y` (rad/s, differential), `omega_z` (rad/s, differential), `energy` (J, derived), `momentum_norm` (kg*m^2/s, derived), `torque_x` (N*m, input), `torque_y` (N*m, input),

`torque_z` (N*m, input).

Selected methods.

```
angular_velocity(axis)
```

```
applied_torque(axis)
```

`FlexibleBodyProvider` class

```
FlexibleBodyProvider(*, mass_matrix: Sequence[Sequence[float]], stiffness_matrix:
    Sequence[Sequence[float]], mode_count: int | None = None, modal_forces: Sequence[float] |
    None = None, initial_modal_displacements: Sequence[float] | None = None,
    initial_modal_velocities: Sequence[float] | None = None, damping_ratios: Sequence[float] |
    None = None, final_time: float = 0.0, time_step: float = 0.001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate a flexible body in the registered structural modal basis.

Selected methods.

```
execute(problem: object | None = None) -> FlexibleBodyResult
```

Advance independent mass-normalized modal coordinates.

`FlexibleBodyResult` class

```
FlexibleBodyResult(frequencies: tuple[float, ...], modes: tuple[tuple[float, ...], ...],
    modal_displacements: tuple[float, ...], modal_velocities: tuple[float, ...],
    strain_energy: float, kinetic_energy: float, time: float, step_count: int) -> None
```

Modal basis, reduced state, energy, and integration diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

`JointSystemProvider` class

```
JointSystemProvider(*, masses: Sequence[float] = (1.0,), constraint_matrix:
    Sequence[Sequence[float]] = (), constraint_targets: Sequence[float] = (), external_force:
    ForceSource = 0.0, damping: Sequence[float] | None = None, stiffness: Sequence[float] |
    None = None, penalty_stiffness: float = 10000.0, constraint_damping: float = 100.0,
    enforcement: str = 'projection', initial_positions: Sequence[float] | None = None,
    initial_velocities: Sequence[float] | None = None, final_time: float = 0.0, time_step:
    float = 0.001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate constrained generalized coordinates with joint reactions.

Selected methods.

```
execute(problem: object | None = None) -> JointSystemResult
```

Advance generalized coordinates and enforce all declared joints.

`JointSystemResult` class

```
JointSystemResult(state: JointSystemState, constraint_residual: np.ndarray,
    velocity_constraint_residual: np.ndarray, reaction_forces: np.ndarray, time: float,
    step_count: int, kinetic_energy: float, potential_energy: float, constraint_work: float)
    -> None
```

Final state, constraint residuals, reaction forces, and energies.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied arrays and scalar diagnostics.

`JointSystemsConfiguration` class

```
JointSystemsConfiguration(provider: str = 'multibody.joint_systems', mesh: str =
    'core.dof.0d', discretization: str = 'lumped_dof', time_integration: str =
    'classical_rk4', solver: str = 'constraint_projected_dynamics', parameters: Mapping[str,
    object] = <factory>) -> None
```

Select shared numerical contracts and algebraic joint parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`JointSystemsPublicApi` class

```
JointSystemsPublicApi(*, provider_registry=None) -> None
```

Execute joint-system cases through the multibody registry.

Selected methods.

```
validate(configuration: JointSystemsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: JointSystemsConfiguration)
```

Validate and execute the selected provider.

`JointSystemState` class

```
JointSystemState(positions: np.ndarray, velocities: np.ndarray) -> None
```

Generalized positions and velocities for a constrained multibody system.

`KinematicsConfiguration` class

```
KinematicsConfiguration(provider: str = 'multibody.kinematics', mesh: str = 'core.dof.0d',
    discretization: str = 'lumped_dof', time_integration: str = 'classical_rk4', solver: str =
    'serial_chain_forward_inverse', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and serial-chain parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`KinematicsProvider` class

```
KinematicsProvider(*, link_lengths: Sequence[float] = (1.0, 1.0), initial_joint_angles:
    Sequence[float] | None = None, joint_rates: Sequence[float] | None = None,
    target_position: Sequence[float] | None = None, inverse_damping: float = 1e-06,
    inverse_tolerance: float = 1e-10, inverse_max_iterations: int = 100, final_time: float =
    0.0, time_step: float = 0.001, time_integration: str = 'classical_rk4', **_: object) ->
    None
```

Evaluate a planar serial chain and solve its inverse kinematics.

Selected methods.

```
execute(problem: object | None = None) -> KinematicsResult
```

Advance joint angles, optionally solve a target, and return geometry.

`KinematicsPublicApi` class

```
KinematicsPublicApi(*, provider_registry=None) -> None
```

Execute kinematics cases through the multibody registry.

Selected methods.

```
validate(configuration: KinematicsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: KinematicsConfiguration)
```

Validate and execute the selected provider.

`KinematicsResult` class

```
KinematicsResult(state: KinematicsState, joint_positions: np.ndarray, end_effector:
    np.ndarray, jacobian: np.ndarray, end_effector_velocity: np.ndarray, time: float,
    step_count: int, inverse_residual: float, inverse_iterations: int) -> None
```

Chain geometry, end-effector velocity, Jacobian, and inverse residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied arrays and scalar diagnostics.

`KinematicsState` class

```
KinematicsState(joint_angles: np.ndarray, joint_rates: np.ndarray) -> None
```

Joint angles and angular rates of a serial planar mechanism.

`LinearJointSystem` class

```
LinearJointSystem(*, domain: object, model_id: str = <factory>, masses: tuple[float, ...],
    constraint_matrix: tuple[tuple[float, ...], ...], constraint_targets: tuple[float, ...],
    forces: tuple[float | None, ...]) -> None
```

Integrate masses constrained by a linear joint matrix and target values.

Selected methods.

```
position(i)
```

```
velocity(i)
```

```
reaction(i)
```

```
applied_force(i)
```

`MechanismProvider` class

```
MechanismProvider(*, ground_length: float = 2.0, crank_length: float = 1.0, coupler_length:
    float = 2.0, rocker_length: float = 1.5, initial_input_angle: float = 0.5, input_rate:
    float = 0.0, assembly_branch: str = 'open', final_time: float = 0.0, time_step: float =
    0.001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Solve a closed planar four-bar mechanism.

Selected methods.

```
execute(problem: object | None = None) -> MechanismResult
```

Advance the input crank and solve position and velocity closure.

`MechanismResult` class

```
MechanismResult(state: MechanismState, ground_pivot: np.ndarray, crank_pin: np.ndarray,
    rocker_pin: np.ndarray, closure_residual: float, transmission_angle: float, time: float,
    step_count: int) -> None
```

Closed four-bar geometry, velocities, transmission angle, and residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return mechanism geometry and scalar diagnostics.

`MechanismsConfiguration` class

```
MechanismsConfiguration(provider: str = 'multibody.mechanisms', mesh: str = 'core.dof.0d',
    discretization: str = 'closed_chain_kinematics', time_integration: str = 'classical_rk4',
    solver: str = 'four_bar_closure', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and four-bar parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`MechanismsPublicApi` class

```
MechanismsPublicApi(*, provider_registry=None) -> None
```

Execute mechanism cases through the multibody registry.

Selected methods.

```
validate(configuration: MechanismsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: MechanismsConfiguration)
```

Validate and execute the selected provider.

`MechanismState` class

```
MechanismState(input_angle: float, coupler_angle: float, rocker_angle: float, input_rate:
    float, coupler_rate: float, rocker_rate: float) -> None
```

Input, coupler, and rocker angles with their angular velocities.

`MultibodyProviderRegistry` class

```
MultibodyProviderRegistry() -> None
```

Register multibody providers by stable identifier.

Selected methods.

```
register(provider_id: str, factory: Callable[..., object], *, catalog_role: str =
    'unclassified') -> None
```

```
catalog_role(provider_id: str) -> str
```

```
create(provider_id: str, **parameters: object) -> object
```

```
ids() -> tuple[str, ...]
```

`PendulumProvider` class

```
PendulumProvider(*, length: float = 1.0, mass: float = 1.0, gravity: float = 9.81,
    initial_angle: float = 0.1, initial_angular_velocity: float = 0.0, final_time: float =
    0.0, time_step: float = 0.001, time_integration: str = 'classical_rk4', **_: object) ->
    None
```

Integrate `theta_ddot + g/L sin(theta) = 0` through Core.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:ode.pendulum') ->
    CallableSpatialOperator
```

Expose pendulum dynamics through the common transient Study.

```
execute(problem: object | None = None) -> PendulumResult
```

Advance angle and angular velocity and report energy conservation.

`PendulumResult` class

```
PendulumResult(angle: float, angular_velocity: float, time: float, step_count: int, energy:
    float, energy_residual: float) -> None
```

Final pendulum state and conservative-energy residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

`RigidBodyDynamicsConfiguration` class

```
RigidBodyDynamicsConfiguration(provider: str = 'multibody.rigid_body_dynamics', mesh: str =
    'core.dof.0d', discretization: str = 'rigid_body_state', time_integration: str =
    'classical_rk4', solver: str = 'newton_euler_6dof', parameters: Mapping[str, object] =
    <factory>) -> None
```

Select shared time integration and rigid-body source parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`RigidBodyDynamicsProvider` class

```
RigidBodyDynamicsProvider(*, mass: float = 1.0, inertia: Sequence[float] = (1.0, 1.0, 1.0),
    external_force: VectorSource = (0.0, 0.0, 0.0), external_torque: VectorSource = (0.0, 0.0,
    0.0), initial_position: Sequence[float] = (0.0, 0.0, 0.0), initial_velocity:
    Sequence[float] = (0.0, 0.0, 0.0), initial_quaternion: Sequence[float] = (0.0, 0.0, 0.0,
    1.0), initial_angular_velocity: Sequence[float] = (0.0, 0.0, 0.0), final_time: float =
    0.0, time_step: float = 0.001, time_integration: str = 'classical_rk4', **_: object) ->
    None
```

Integrate translational and rotational rigid-body motion.

Selected methods.

```
execute(problem: object | None = None) -> RigidBodyDynamicsResult
```

Advance the six-degree-of-freedom state and report balances.

`RigidBodyDynamicsPublicApi` class

```
RigidBodyDynamicsPublicApi(*, provider_registry=None) -> None
```

Execute rigid-body dynamics through the multibody registry.

Selected methods.

```
validate(configuration: RigidBodyDynamicsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: RigidBodyDynamicsConfiguration)
```

Validate and execute the selected provider.

`RigidBodyDynamicsResult` class

```
RigidBodyDynamicsResult(state: RigidBodyState, transform: AssemblyTransform, linear_momentum:
    np.ndarray, angular_momentum: np.ndarray, translational_energy: float, rotational_energy:
    float, time: float, step_count: int, force_balance_residual: float,
    torque_balance_residual: float) -> None
```

Final rigid-body state, transform, conservation data, and residuals.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied state arrays and transform-compatible diagnostics.

`RigidBodyState` class

```
RigidBodyState(position: np.ndarray, velocity: np.ndarray, quaternion: np.ndarray,
    angular_velocity: np.ndarray) -> None
```

Center-of-mass and body-frame rotational rigid-body state.

`SimplePendulum` class

```
SimplePendulum(*, domain: object, model_id: str = <factory>, length: float, mass: float = 1.0,
    gravity: float = 9.81, torque: float | None = 0.0) -> None
```

Integrate pendulum angle and angular velocity under gravity and applied torque.

Declared fields. `angle` (rad, differential), `angular_velocity` (rad/s, differential), `energy` (J, derived), `applied_torque` (N*m, input).

`TorsionalDrivetrain` class

```
TorsionalDrivetrain(*, domain: object, model_id: str = <factory>, inertias: tuple[float, ...],
    shaft_stiffness: tuple[float, ...], shaft_damping: tuple[float, ...] = (), gear_ratios:
    tuple[float, ...] = (), external_torques: tuple[float | None, ...] = ()) -> None
```

Integrate shaft and inertia motion with stiffness, damping, torque and gear data.

Selected methods.

```
angle(shaft)
```

```
angular_velocity(shaft)
```

```
applied_torque(shaft)
```

A.13 onesim.physics.multiphysics

`adapt_electro_thermal_result` function

```
adapt_electro_thermal_result(result: object, *, result_id: str =
    'multiphysics.electro_thermal.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish electro-thermal fields and coupling diagnostics.

`adapt_fluid_structure_result` function

```
adapt_fluid_structure_result(result: object, *, result_id: str =
    'multiphysics.fluid_structure.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish fluid/structure fields, interface traction and energies.

`adapt_general_coupled_subsystems_result` function

```
adapt_general_coupled_subsystems_result(result: object, *, result_id: str =
    'multiphysics.general_coupled_subsystems.result', plan_reference: ArtifactReference | None
    = None, diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish named subsystem DOFs and block-coupling diagnostics.

adapt_magneto_mechanics_result function

```
adapt_magneto_mechanics_result(result: object, *, result_id: str =
    'multiphysics.magneto_mechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish magnetostriction, Maxwell-pressure and mechanical fields.

adapt_reactive_coupling_result function

```
adapt_reactive_coupling_result(result: object, *, result_id: str =
    'multiphysics.reactive_coupling.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish arbitrary species densities, temperature and balance data.

adapt_thermo_mechanics_result function

```
adapt_thermo_mechanics_result(result: object, *, result_id: str =
    'multiphysics.thermo_mechanics.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish thermoelastic fields and coupled energy/balance diagnostics.

AdiabaticReaction class

```
AdiabaticReaction(*, domain: object, model_id: str = <factory>, network: ReactionNetworkIR,
    volumetric_heat_capacity: float, kinetics: object = None, thermal_source: object = 0.0,
    species_sources: Mapping = <factory>) -> None
```

Advance lumped species and temperature using a reaction network and volumetric heat capacity.

Selected methods.

```
concentration(species)
```

```
source(species)
```

DEFAULT_MULTIPHYSICS_PROVIDER_REGISTRY value

Default registry for the domain provider route. Inspect the registered IDs when choosing a provider.

ElectroThermalConfiguration class

```
ElectroThermalConfiguration(provider: str = 'multiphysics.electro_thermal', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'joule_heating_fixed_point',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and electro-thermal parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`ElectroThermalProvider` class

```
ElectroThermalProvider(*, cell_count: int = 64, density: float = 1.0, heat_capacity: float =
    1.0, thermal_conductivity: float = 1.0, electrical_conductivity: float = 1.0,
    conductivity_temperature_coefficient: float = 0.0, maximum_conductivity_exponent: float =
    50.0, reference_temperature: float = 0.0, permittivity: float = 1.0,
    applied_electric_field: FieldSource | None = 1.0, charge_density: np.ndarray | None =
    None, external_heat_source: FieldSource = 0.0, initial_temperature: np.ndarray | None =
    None, final_time: float = 0.0, time_step: float = 0.0001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate conductive heat transfer with temperature-dependent Joule heating.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.electro_thermal.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish electro-thermal fields and coupling diagnostics.

```
execute(problem: object | None = None) -> ElectroThermalResult
```

Advance coupled temperature and evaluate electrical/Joule fields.

`ElectroThermalPublicApi` class

```
ElectroThermalPublicApi(*, provider_registry=None) -> None
```

Execute electro-thermal cases through the multiphysics registry.

Selected methods.

```
validate(configuration: ElectroThermalConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: ElectroThermalConfiguration)
```

Validate and execute the selected provider.

`ElectroThermalResult` class

```
ElectroThermalResult(coordinates: np.ndarray, state: ElectroThermalState, electric_field:
    np.ndarray, conductivity: np.ndarray, current_density: np.ndarray, joule_heat: np.ndarray,
    time: float, step_count: int, thermal_balance_residual: float, coupling_graph_accepted:
    bool, electric_field_result: ElectricFieldResult | None) -> None
```

Final temperature, electrical state, Joule source, and balance data.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return serializable copied fields and coupling diagnostics.

`ElectroThermalState` class

```
ElectroThermalState(temperature: np.ndarray) -> None
```

Temperature field used by the coupled electrical and thermal balance.

`FluidColumnPiston` class

```
FluidColumnPiston(*, domain: object, model_id: str = <factory>, piston_mass: float,
    fluid_density: float, column_length: float, area: float, spring_stiffness: float,
    external_force: float | None = 0.0) -> None
```

Integrate piston motion with the effective inertia of a fluid column and a spring.

Declared fields. `displacement` (m, differential), `velocity` (m/s, differential), `interface_pressure` (Pa, derived), `stored_energy` (J, derived), `load` (N, input).

`FluidStructureConfiguration` class

```
FluidStructureConfiguration(provider: str = 'multiphysics.fluid_structure', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'monolithic_interface_momentum',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and FSI interface parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`FluidStructureProvider` class

```
FluidStructureProvider(*, cell_count: int = 64, fluid_density: float = 1.0, fluid_viscosity:
    float = 0.01, structural_density: float = 1.0, elastic_modulus: float = 1.0,
    structural_damping: float = 0.0, interface_impedance: float = 1.0, interface_pressure:
    FieldSource = 0.0, fluid_body_force: FieldSource = 0.0, initial_fluid_velocity: np.ndarray
    | None = None, initial_structural_displacement: np.ndarray | None = None,
    initial_structural_velocity: np.ndarray | None = None, final_time: float = 0.0, time_step:
    float = 0.0001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate a partition-independent linear fluid-structure interface model.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.fluid_structure.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish fluid/structure fields, interface traction and energies.

```
execute(problem: object | None = None) -> FluidStructureResult
```

Advance fluid/structure fields and return interface diagnostics.

`FluidStructurePublicApi` class

```
FluidStructurePublicApi(*, provider_registry=None) -> None
```

Execute FSI cases through the multiphysics registry.

Selected methods.

```
validate(configuration: FluidStructureConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: FluidStructureConfiguration)
```

Validate and execute the selected provider.

`FluidStructureResult` class

```
FluidStructureResult(coordinates: np.ndarray, state: FluidStructureState, interface_traction:
    np.ndarray, time: float, step_count: int, fluid_kinetic_energy: float,
    structural_kinetic_energy: float, structural_strain_energy: float, interface_power: float,
    interface_residual: float) -> None
```

Final coupled fields, interface traction, energy, and residual data.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied coupled fields and energy/interface diagnostics.

`FluidStructureState` class

```
FluidStructureState(fluid_velocity: np.ndarray, structural_displacement: np.ndarray,
    structural_velocity: np.ndarray) -> None
```

Fluid velocity, structural displacement, and structural velocity fields.

`GeneralCoupledSubsystemsConfiguration` class

```
GeneralCoupledSubsystemsConfiguration(provider: str = 'multiphysics.general_coupled_subsystems'
    , mesh: str = 'core.dof.0d', discretization: str = 'block_state', time_integration: str =
    'classical_rk4', solver: str = 'monolithic_or_partitioned_linear_coupling', parameters:
    Mapping[str, object] = <factory>) -> None
```

Select coupling execution semantics and subsystem matrix parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this configuration into the shared Core configuration.

`GeneralCoupledSubsystemsProvider` class

```
GeneralCoupledSubsystemsProvider(*, subsystem_ids: Sequence[str] = ('subsystem_a',
    'subsystem_b'), subsystem_sizes: Sequence[int] = (1, 1), system_matrix:
    Sequence[Sequence[float]] = ((-1.0, 1.0), (1.0, -1.0)), source: Source = (0.0, 0.0),
    initial_state: Sequence[float] = (1.0, 0.0), coupling_mode: str = 'monolithic',
    partition_iterations: int = 8, partition_tolerance: float = 1e-10, relaxation: float =
    1.0, final_time: float = 0.0, time_step: float = 0.001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate arbitrary linear coupled subsystems with Core coupling contracts.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.general_coupled_subsystems.res
    ult', plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish named subsystem DOFs and block-coupling diagnostics.

```
coupling_graph() -> CouplingGraph
```

Build and validate Core coupling entities for active cross blocks.

```
execute(problem: object | None = None) -> GeneralCoupledSubsystemsResult
```

Advance the configured coupled system in monolithic or partitioned mode.

`GeneralCoupledSubsystemsPublicApi` class

```
GeneralCoupledSubsystemsPublicApi(*, provider_registry=None) -> None
```

Execute generic coupled systems through the multiphysics registry.

Selected methods.

```
validate(configuration: GeneralCoupledSubsystemsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: GeneralCoupledSubsystemsConfiguration)
```

Validate and execute the selected provider.

`GeneralCoupledSubsystemsResult` class

```
GeneralCoupledSubsystemsResult(state: GeneralCoupledSubsystemsState, derivative: np.ndarray,
    time: float, step_count: int, coupling_residual: float, partition_iterations: int,
    graph_accepted: bool, graph_diagnostics: tuple[str, ...]) -> None
```

Coupled state, graph validation, and partition-iteration diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return state, subsystem mapping, and execution diagnostics.

`GeneralCoupledSubsystemsState` class

```
GeneralCoupledSubsystemsState(values: np.ndarray, subsystem_ids: tuple[str, ...],
    subsystem_sizes: tuple[int, ...]) -> None
```

One concatenated state vector with named subsystem slices.

Selected methods.

```
subsystem_mapping() -> dict[str, np.ndarray]
```

Return independent copies keyed by stable subsystem identity.

`HeatedResistor` class

```
HeatedResistor(*, domain: object, model_id: str = <factory>, resistance: float, heat_capacity:
    float, applied_voltage: float | None, thermal_power: float | None = 0.0,
    temperature_coefficient: float = 0.0, reference_temperature: float = 300.0) -> None
```

Integrate a temperature-dependent resistor's heating balance under applied voltage and thermal input.

Declared fields. `temperature` (K, differential), `resistance` (Ohm, derived), `current` (A, derived), `joule_power` (W, derived), `thermal_energy` (J, derived), `supply` (V, input), `heat_source` (W, input).

`MagnetoMechanicsConfiguration` class

```
MagnetoMechanicsConfiguration(provider: str = 'multiphysics.magneto_mechanics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'coupled_magnetostriction_momentum',
     parameters: Mapping[str, object] = <factory>) -> None
```

Select Core numerical contracts and magneto-mechanical parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this public configuration to the shared Core contract.

`MagnetoMechanicsProvider` class

```
MagnetoMechanicsProvider(*, cell_count: int = 64, structural_density: float = 1.0,
    youngs_modulus: float = 1.0, poisson_ratio: float = 0.0, magnetic_permeability: float =
    1.0, magnetostriction_coefficient: float = 0.0, magnetic_reference_field: float = 1.0,
    magnetic_force_coefficient: float = 1.0, structural_damping: float = 0.0,
    applied_magnetic_field: FieldSource | None = 0.0, current_density: np.ndarray | None =
    None, initial_displacement: np.ndarray | None = None, initial_velocity: np.ndarray | None
    = None, final_time: float = 0.0, time_step: float = 0.0001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Integrate magnetostriction and Maxwell-stress loading of an elastic bar.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.magneto_mechanics.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish magnetostriction, Maxwell-pressure and mechanical fields.

```
execute(problem: object | None = None) -> MagnetoMechanicsResult
```

Advance displacement and velocity and return the final coupled balance.

`MagnetoMechanicsPublicApi` class

```
MagnetoMechanicsPublicApi(*, provider_registry=None) -> None
```

Validate and execute magneto-mechanical cases through the registry.

Selected methods.

```
validate(configuration: MagnetoMechanicsConfiguration) -> tuple[str, ...]
```

Return public-boundary errors without modifying simulation state.

```
run(configuration: MagnetoMechanicsConfiguration)
```

Validate and execute the selected registered coupling provider.

`MagnetoMechanicsResult` class

```
MagnetoMechanicsResult(coordinates: np.ndarray, state: MagnetoMechanicsState, magnetic_field:
    np.ndarray, magnetic_strain: np.ndarray, strain: np.ndarray, stress: np.ndarray,
    magnetic_pressure: np.ndarray, magnetic_force: np.ndarray, time: float, step_count: int,
    momentum_balance_residual: float, elastic_energy: float, magnetic_energy: float,
    magnetic_field_result: MagneticFieldResult | None) -> None
```

Final coupled fields, energies, residual, and optional magnetic solve.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied fields and scalar diagnostics suitable for serialization.

`MagnetoMechanicsState` class

```
MagnetoMechanicsState(displacement: np.ndarray, velocity: np.ndarray) -> None
```

Displacement and velocity of the mechanically active one-dimensional body.

`MULTIPHYSICS` value

Public identity for the multiphysics methodology.

`MultiphysicsProviderRegistry` class

```
MultiphysicsProviderRegistry() -> None
```

Register multiphysics providers by stable identifier.

Selected methods.

```
register(provider_id: str, factory: Callable[..., object], *, catalog_role: str =
    'unclassified') -> None
```

```
catalog_role(provider_id: str) -> str
```

```
create(provider_id: str, **parameters: object) -> object
```

```
ids() -> tuple[str, ...]
```

`NUMERICS_BINDING` value

Public binding value for the multiphysics numerical interface.

`ReactiveCouplingConfiguration` class

```
ReactiveCouplingConfiguration(provider: str = 'multiphysics.reactive_coupling', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference_operator_split',
    time_integration: str = 'reaction_provider_plus_explicit_transport', solver: str =
    'plasma_reaction_network', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared Core contracts and user-extensible reactive parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the common numerical configuration boundary.

`ReactiveCouplingProvider` class

```
ReactiveCouplingProvider(*, species: SpeciesSet, reaction_network: ReactionNetwork,
    initial_densities: Mapping[str, np.ndarray], initial_temperature: np.ndarray, cell_count:
    int | None = None, species_diffusivities: Mapping[str, float] | None = None,
    default_species_diffusivity: float = 0.0, thermal_diffusivity: float = 0.0,
    temperature_capacity: float = 1.0, reaction_integration: str = 'explicit_euler',
    strict_conservation: bool = False, positivity_floor: float = 0.0, final_time: float = 0.0,
    time_step: float = 0.0001, **_: object) -> None
```

Couple a user-defined plasma reaction network to Core transport operators.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.reactive_coupling.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish arbitrary species densities, temperature and balance data.

```
execute(problem: object | None = None) -> ReactiveCouplingResult
```

Advance reaction sources and diffusion on one shared Core layout.

`ReactiveCouplingPublicApi` class

```
ReactiveCouplingPublicApi(*, provider_registry=None) -> None
```

Validate and execute reactive coupling through the provider registry.

Selected methods.

```
validate(configuration: ReactiveCouplingConfiguration) -> tuple[str, ...]
```

Return public-boundary validation errors.

```
run(configuration: ReactiveCouplingConfiguration)
```

Validate and execute the configured reaction–transport case.

`ReactiveCouplingResult` class

```
ReactiveCouplingResult(coordinates: np.ndarray, state: ReactiveCouplingState, time: float,
    step_count: int, reaction_energy_source: float, species_diffusion_residual: float,
    thermal_balance_residual: float) -> None
```

Final plasma state, transport diagnostics and coupled energy residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied extensible species fields and coupling diagnostics.

`ReactiveCouplingState` class

```
ReactiveCouplingState(plasma_state: PlasmaState) -> None
```

Reacting multi-species state represented by the shared `PlasmaState`.

`ThermalExchange` class

```
ThermalExchange(*, domain: object, model_id: str = <factory>, first_capacity: float,
    second_capacity: float, conductance: float, first_power: float | None = 0.0, second_power:
    float | None = 0.0) -> None
```

Integrate two thermal capacities connected by a conductance, with independent power inputs.

Declared fields. `first_temperature` (K, differential), `second_temperature` (K, differential), `heat_flow` (W, derived), `stored_energy` (J, derived), `first_source` (W, input), `second_source` (W, input).

`ThermoMechanicsConfiguration` class

```
ThermoMechanicsConfiguration(provider: str = 'multiphysics.thermo_mechanics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'coupled_thermoelastic_momentum',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select shared numerical contracts and thermoelastic parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this API configuration to the shared Core contract.

`ThermoMechanicsProvider` class

```
ThermoMechanicsProvider(*, cell_count: int = 64, density: float = 1.0, heat_capacity: float =
    1.0, thermal_conductivity: float = 1.0, youngs_modulus: float = 1.0, poisson_ratio: float
    = 0.0, thermal_expansion: float = 0.0, reference_temperature: float = 1.0,
    structural_damping: float = 0.0, damping_heat_fraction: float = 1.0, external_heat_source:
    FieldSource = 0.0, initial_temperature: np.ndarray | None = None, initial_displacement:
    np.ndarray | None = None, initial_velocity: np.ndarray | None = None, final_time: float =
    0.0, time_step: float = 0.0001, time_integration: str = 'classical_rk4', **_: object) ->
    None
```

Integrate thermoelastic deformation and thermal conduction without solver duplication.

Selected methods.

```
result_adapter(result: object, *, result_id: str = 'multiphysics.thermo_mechanics.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish thermoelastic fields and coupled energy/balance diagnostics.

```
execute(problem: object | None = None) -> ThermoMechanicsResult
```

Advance thermal and structural fields on the shared Core grid.

`ThermoMechanicsPublicApi` class

```
ThermoMechanicsPublicApi(*, provider_registry=None) -> None
```

Validate and execute thermo-mechanics through the multiphysics registry.

Selected methods.

```
validate(configuration: ThermoMechanicsConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: ThermoMechanicsConfiguration)
```

Validate and execute the selected coupling provider.

`ThermoMechanicsResult` class

```
ThermoMechanicsResult(coordinates: np.ndarray, state: ThermoMechanicsState, thermal_strain:
    np.ndarray, strain: np.ndarray, stress: np.ndarray, dissipation_heat: np.ndarray, time:
    float, step_count: int, thermal_balance_residual: float, momentum_balance_residual: float,
    elastic_energy: float, thermal_energy: float, coupling_graph_accepted: bool) -> None
```

Coupled fields, thermoelastic strain/stress and balance diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied fields and scalar coupled-balance diagnostics.

`ThermoMechanicsState` class

```
ThermoMechanicsState(temperature: np.ndarray, displacement: np.ndarray, velocity: np.ndarray)
    -> None
```

Temperature, displacement and velocity on the shared one-dimensional mesh.

`VoiceCoil` class

```
VoiceCoil(*, domain: object, model_id: str = <factory>, resistance: float, inductance: float,
    force_factor: float, mass: float, spring_stiffness: float, damping: float,
    applied_voltage: float | None, external_force: float | None = 0.0) -> None
```

Integrate the coupled current, displacement and velocity of an electromechanical actuator.

Declared fields. `current` (A, differential), `displacement` (m, differential), `velocity` (m/s, differential), `magnetic_force` (N, derived), `back_emf` (V, derived), `stored_energy` (J, derived), `input_power` (W, derived), `dissipated_power` (W, derived), `supply` (V, input), `load` (N, input).

A.14 onesim.physics.optics

`adapt_optics_result` function

```
adapt_optics_result(result: OpticsResult, *, result_id: str = 'waves.optics.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish a complex paraxial envelope and intensity on transverse nodes.

`adapt_photonics_result` function

```
adapt_photonics_result(result: PhotonicsResult, *, result_id: str = 'waves.photonics.result',
    plan_reference: ArtifactReference | None = None, diagnostics: Sequence[str] = ()) ->
    SimulationResultSet
```

Publish photon density plus the delegated optical observables.

`ComplexParaxialEnvelope` class

```
ComplexParaxialEnvelope(*, domain: object, model_id: str = <factory>, wavenumber: float,
    group_speed: float, permittivity: float, reference_index: float = 1.0,
    refractive_index_perturbation: float = 0.0, absorption: float = 0.0, source: object = 0.0)
    -> None
```

One genuinely complex electric-envelope field in the travelling frame.

Declared fields. `envelope` (V/m, differential), `intensity` (W/m^2, derived), `driving` (V/m/s, input).

`OpticsConfiguration` class

```
OpticsConfiguration(provider: str = 'waves.optics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'paraxial_finite_difference',
    time_integration: str = 'classical_rk4', solver: str = 'scalar_paraxial_envelope',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select Core propagation contracts and optical parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to common Core numerical configuration.

`OpticsProvider` class

```
OpticsProvider(*, cell_count: int = 64, wavenumber: float = 1.0, reference_index: float = 1.0,
    refractive_index_perturbation: FieldSource = 0.0, absorption: FieldSource = 0.0,
    source_real: FieldSource = 0.0, source_imaginary: FieldSource = 0.0, initial_real:
    np.ndarray | None = None, initial_imaginary: np.ndarray | None = None,
    propagation_distance: float = 0.0, step_size: float = 0.0001, time_integration: str =
    'classical_rk4', **unknown: object) -> None
```

Propagate a scalar paraxial envelope with Core spatial/time infrastructure.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.optics') -> CallableSpatialOperator
```

Expose paraxial envelope propagation through the common Study.

```
execute(problem: object | None = None) -> OpticsResult
```

Propagate real/imaginary envelope components along optical distance.

`OpticsPublicApi` class

```
OpticsPublicApi(*, provider_registry=None) -> None
```

Validate and execute optics through the waves registry.

Selected methods.

```
validate(c: OpticsConfiguration) -> tuple[str, ...]
```

```
run(c: OpticsConfiguration)
```

Validate and execute optics provider.

`OpticsResult` class

```
OpticsResult(coordinates: np.ndarray, state: OpticsState, distance: float, step_count: int,
    intensity: np.ndarray, power: float, residual: float) -> None
```

Paraxial field, intensity, power and propagation residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied envelope components, intensity and diagnostics.

`OpticsState` class

```
OpticsState(real: np.ndarray, imaginary: np.ndarray) -> None
```

Complex envelope represented by finite real and imaginary 1D fields.

`ParaxialEnvelope` class

```
ParaxialEnvelope(*, domain: object, model_id: str = <factory>, wavenumber: float, group_speed:
    float, permittivity: float, reference_index: float = 1.0, refractive_index_perturbation:
    float = 0.0, absorption: float = 0.0, source: object = 0.0) -> None
```

Envelope in a frame travelling distance z=group_speed*t.

Declared fields. `real` (V/m, differential), `imaginary` (V/m, differential), `intensity` (W/m^2, derived), `driving` (V/m/s, input).

`PhotonEnvelope` class

```
PhotonEnvelope(*, domain: object, model_id: str = <factory>, wavenumber: float, group_speed:
    float, permittivity: float, reference_index: float = 1.0, refractive_index_perturbation:
    float = 0.0, absorption: float = 0.0, source: object = 0.0, photon_energy: float) -> None
```

Advance the paraxial envelope with a photon-energy scale for related observables.

Declared fields. `real` (V/m, differential), `imaginary` (V/m, differential), `intensity` (W/m^2, derived), `driving` (V/m/s, input), `photon_flux_density` (1/m^2/s, derived), `photon_density` (1/m^3, derived).

`PhotonicsConfiguration` class

```
PhotonicsConfiguration(provider: str = 'waves.photonics', mesh: str =
    'core.coordinate_projection.1d', discretization: str = 'delegated_paraxial_photonics',
    time_integration: str = 'classical_rk4', solver: str = 'optics_provider_composition',
    parameters: Mapping[str, object] = <factory>) -> None
```

Select delegated optics contracts and photon normalization.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the shared Core numerical configuration.

`PhotonicsProvider` class

```
PhotonicsProvider(*, photon_energy: float = 1.0, **optics_parameters: object) -> None
```

Evaluate photon observables by delegating field propagation to Optics.

Selected methods.

```
execute(problem: object | None = None) -> PhotonicsResult
```

Delegate optical propagation and convert intensity to photon observables.

```
as_spatial_operator(*, method_id: str = 'spatial-method:fdm.photonics') ->
    CallableSpatialOperator
```

Expose delegated optics and photon observables through Core.

`PhotonicsPublicApi` class

```
PhotonicsPublicApi(*, provider_registry=None) -> None
```

Validate and execute photonics via the waves registry.

Selected methods.

```
validate(c: PhotonicsConfiguration) -> tuple[str, ...]
```

```
run(c: PhotonicsConfiguration)
```

Validate and execute selected photonics provider.

`PhotonicsResult` class

```
PhotonicsResult(state: PhotonicsState, optics_result: OpticsResult, photon_energy: float) ->
    None
```

Optical solution plus photon density, flux and number.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return photon observables together with the delegated optical result.

`PhotonicsState` class

```
PhotonicsState(photon_density: np.ndarray) -> None
```

Photon-density field derived from an optical envelope state.

A.15 onesim.physics.plasma

`AbsorbingPlasmaWall` class

```
AbsorbingPlasmaWall(*, boundary: object, physics: object, secondary_electron_yield: float =
    0.0, secondary_electron_energy: object | None = None, provider: object | None = None) ->
    None
```

Declare the absorbing wall particle and energy fluxes, with optional secondary emission.

Selected methods.

```
bind_wall(*, geometry)
```

```
semantic_conditions(condition_id)
```

`ChargedParticle` class

```
ChargedParticle(*, domain: object, model_id: str = <factory>, mass: float, charge: float,
    electric: tuple[float | None, float | None, float | None] = (0.0, 0.0, 0.0), magnetic:
    tuple[float | None, float | None, float | None] = (0.0, 0.0, 0.0)) -> None
```

Integrate position and velocity under the Lorentz force from prescribed or coupled fields.

Declared fields. `x` (m, differential), `y` (m, differential), `z` (m, differential), `velocity_x` (m/s, differential), `velocity_y` (m/s, differential), `velocity_z` (m/s, differential), `electric_x` (V/m, input), `electric_y` (V/m, input), `electric_z` (V/m, input), `magnetic_x` (T, input), `magnetic_y` (T, input), `magnetic_z` (T, input).

Selected methods.

```
electric_input(axis)
```

```
magnetic_input(axis)
```

`ColdPlasmaOscillation` class

```
ColdPlasmaOscillation(*, domain: object, model_id: str = <factory>, density: float,
    electron_mass: float, elementary_charge: float, permittivity: float, collision_frequency:
    float, external_current: float | None = 0.0) -> None
```

Integrate homogeneous electron velocity, electric field and thermal energy with collisions and applied current.

Declared fields. `velocity` (m/s, differential), `electric_field` (V/m, differential), `thermal_energy` (J/m^3, differential), `imposed_current` (A/m^2, input).

`create_plasma_factories` function

```
create_plasma_factories(*, physics_registry=None, coupling_registry=None)
```

Construct independently owned plasma and microwave factory adapters.

`create_plasma_runtime` function

```
create_plasma_runtime(*, physics_registry=None, coupling_registry=None, **dependencies)
```

Select the existing plasma bundle explicitly within the common runtime.

`DriftDiffusionPlasma` class

```
DriftDiffusionPlasma(*, domain: object, model_id: str = <factory>, gas: object, chemistry:
    object, electron_energy: object, electron_transport: object, ion_transport: object,
    electron_species: str = 'electron', prescribed_inputs: object = <factory>) -> None
```

Declare the transported species and mean electron energy system with explicit gas, chemistry and transport collaborators.

Selected methods.

```
field(name)
```

```
fields()
```

```
density(species)
```

```
initialization_targets(reference)
```

```
lower_initial_conditions(initials)
```

`EcrExcitation` class

```
EcrExcitation(*, frequency: float, power: float, impressed_current_density: object | None =
    None, port: object | None = None, absorption: float = 1.0) -> None
```

Set microwave frequency and power with an explicit current or port excitation as supported.

`ElectronCollisionModel` class

```
ElectronCollisionModel(*, gas: object, chemistry: object, energy_distribution: object,
    provider: object | None = None) -> None
```

Specify how the gas, chemistry and electron energy distribution determine electron collisions.

Selected methods.

```
to_mapping()
```

```
collision_frequency(*, mean_electron_energy, points=None, time=0.0)
```

```
derivative(*, mean_electron_energy, points=None, time=0.0)
```

```
spatial_degree(energy_degree)
```

`FreeStreamingWaterbag` class

```
FreeStreamingWaterbag(*, domain: object, model_id: str = <factory>, phase_density: float,
    particle_mass: float) -> None
```

f(x,v)=phase_density between lower/upper contours, zero outside.

Declared fields. `lower_velocity` (m/s, differential), `upper_velocity` (m/s, differential), `number_density` (1/m^3, derived), `bulk_velocity` (m/s, derived), `kinetic_energy_density` (J/m^3, derived).

Selected methods.

```
reflect_boundary_state(interior, *, normal)
```

```
reflect_boundary_state_jacobian(interior, *, normal)
```

`IonMobilityModel` class

```
IonMobilityModel(*, gas: object, chemistry: object, provider: object | None = None) -> None
```

Select ion mobility using the gas and chemistry contracts.

Selected methods.

```
to_mapping()
```

```
coefficients(*, species, points=None, time=0.0)
```

`IsothermalElectronWaterbag` class

```
IsothermalElectronWaterbag(*, domain: object, model_id: str = <factory>, phase_density: float,
    particle_mass: float, electron_temperature: float) -> None
```

Advance an electron waterbag with explicit particle mass and electron temperature.

Declared fields. `lower_velocity` (m/s, differential), `upper_velocity` (m/s, differential), `number_density` (1/m^3, derived), `bulk_velocity` (m/s, derived), `kinetic_energy_density` (J/m^3, derived).

`MagnetizedColdPlasma` class

```
MagnetizedColdPlasma(*, collision_model: object, model_id: str = <factory>, property_service:
    object | None = None) -> None
```

Compose the cold-plasma material response using a collision model and optional property service.

Declared fields. `electron_density` (1/m^3, input), `mean_electron_energy` (J, input), `magnetic_field` (T, input, x/y/z).

Selected methods.

```
fields()
```

```
integration_power_degree(*, electric_degree, input_degrees)
```

```
integration_response_degree(input_degrees)
```

```
port(name)
```

```
evaluate(*, electron_density, mean_electron_energy, magnetic_field, frequency,
    phasor_convention, amplitude_convention, points=None, time=0.0)
```

`MagnetizedElectronTransport` class

```
MagnetizedElectronTransport(*, collisions: object, provider: object | None = None) -> None
```

Select an electron transport model with an explicit collision collaborator.

Selected methods.

```
to_mapping()
```

```
coefficients(*, mean_electron_energy, magnetic_field, points=None, time=0.0, linearize=True)
```

```
magnetic_direction(*, mean_electron_energy, magnetic_field, direction, points=None, time=0.0)
```

`Maxwellian` class

```
Maxwellian(*, relative_tolerance: float = 1e-08) -> None
```

Use a Maxwellian electron energy distribution with an explicit numerical tolerance.

`MeanElectronEnergyEquation` class

```
MeanElectronEnergyEquation(*, enthalpy_factor: float = 1.6666666666666667, provider: object |
    None = None) -> None
```

Declare the electron-energy balance and enthalpy factor for a fluid plasma.

Selected methods.

```
to_mapping()
```

```
bind(*, geometry, model)
```

NativeEcrDiscretization class

```
NativeEcrDiscretization(*, maxwell: object, numerical_service: object | None = None,
    maxwell_registry: object | None = None) -> None
```

Select native FV transport/Poisson and an explicit Maxwell discretization.

Selected methods.

```
validate(model, mesh, numerics)
```

```
bind_operator(*, model, mesh, numerics, backend, dependencies, boundary_conditions=(),
    inputs=None, initial_conditions=(), constraints=(), **kwargs)
```

```
preflight(*, composition, model)
```

```
bind_transient_subsystem(composition, model, *, start, adaptive)
```

NativeEcrPlasma class

```
NativeEcrPlasma(*, domain: object, model_id: str = <factory>, chemistry: object,
    transport_model: object, excitation: EcrExcitation, walls: tuple, plasma_parameters:
    object = <factory>, potential_gauge: NativeEcrPotentialGauge | None = None, wall_registry:
    object | None = None) -> None
```

Existing native species/energy/Poisson/Maxwell physical composition.

Selected methods.

```
public_name(name)
```

```
floating_potential(boundary_id)
```

```
density(species)
```

```
to_mapping()
```

NativeEcrPotentialGauge class

```
NativeEcrPotentialGauge(*, mean_potential: float = 0.0, absolute_charge_tolerance: float =
    0.0, relative_charge_tolerance: float = 1e-08) -> None
```

Volume-mean potential in V and a checked zero charge multiplier.

NativeEcrStateTransfer class

```
NativeEcrStateTransfer(*, cell_projection: object, maxwell_projection: object, history_policy:
    str = 'reset', physical_history_transfer: object | None = None, terminal_projection:
    object | None = None) -> None
```

Transfer all species, energy, potential and complex Maxwell moments.

Selected methods.

```
with_history_policy(policy)
```

```
bind(*, source, source_simulation, target_simulation)
```

`NoParticleFlux` class

```
NoParticleFlux(*, boundary: object, physics: object, provider: object | None = None) -> None
```

Declare the zero particle-flux wall for the owned plasma model.

Selected methods.

```
bind_wall(*, geometry)
```

```
semantic_conditions(condition_id)
```

`operators` namespace

Plasma / operators.

`PlasmaTransportCoefficients` class

```
PlasmaTransportCoefficients(mobility: object, diffusivity: object, mobility_energy_derivative:
    object, diffusivity_energy_derivative: object) -> None
```

Carry mobilities, diffusivities and their energy derivatives for a bound plasma transport model.

`PlasmaWallFlux` class

```
PlasmaWallFlux(particles: object, electron_energy: object) -> None
```

Carry the wall particle and electron energy flux values returned by a wall law.

`UniformGuidingCenter` class

```
UniformGuidingCenter(*, domain: object, model_id: str = <factory>, mass: float,
    magnetic_field: float | None, magnetic_moment: float, electric_x: float | None = 0.0,
    electric_y: float | None = 0.0) -> None
```

Integrate reduced particle motion in a spatially uniform field model.

Declared fields. `x` (m, differential), `y` (m, differential), `z` (m, differential), `parallel_velocity` (m/s, differential), `field_x` (V/m, input), `field_y` (V/m, input), `field_b` (T, input).

Selected methods.

```
electric_input(axis)
```

A.16 onesim.physics.solid_mechanics

`BilinearCohesiveLaw` class

```
BilinearCohesiveLaw(*, peak_traction: float, onset_opening: float, final_opening: float) ->
    None
```

Evaluate a bilinear irreversible traction-separation law.

Selected methods.

```
traction_derivative(opening)
```

Selected branch tangent for monotonic opening without prior damage.

```
damage_derivative(opening)
```

```
evaluate(opening: float, previous_damage: float = 0.0) -> CohesiveZoneResult
```

`BodyForce` class

```
BodyForce(*, value: object) -> None
```

Supply a Cartesian force per volume to the elastic model.

`CohesiveOpening` class

```
CohesiveOpening(*, domain: object, model_id: str = <factory>, opening_rate: float | None,
    cohesive_law: object) -> None
```

Advance opening and damage with a selected traction–separation law.

Declared fields. `opening` (m, differential), `work_per_area` (J/m^2, differential), `traction` (Pa, derived), `damage` (1, derived), `imposed_rate` (m/s, input).

`FatigueCrackGrowth` class

```
FatigueCrackGrowth(*, domain: object, model_id: str = <factory>, stress_range: float | None,
    frequency: float | None, growth_law: object, geometry_factor: float = 1.0) -> None
```

Advance a crack-length state using the selected cyclic growth law and load range.

Declared fields. `crack_length` (m, differential), `loading_range` (Pa, input), `cycling_rate` (1/s, input).

Selected methods.

```
rate(crack_length, *, stress_range=None, frequency=None)
```

`FixedDisplacement` class

```
FixedDisplacement(*, boundary: object, physics: object, value: object) -> None
```

Prescribe Cartesian displacement on a selected elastic boundary.

`HertzContactLaw` class

```
HertzContactLaw(effective_modulus: float, effective_radius: float) -> None
```

Compute elastic contact response from effective modulus and radius.

Selected methods.

```
force(indentation)
```

```
stored_energy(indentation)
```

```
force_derivative(indentation)
```

```
stored_energy_derivative(indentation)
```

`HertzImpact` class

```
HertzImpact(*, domain: object, model_id: str = <factory>, reduced_mass: float, contact_law:
    object, force: float | None = 0.0) -> None
```

Advance normal contact motion using a Hertz contact law and reduced mass.

Declared fields. `indentation` (m, differential), `approach_velocity` (m/s, differential), `contact_force` (N, derived), `mechanical_energy` (J, derived), `load` (N, input).

`IsotropicThermalExpansion` class

```
IsotropicThermalExpansion(*, reference_temperature: float) -> None
```

Set the stress-free reference temperature for isotropic thermal strain.

Selected methods.

```
evaluate(*, temperature, coefficient)
```

```
evaluate_array(*, temperature, coefficient)
```

The same physical expansion law on explicitly supplied point states.

`LinearElasticity` class

```
LinearElasticity(*, domain: object, model_id: str = <factory>, material: object,
    thermal_strain: IsotropicThermalExpansion | None = None, loading: object | None = None) ->
    None
```

Solve displacement and stress from an elastic material, loads and optional thermal strain.

Declared fields. `displacement` (m, algebraic, x/y/z), `temperature` (K, input), `body_force` (N/m^3, input, x/y/z), `stress` (Pa, derived, xx/yy/zz/xy/yz/xz), `von_mises_stress` (Pa, derived), `thermal_strain_tensor` (1, derived, xx/yy/zz/xy/yz/xz).

`MaxwellRelaxation` class

```
MaxwellRelaxation(*, domain: object, model_id: str = <factory>, youngs_modulus: float,
    viscosity: float, strain_rate: float | None = 0.0) -> None
```

Advance the viscoelastic stress of a Maxwell element driven by strain rate.

Declared fields. `stress` (Pa, differential), `imposed_strain_rate` (1/s, input).

`ParisLaw` class

```
ParisLaw(coefficient: float, exponent: float) -> None
```

da/dN=C*DeltaK**m, with SI DeltaK in Pa*sqrt(m), a in m, N cycles.

Selected methods.

```
advance_per_cycle(stress_intensity_range)
```

```
rate_derivative(stress_intensity_range)
```

`StructuralOscillator` class

```
StructuralOscillator(*, domain: object, model_id: str = <factory>, mass: float, stiffness:
    float, damping: float = 0.0, force: float | None = 0.0) -> None
```

Advance displacement and velocity for a single mass, spring and damper.

Declared fields. `displacement` (m, differential), `velocity` (m/s, differential), `mechanical_energy` (J, derived), `load` (N, input).

`Traction` class

```
Traction(*, boundary: object, physics: object, value: object) -> None
```

Prescribed Cartesian surface force sigma*n, in Pa, on a physical face.

Selected methods.

```
traction(*, points, time)
```

`TractionFree` class

```
TractionFree(*, boundary: object, physics: object) -> None
```

Set zero surface traction on a selected elastic boundary.

Selected methods.

```
traction(*, points, time)
```

A.17 onesim.physics.systems

`adapt_control_signal_flow_result` function

```
adapt_control_signal_flow_result(result: ControlSignalFlowResult, *, result_id: str =
    'systems.control_signal_flow.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish signal-block outputs, dynamic states, and graph residual.

`adapt_electrical_networks_result` function

```
adapt_electrical_networks_result(result: ElectricalNetworksResult, *, result_id: str =
    'systems.electrical_networks.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish nodal voltages, branch currents, and charge balance.

`adapt_field_system_coupling_result` function

```
adapt_field_system_coupling_result(result: FieldSystemCouplingResult, *, result_id: str =
    'systems.field_system_coupling.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish conjugate field/system port values and power balance.

`adapt_hydraulic_systems_result` function

```
adapt_hydraulic_systems_result(result: HydraulicSystemsResult, *, result_id: str =
    'systems.hydraulic_systems.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish hydraulic node pressures, branch flows, and continuity balance.

`adapt_pneumatic_systems_result` function

```
adapt_pneumatic_systems_result(result: PneumaticSystemsResult, *, result_id: str =
    'systems.pneumatic_systems.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish ideal-gas chamber state, branch mass flow, and conservation.

adapt_process_models_result function

```
adapt_process_models_result(result: ProcessModelsResult, *, result_id: str =
    'systems.process_models.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish CSTR unit concentrations, temperatures, streams, and balance.

adapt_systems_result function

```
adapt_systems_result(result: object, *, provider_id: str | None = None, result_id: str =
    'systems.result', plan_reference: ArtifactReference | None = None, diagnostics:
    Sequence[str] = ()) -> SimulationResultSet
```

Adapt any built-in Systems result through one explicit entry point.

adapt_thermal_networks_result function

```
adapt_thermal_networks_result(result: ThermalNetworksResult, *, result_id: str =
    'systems.thermal_networks.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish thermal node temperatures, link heat flow, and energy balance.

adapt_zero_d_one_d_result function

```
adapt_zero_d_one_d_result(result: ZeroDOneDResult, *, result_id: str =
    'systems.zero_d_one_d_models.result', plan_reference: ArtifactReference | None = None,
    diagnostics: Sequence[str] = ()) -> SimulationResultSet
```

Publish a 0D global state or a 1D node field with its coordinates.

ControlSignalFlowConfiguration class

```
ControlSignalFlowConfiguration(provider: str = 'systems.control_signal_flow', mesh: str =
    'core.lumped.signal_graph', discretization: str = 'state_space', time_integration: str =
    'classical_rk4', solver: str = 'topological_signal_evaluation', parameters: Mapping[str,
    object] = <factory>) -> None
```

Select shared time-integration contracts and graph block parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the common Core numerical configuration.

ControlSignalFlowProvider class

```
ControlSignalFlowProvider(*, blocks: Iterable[SignalBlock], final_time: float = 0.0,
    time_step: float = 0.001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Execute an acyclic, extensible signal graph with Core time integration.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:lumped_parameter') ->
    CallableSpatialOperator
```

Expose the signal graph through the common transient Study.

```
execute(problem: object | None = None) -> ControlSignalFlowResult
```

Integrate graph state and record every block output at each time node.

`ControlSignalFlowPublicApi` class

```
ControlSignalFlowPublicApi(*, provider_registry=None) -> None
```

Validate and run a registered signal-flow provider.

Selected methods.

```
validate(configuration: ControlSignalFlowConfiguration) -> tuple[str, ...]
```

Return public-boundary configuration errors.

```
run(configuration: ControlSignalFlowConfiguration)
```

Validate and execute the selected provider.

`ControlSignalFlowResult` class

```
ControlSignalFlowResult(state: ControlSignalFlowState, times: np.ndarray, signals:
    Mapping[str, np.ndarray], state_residual: float, evaluation_order: tuple[str, ...]) -> None
```

Signal histories and state-integrator residual for a graph execution.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied histories and stable graph execution metadata.

`ControlSignalFlowState` class

```
ControlSignalFlowState(dynamic_state: np.ndarray) -> None
```

Continuous dynamic-block state vector together with the output history.

`CoupledSystemModelSpec` class

```
CoupledSystemModelSpec(system_id: str, name: str, kind: SystemModelKind, physics_id: str,
    components: tuple[SystemComponentSpec, ...], connections: tuple[SystemConnectionSpec,
    ...], field_couplings: tuple[FieldSystemCouplingSpec, ...]) -> None
```

Compose system components, connections and field links for the systems service.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'CoupledSystemModelSpec'"
```

```
to_mapping() -> dict[str, object]
```

`DEFAULT_SYSTEMS_PROVIDER_REGISTRY` value

Default registry for the domain provider route. Inspect the registered IDs when choosing a provider.

`ElectricalBranch` class

```
ElectricalBranch(branch_id: str, from_node: str, to_node: str, resistance: float = 1.0,
    series_voltage: ScalarSource = 0.0) -> None
```

Resistive connection with optional ideal series voltage.

`ElectricalNetworksConfiguration` class

```
ElectricalNetworksConfiguration(provider: str = 'systems.electrical_networks', mesh: str =
    'core.lumped.electrical_network', discretization: str = 'network_ode', time_integration:
    str = 'classical_rk4', solver: str = 'nodal_charge_balance', parameters: Mapping[str,
    object] = <factory>) -> None
```

Select the central systems provider and RC-network parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate the network request to the common Core contract.

`ElectricalNetworksProvider` class

```
ElectricalNetworksProvider(*, nodes: Iterable[ElectricalNode], branches:
    Iterable[ElectricalBranch] = (), final_time: float = 0.0, time_step: float = 0.001,
    time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate arbitrary linear RC networks using the Core integrator.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:ode.electrical_network') ->
    CallableSpatialOperator
```

Expose the lumped electrical network through the common Study.

```
execute(problem: object | None = None) -> ElectricalNetworksResult
```

Advance nodal charge balances and record branch currents.

`ElectricalNetworksPublicApi` class

```
ElectricalNetworksPublicApi(*, provider_registry=None) -> None
```

Validate and execute electrical networks through the systems registry.

Selected methods.

```
validate(configuration: ElectricalNetworksConfiguration) -> tuple[str, ...]
```

Validate every numerical axis before constructing the provider.

```
run(configuration: ElectricalNetworksConfiguration)
```

Execute a valid internal electrical-network provider.

`ElectricalNetworksResult` class

```
ElectricalNetworksResult(times: np.ndarray, voltages: dict[str, np.ndarray], currents:
    dict[str, np.ndarray], dynamic_voltages: np.ndarray, charge_balance_residual: float) ->
    None
```

Voltage/current histories and terminal charge-balance residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

`ElectricalNode` class

```
ElectricalNode(node_id: str, capacitance: float = 1.0, initial_voltage: float = 0.0,
    current_source: ScalarSource = 0.0, imposed_voltage: ScalarSource | None = None) -> None
```

Electrical potential node with capacitance or imposed voltage.

`FieldSystemCouplingProvider` class

```
FieldSystemCouplingProvider(*, effort_ratio: float = 1.0, **_: object) -> None
```

Transfer conjugate effort/flow variables without creating a solver.

Selected methods.

```
execute(*, field_effort: float, system_flow: float) -> FieldSystemCouplingResult
```

Map effort field-to-system and reaction flow system-to-field.

`FieldSystemCouplingResult` class

```
FieldSystemCouplingResult(field_effort: float, field_flow: float, system_effort: float,
    system_flow: float, field_power: float, system_power: float, power_residual: float) -> None
```

Transferred effort/flow variables and bidirectional power residual.

Selected methods.

```
to_mapping() -> dict[str, float]
```

`FieldSystemCouplingSpec` class

```
FieldSystemCouplingSpec(coupling_id: str, name: str, component_id: str, domain_ids: tuple[str,
    ...], effort_variable: str, flow_variable: str, effort_ratio: float) -> None
```

Connect one named component and spatial support using effort and flow variables and their ratio.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'FieldSystemCouplingSpec'"
```

```
to_mapping() -> dict[str, object]
```

`FirstOrderControl` class

```
FirstOrderControl(*, domain: object, model_id: str = <factory>, time_constant: float, gain:
    float = 1.0, input_signal: float | None = 1.0, input_unit: str = '1', output_unit: str =
    '1') -> None
```

Advance a first-order response driven by a signal, with explicit input and output units.

`HydraulicAccumulator` class

```
HydraulicAccumulator(*, domain: object, model_id: str = <factory>, compliance: float,
    resistance: float, reference_pressure: float | None = 0.0, external_flow: float | None =
    0.0) -> None
```

Advance global pressure with compliance, resistance and a supplied flow.

Declared fields. `pressure` (Pa, differential), `reservoir` (Pa, input), `inflow` (m^3/s, input).

`HydraulicBranch` class

```
HydraulicBranch(branch_id: str, from_node: str, to_node: str, resistance: float = 1.0,
    pressure_offset: ScalarSource = 0.0, parameters: Mapping[str, object] = <factory>,
    flow_law: Callable[[float, float, float, Mapping[str, object]], float] | None = None) ->
    None
```

Directed hydraulic connection between two nodes.

`HydraulicNode` class

```
HydraulicNode(node_id: str, compliance: float = 1.0, initial_pressure: float = 0.0,
    external_flow: ScalarSource = 0.0, imposed_pressure: ScalarSource | None = None) -> None
```

Hydraulic pressure node with storage, flow injection, or prescribed pressure.

`HydraulicSystemsConfiguration` class

```
HydraulicSystemsConfiguration(provider: str = 'systems.hydraulic_systems', mesh: str =
    'core.lumped.hydraulic_network', discretization: str = 'network_ode', time_integration:
    str = 'classical_rk4', solver: str = 'nodal_continuity', parameters: Mapping[str, object]
    = <factory>) -> None
```

Select Core time contracts and declarative hydraulic-network data.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the common Core numerical configuration.

`HydraulicSystemsProvider` class

```
HydraulicSystemsProvider(*, nodes: Iterable[HydraulicNode], branches:
    Iterable[HydraulicBranch] = (), final_time: float = 0.0, time_step: float = 0.001,
    time_integration: str = 'classical_rk4', **_: object) -> None
```

Advance arbitrary compliant-node hydraulic networks using Core integration.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:lumped_parameter') ->
    CallableSpatialOperator
```

Expose the hydraulic network through the provider-neutral Study.

```
execute(problem: object | None = None) -> HydraulicSystemsResult
```

Integrate nodal pressure balances and record branch flows.

`HydraulicSystemsPublicApi` class

```
HydraulicSystemsPublicApi(*, provider_registry=None) -> None
```

Validate and execute hydraulic networks through the systems registry.

Selected methods.

```
validate(configuration: HydraulicSystemsConfiguration) -> tuple[str, ...]
```

```
run(configuration: HydraulicSystemsConfiguration)
```

Validate and execute the selected hydraulic provider.

`HydraulicSystemsResult` class

```
HydraulicSystemsResult(state: HydraulicSystemsState, times: np.ndarray, pressures:
    Mapping[str, np.ndarray], flows: Mapping[str, np.ndarray], continuity_residual: float) ->
    None
```

Pressure/flow histories and nodal continuity residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return serializable copies of all node and branch histories.

`HydraulicSystemsState` class

```
HydraulicSystemsState(pressures: np.ndarray) -> None
```

Dynamic pressure vector ordered by non-boundary network nodes.

`PneumaticBranch` class

```
PneumaticBranch(branch_id: str, from_chamber: str, to_chamber: str, conductance: float = 1.0,
    pressure_offset: ScalarSource = 0.0, parameters: Mapping[str, object] = <factory>,
    mass_flow_law: Callable[[float, float, float, Mapping[str, object]], float] | None = None)
    -> None
```

Directed gas connection with extensible mass-flow law.

`PneumaticChamber` class

```
PneumaticChamber(chamber_id: str, volume: float = 1.0, initial_mass: float = 1.0,
    initial_temperature: float = 300.0, external_mass_flow: ScalarSource = 0.0,
    external_temperature: ScalarSource = 300.0, heat_transfer_coefficient: float = 0.0,
    ambient_temperature: ScalarSource = 300.0, imposed_pressure: ScalarSource | None = None,
    imposed_temperature: ScalarSource | None = None) -> None
```

Gas volume with ideal-gas mass and internal-energy storage.

`PneumaticSystemsConfiguration` class

```
PneumaticSystemsConfiguration(provider: str = 'systems.pneumatic_systems', mesh: str =
    'core.lumped.pneumatic_network', discretization: str = 'ideal_gas_network_ode',
    time_integration: str = 'classical_rk4', solver: str = 'mass_energy_balance', parameters:
    Mapping[str, object] = <factory>) -> None
```

Select Core integration and declarative pneumatic-network parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the shared Core numerical configuration.

`PneumaticSystemsProvider` class

```
PneumaticSystemsProvider(*, chambers: Iterable[PneumaticChamber], branches:
    Iterable[PneumaticBranch] = (), gas_constant: float = 287.0, heat_capacity_cv: float =
    718.0, heat_capacity_cp: float = 1005.0, final_time: float = 0.0, time_step: float =
    0.001, time_integration: str = 'classical_rk4', **_: object) -> None
```

Integrate extensible ideal-gas chambers and branches with Core time stepping.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:lumped_parameter') ->
    CallableSpatialOperator
```

Expose the pneumatic network through the common Study runner.

```
execute(problem: object | None = None) -> PneumaticSystemsResult
```

Advance ideal-gas mass and energy balances and record all histories.

`PneumaticSystemsPublicApi` class

```
PneumaticSystemsPublicApi(*, provider_registry=None) -> None
```

Validate and execute pneumatic systems through the systems registry.

Selected methods.

```
validate(configuration: PneumaticSystemsConfiguration) -> tuple[str, ...]
```

```
run(configuration: PneumaticSystemsConfiguration)
```

Validate and execute the selected pneumatic provider.

`PneumaticSystemsResult` class

```
PneumaticSystemsResult(state: PneumaticSystemsState, times: np.ndarray, masses: Mapping[str,
    np.ndarray], temperatures: Mapping[str, np.ndarray], pressures: Mapping[str, np.ndarray],
    mass_flows: Mapping[str, np.ndarray], conservation_residual: float) -> None
```

Gas chamber histories, branch mass flows and conservation residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return serializable copies of pneumatic state and branch histories.

`PneumaticSystemsState` class

```
PneumaticSystemsState(values: np.ndarray) -> None
```

Packed dynamic chamber mass and temperature state.

`ProcessModelsConfiguration` class

```
ProcessModelsConfiguration(provider: str = 'systems.process_models', mesh: str =
    'core.lumped.process_network', discretization: str = 'cstr_balances', time_integration:
    str = 'classical_rk4', solver: str = 'species_energy_balance', parameters: Mapping[str,
    object] = <factory>) -> None
```

Select Core integration and declarative process flowsheet data.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the shared Core numerical configuration.

`ProcessModelsProvider` class

```
ProcessModelsProvider(*, component_ids: Iterable[str], units: Iterable[ProcessUnit], streams:
    Iterable[ProcessStream] = (), final_time: float = 0.0, time_step: float = 0.001,
    time_integration: str = 'classical_rk4', **_: object) -> None
```

Advance arbitrary CSTR-style process flowsheets with the Core integrator.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:lumped_parameter') ->
    CallableSpatialOperator
```

Expose CSTR balances through the provider-neutral Study runner.

```
execute(problem: object | None = None) -> ProcessModelsResult
```

Integrate mass/energy balances and record unit and stream histories.

`ProcessModelsPublicApi` class

```
ProcessModelsPublicApi(*, provider_registry=None) -> None
```

Validate and execute process models via the systems registry.

Selected methods.

```
validate(c: ProcessModelsConfiguration) -> tuple[str, ...]
```

```
run(c: ProcessModelsConfiguration)
```

Validate and execute the selected process provider.

`ProcessModelsResult` class

```
ProcessModelsResult(state: ProcessModelsState, times: np.ndarray, concentrations: Mapping[str,
    Mapping[str, np.ndarray]], temperatures: Mapping[str, np.ndarray], flows: Mapping[str,
    np.ndarray], balance_residual: float) -> None
```

Component, temperature and stream histories with balance residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied process fields and diagnostics.

`ProcessModelsState` class

```
ProcessModelsState(values: np.ndarray) -> None
```

Packed concentration and temperature state of all dynamic process units.

`ProcessStream` class

```
ProcessStream(stream_id: str, source_unit: str | None, target_unit: str | None,
    volumetric_flow: ScalarSource = 0.0, feed_concentrations: Mapping[str, ScalarSource] =
    <factory>, feed_temperature: ScalarSource = 300.0) -> None
```

Directed volumetric stream; external feeds use `source_unit=None`.

`ProcessUnit` class

```
ProcessUnit(unit_id: str, volume: float = 1.0, initial_concentrations: Mapping[str, float] =
    <factory>, initial_temperature: float = 300.0, density: float = 1.0, heat_capacity: float
    = 1.0, heat_source: ScalarSource = 0.0, reaction_law: ReactionLaw | None = None) -> None
```

Perfectly mixed process volume with arbitrary component state.

`ResistorCapacitor` class

```
ResistorCapacitor(*, domain: object, model_id: str = <factory>, resistance: float,
    capacitance: float, supply_voltage: float | None) -> None
```

Advance capacitor voltage under a supplied voltage through a resistor.

Declared fields. `voltage` (V, differential), `supply` (V, input), `current` (A, derived).

`SealedGasChamber` class

```
SealedGasChamber(*, domain: object, model_id: str = <factory>, volume: float, gas_constant:
    float = 287.0, heat_capacity_cv: float = 718.0, thermal_conductance: float = 1.0,
    ambient_temperature: float | None = 300.0) -> None
```

Advance the thermal balance of a sealed gas volume and derive its pressure.

Declared fields. `mass` (kg, differential), `temperature` (K, differential), `pressure` (Pa, derived), `ambient` (K, input).

`SignalBlock` class

```
SignalBlock(block_id: str, kind: str, inputs: tuple[str, ...] = (), parameters: Mapping[str,
    object] = <factory>, source: SignalSource | None = None, custom_evaluator:
    Callable[[tuple[float, ...], float, Mapping[str, object]], float] | None = None,
    custom_derivative: Callable[[tuple[float, ...], float, Mapping[str, object], tuple[float,
    ...]], tuple[float, ...]] | None = None) -> None
```

One declarative signal-processing or continuous-control block.

Selected methods.

```
initial_state() -> tuple[float, ...]
```

Return deterministic initial state from declarative parameters.

`SystemComponentKind` enumeration

```
SystemComponentKind(*values)
```

Select the component role in a systems specification.

`SystemComponentSpec` class

```
SystemComponentSpec(component_id: str, name: str, kind: SystemComponentKind, domain_ids:
    tuple[str, ...], parameters: tuple[tuple[str, float], ...]) -> None
```

Declare the component ID, kind, support and numeric parameters.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'SystemComponentSpec'"
```

```
to_mapping() -> dict[str, object]
```

`SystemConnectionKind` enumeration

```
SystemConnectionKind(*values)
```

Select the connection role in a systems specification.

`SystemConnectionSpec` class

```
SystemConnectionSpec(connection_id: str, name: str, kind: SystemConnectionKind,
    from_component_id: str, to_component_id: str, parameters: tuple[tuple[str, float], ...])
    -> None
```

Declare directed component endpoints and connection parameters.

Selected methods.

```
from_mapping(raw: Mapping[str, object]) -> "'SystemConnectionSpec'"
```

```
to_mapping() -> dict[str, object]
```

`SystemModelKind` enumeration

```
SystemModelKind(*values)
```

Select the network model family in a systems specification.

`SystemsProviderRegistry` class

```
SystemsProviderRegistry() -> None
```

Register systems providers by stable public identifier.

Selected methods.

```
register(provider_id: str, factory: Callable[..., object], *, catalog_role: str =
    'unclassified') -> None
```

```
catalog_role(provider_id: str) -> str
```

```
create(provider_id: str, **parameters: object) -> object
```

```
ids() -> tuple[str, ...]
```

`ThermalCapacitance` class

```
ThermalCapacitance(*, domain: object, model_id: str = <factory>, heat_capacity: float,
    conductance: float, ambient_temperature: float | None, heating: float | None = 0.0) -> None
```

Advance one temperature with thermal storage, ambient exchange and heating.

Declared fields. `temperature` (K, differential), `ambient` (K, input), `power` (W, input).

`ThermalLink` class

```
ThermalLink(link_id: str, from_node: str, to_node: str, conductance: float = 1.0, parameters:
    Mapping[str, object] = <factory>, heat_flow_law: Callable[[float, float, float,
    Mapping[str, object]], float] | None = None) -> None
```

Directed-neutral heat link with default `Q=G(T_from-T_to)` law.

`ThermalNetworksConfiguration` class

```
ThermalNetworksConfiguration(provider: str = 'systems.thermal_networks', mesh: str =
    'core.lumped.thermal_network', discretization: str = 'thermal_rc_network',
    time_integration: str = 'classical_rk4', solver: str = 'nodal_energy_balance', parameters:
    Mapping[str, object] = <factory>) -> None
```

Select Core integration and declarative thermal network data.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate to the shared Core numerical configuration.

`ThermalNetworksProvider` class

```
ThermalNetworksProvider(*, nodes: Iterable[ThermalNode], links: Iterable[ThermalLink] = (),
    final_time: float = 0.0, time_step: float = 0.001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Advance extensible thermal networks with the shared Core time integrator.

Selected methods.

```
as_spatial_operator(*, method_id: str = 'spatial-method:ode.thermal_network') ->
    CallableSpatialOperator
```

Expose the lumped thermal network through the common Study.

```
execute(problem: object | None = None) -> ThermalNetworksResult
```

Integrate node energy balances and record link heat flows.

`ThermalNetworksPublicApi` class

```
ThermalNetworksPublicApi(*, provider_registry=None) -> None
```

Validate and execute thermal networks through the systems registry.

Selected methods.

```
validate(c: ThermalNetworksConfiguration) -> tuple[str, ...]
```

```
run(c: ThermalNetworksConfiguration)
```

Validate and execute selected thermal-network provider.

`ThermalNetworksResult` class

```
ThermalNetworksResult(state: ThermalNetworksState, times: np.ndarray, temperatures:
    Mapping[str, np.ndarray], heat_flows: Mapping[str, np.ndarray], energy_balance_residual:
    float) -> None
```

Temperature/heat-flow histories and nodal energy-balance residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied temperature, heat-flow, and residual data.

`ThermalNetworksState` class

```
ThermalNetworksState(temperatures: np.ndarray) -> None
```

Dynamic temperatures ordered by non-boundary thermal nodes.

`ThermalNode` class

```
ThermalNode(node_id: str, thermal_capacity: float = 1.0, initial_temperature: float = 300.0,
    heat_source: ScalarSource = 0.0, convection_conductance: float = 0.0, ambient_temperature:
    ScalarSource = 300.0, imposed_temperature: ScalarSource | None = None) -> None
```

Thermal storage node with `C dT/dt=Q_ext+Q_links+h(T_inf-T)`.

`WellMixedTank` class

```
WellMixedTank(*, domain: object, model_id: str = <factory>, volume: float, flow_rate: float,
    feed_concentration: float | None, feed_temperature: float | None = 300.0, density: float =
    1000.0, heat_capacity: float = 4180.0) -> None
```

Advance a process tank's homogeneous concentration and temperature with an incoming feed.

Declared fields. `concentration` (mol/m^3, differential), `temperature` (K, differential), `feed` (mol/m^3, input), `feed_thermal` (K, input).

`ZeroDOneDModelsConfiguration` class

```
ZeroDOneDModelsConfiguration(provider: str = 'systems.zero_d_one_d_models', mesh: str =
    'core.coordinate_projection.1d_or_lumped.0d', discretization: str =
    'state_or_finite_difference', time_integration: str = 'classical_rk4', solver: str =
    'explicit_state_evolution', parameters: Mapping[str, object] = <factory>) -> None
```

Select shared Core integration and reduced-model parameters.

Selected methods.

```
to_numerical_configuration() -> NumericalConfiguration
```

Translate this API choice to the common numerical configuration.

`ZeroDOneDModelsProvider` class

```
ZeroDOneDModelsProvider(*, dimension: int, initial_state: float | np.ndarray, cell_count: int
    | None = None, diffusivity: float = 0.0, advection_velocity: float = 0.0, rhs: StateRhs |
    None = None, final_time: float = 0.0, time_step: float = 0.001, time_integration: str =
    'classical_rk4', **_: object) -> None
```

Advance declarative ODE or advection–diffusion–source models.

Selected methods.

```
as_spatial_operator(*, method_id: str | None = None) -> CallableSpatialOperator
```

Expose the reduced ODE/1D transport model through common Study.

```
execute(problem: object | None = None) -> ZeroDOneDResult
```

Advance state through the selected shared Core time integrator.

`ZeroDOneDModelsPublicApi` class

```
ZeroDOneDModelsPublicApi(*, provider_registry=None) -> None
```

Validate and execute 0D/1D models through the systems registry.

Selected methods.

```
validate(c: ZeroDOneDModelsConfiguration) -> tuple[str, ...]
```

```
run(c: ZeroDOneDModelsConfiguration)
```

Validate and execute the selected reduced-model provider.

`ZeroDOneDResult` class

```
ZeroDOneDResult(state: ZeroDOneDState, times: np.ndarray, history: np.ndarray, coordinates:
    np.ndarray | None, equation_residual: float, dimension: int) -> None
```

State histories, coordinates and discrete equation residual.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Return copied time history, final state and optional Core coordinates.

`ZeroDOneDState` class

```
ZeroDOneDState(values: np.ndarray) -> None
```

Finite model state; scalar 0D states are stored as a length-one vector.

APPENDIX B

Numerical API reference

Use the import package above each entry. Pass method declarations into `Numerics` or into the collaborator named in the entry. Methods such as `bind` and `build` are principally used by providers. Their signatures are included for authors extending the API. Keep an application notebook on the declaration route unless it needs a bound-operator diagnostic.

B.1 onesim.numerics

`coupled_systems` namespace

Numerics.coupling public interface.

`discretization` namespace

numerics.discretization public components.

`fdm` namespace

numerics.fdm public components.

`fem` namespace

Numerics.fem public interface.

`field_transfer` namespace

Transfer public interface.

`fvm` namespace

Numerics.fvm public interface.

`linear` namespace

Numerics.linear public interface.

`nonlinear` namespace

Numerics.nonlinear public interface.

`NumericalExecutionPlan` class

```
NumericalExecutionPlan(plan_id: str, problem_id: str, bindings: tuple[ProviderBinding, ...],
    stage_requirement_ids: tuple[str, ...], ir_content_sha256: str = '', study_kind: str =
    'stationary', equation_ids: tuple[str, ...] = (), term_ids: tuple[str, ...] = (),
    field_ids: tuple[str, ...] = (), spatial_method_ids: tuple[str, ...] = (), time_method_id:
    str | None = None, coupling_ids: tuple[str, ...] = (), solver_id: str | None = None,
    backend_id: str | None = None, term_method_bindings: tuple[tuple[str, str], ...] = (),
    splitting_id: str | None = None, preconditioner_id: str | None = None, resource_policy_id:
    str | None = None, requires_spatial_discretization: bool = True, coupling_mode: str =
    'fully_coupled', coupling_blocks: tuple[tuple[str, ...], ...] = (), coupling_strategy_id:
    str | None = None, nonlinear_solver_id: str | None = None, schema_version: str | None =
    None, component_configurations: tuple[tuple[str, str], ...] = ()) -> None
```

Inspect the resolved provider and method bindings, study kind, stage requirements, and execution identity.

Methods and properties.

```
provider_for(requirement_id: str) -> ProviderBinding
```

Return the binding selected for one IR requirement.

```
to_dict() -> dict[str, object]
```

Return the structured dictionary representation.

```
from_dict(payload: Mapping[str, object]) -> NumericalExecutionPlan
```

Reconstruct this declaration from the supplied dictionary.

`content_sha256`. Read the content fingerprint of the resolved declaration or plan.

`NumericalIR` class

```
NumericalIR(problem_id: str, equation_ids: tuple[str, ...], field_ids: tuple[str, ...],
    spatial_method_ids: tuple[str, ...], requirements: tuple[NumericalRequirement, ...],
    stage_requirement_ids: tuple[str, ...] = (), term_ids: tuple[str, ...] = (),
    time_method_id: str | None = None, coupling_ids: tuple[str, ...] = (), solver_id: str |
    None = None, backend_id: str | None = None, study_kind: str = 'stationary',
    schema_version: str = '1.0', term_method_bindings: tuple[tuple[str, str], ...] = (),
    splitting_id: str | None = None, preconditioner_id: str | None = None, resource_policy_id:
    str | None = None, requires_spatial_discretization: bool = True, coupling_mode: str =
    'fully_coupled', coupling_blocks: tuple[tuple[str, ...], ...] = (), coupling_strategy_id:
    str | None = None, nonlinear_solver_id: str | None = None, semantic_field_bindings:
    tuple[tuple[str, str, str], ...] = (), component_configurations: tuple[tuple[str, str],
    ...] = ()) -> None
```

Inspect the equation, field, term and method requirements supplied to numerical resolution.

Methods and properties.

```
to_dict() -> dict[str, object]
```

Return the complete provider-neutral, canonical serialization.

```
from_dict(payload: Mapping[str, object]) -> "'NumericalIR'"
```

Reconstruct this declaration from the supplied dictionary.

`content_sha256`. Read the content fingerprint of the resolved declaration or plan.

`Numerics` class

```
Numerics(*, spatial: object, linear_solver: object | None = None, nonlinear_solver: object |
    None = None, linearized_solver: object | None = None, time_integrator: object | None =
    None, step_limits: TimeStepLimits = <factory>, coupling: object | None = None,
    initial_guess: Mapping[FieldRef, object] | None = None, field_scales: Mapping[FieldRef,
    object] | None = None) -> None
```

Compose independent spatial, linear, nonlinear, time, and coupling selections. Initial guesses and scales are mappings keyed by owned field references.

`time` namespace

Numerics.time public interface.

B.2 onesim.numerics.coupled_systems

`AlgebraicClosure` class

```
AlgebraicClosure(*, nonlinear_solver: object, linear_solver: object | None = None,
    initial_values: Mapping = <factory>) -> None
```

Resolve instantaneous field feedback with a bound nonlinear solver. Use source units for initial guesses.

Methods and properties.

```
bind(*, dependencies, context)
```

Bind this declaration to the supplied execution context and collaborators.

`CoupledVelocityPressure` class

```
CoupledVelocityPressure(*, momentum_solver: object, pressure_solver: object, nonlinear_solver:
    object | None = None, factory: object | None = None) -> None
```

Solve velocity, face flux and pressure together with the selected inner solvers.

Methods and properties.

```
bind(*, operator, momentum_solver, pressure_solver, nonlinear_solver)
```

Bind this declaration to the supplied execution context and collaborators.

`MonolithicCoupling` class

```
MonolithicCoupling(*, transient_systems: tuple, algebraic_closure: object | None = None,
    time_integrator: object, nonlinear_solver: object | None = None, linear_solver: object |
    None = None, linearized_solver: object | None = None, stability: object | None = None) ->
    None
```

Advance unique transient blocks with one time algorithm, explicit stage solvers where needed, and shared stability.

Methods and properties.

`order`. Read the nominal selected order.

`PartitionedCoupling` class

```
PartitionedCoupling(*, transient_systems: tuple, algebraic_closure: object | None = None,
    iteration: object, stationary_systems: tuple = (), order: tuple | None = None, on_failure:
    str = 'reject_time_step') -> None
```

Replay one trial step per transient partition while iterating exchanged fields. Nonconvergence rejects the step.

`PISO` class

```
PISO(*, momentum_solver: object, pressure_solver: object, pressure_correctors: int = 2,
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-10, maximum_iterations:
    int = 100, pressure_relaxation: float = 0.5, momentum_relaxation: float = 0.7, factory:
    object | None = None) -> None
```

Configure a supported FV pressure-correction solve with momentum and pressure solvers, correctors and relaxation.

Methods and properties.

```
bind(*, operator, momentum_solver, pressure_solver)
```

Bind this declaration to the supplied execution context and collaborators.

`QuasiStationaryCoupling` class

```
QuasiStationaryCoupling(*, transient_systems: tuple, algebraic_closure: object | None = None,
    stationary_systems: tuple, iteration: object, on_failure: str = 'reject_time_step') -> None
```

Iterate nonoverlapping transient and stationary blocks at a shared trial time.

`SequentialCoupling` class

```
SequentialCoupling(*, transient_systems: tuple, algebraic_closure: object | None = None,
    stationary_systems: tuple, order: tuple) -> None
```

Give an explicit sweep order containing each declared transient or stationary model exactly once.

B.3 onesim.numerics.discretization

`CapabilitySet` class

```
CapabilitySet(features: frozenset[str], guarantees: frozenset[str] = frozenset()) -> None
```

Describe the feature and guarantee identifiers offered by a provider.

`CellAverageProjection` class

```
CellAverageProjection(*, quadrature_order: int | None = None) -> None
```

Initialize FV cell means by physical volume quadrature. Set quadrature_order when the model needs a different rule.

Methods and properties.

```
project(*, initial, space, time, linear_solver=None)
```

Project initial data onto the requested space.

`count` function

```
count(value, name, *, minimum=1, maximum=None)
```

Return a validated built-in integer within inclusive bounds. The name identifies the option in a ValueError.

`evaluate_method_provider` function

```
evaluate_method_provider(prototype_id: str, additional: RequirementSet | None = None, *,
    provider=None, context=None) -> CompatibilityReport
```

Compare the supplied provider's offer against method requirements and any additional requirements. Returns a compatibility report.

`EvaluationCacheControl` protocol

```
EvaluationCacheControl(*args, **kwargs)
```

Structural port for an operator that exposes evaluation cache controls.

Methods and properties.

```
invalidate_evaluation_cache() -> None
```

Invalidate owned evaluation state after an explicit input update.

`L2Projection` class

```
L2Projection(*, quadrature_order: int | None = None) -> None
```

Project initial values by the supported mass solve with a selectable quadrature order.

Methods and properties.

```
project(*, initial, space, time, linear_solver=None)
```

Project initial data onto the requested space.

`LumpedParameters` class

```
LumpedParameters(*, provider: object | None = None, stability: object | None = None,
    algebraic_solver: object | None = None) -> None
```

Choose the zero-dimensional spatial contract for a LumpedMesh. Optional collaborators supply stability or algebraic operations.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`MethodProviderOffer` class

```
MethodProviderOffer(provider_id: str, capabilities: CapabilitySet, available: bool | None =
    None, diagnostics: tuple[str, ...] = (), rejection_kind: str = 'unconfirmed') -> None
```

Pass a provider's actual capability offer to compatibility evaluation. Availability and diagnostics describe the supplied context.

`NodalInterpolation` class

```
NodalInterpolation() -> None
```

Evaluate initial values at native nodes through the spatial initialization port.

Methods and properties.

```
project(*, initial, space, time, linear_solver=None)
```

Project initial data onto the requested space.

`NumericalCompatibilityError` class

```
NumericalCompatibilityError(message, *, kind, option, required_capabilities=())
```

Exception with a machine-readable rejection kind, rejected option and required capabilities.

`NumericalExecutionPlan` class

```
NumericalExecutionPlan(plan_id: str, problem_id: str, bindings: tuple[ProviderBinding, ...],
    stage_requirement_ids: tuple[str, ...], ir_content_sha256: str = '', study_kind: str =
    'stationary', equation_ids: tuple[str, ...] = (), term_ids: tuple[str, ...] = (),
    field_ids: tuple[str, ...] = (), spatial_method_ids: tuple[str, ...] = (), time_method_id:
    str | None = None, coupling_ids: tuple[str, ...] = (), solver_id: str | None = None,
    backend_id: str | None = None, term_method_bindings: tuple[tuple[str, str], ...] = (),
    splitting_id: str | None = None, preconditioner_id: str | None = None, resource_policy_id:
    str | None = None, requires_spatial_discretization: bool = True, coupling_mode: str =
    'fully_coupled', coupling_blocks: tuple[tuple[str, ...], ...] = (), coupling_strategy_id:
    str | None = None, nonlinear_solver_id: str | None = None, schema_version: str | None =
    None, component_configurations: tuple[tuple[str, str], ...] = ()) -> None
```

Inspect the resolved provider and method bindings, study kind, stage requirements, and execution identity.

Methods and properties.

```
provider_for(requirement_id: str) -> ProviderBinding
```

Return the binding selected for one IR requirement.

```
to_dict() -> dict[str, object]
```

Return the structured dictionary representation.

```
from_dict(payload: Mapping[str, object]) -> NumericalExecutionPlan
```

Reconstruct this declaration from the supplied dictionary.

`content_sha256`. Read the content fingerprint of the resolved declaration or plan.

`NumericalIR` class

```
NumericalIR(problem_id: str, equation_ids: tuple[str, ...], field_ids: tuple[str, ...],
    spatial_method_ids: tuple[str, ...], requirements: tuple[NumericalRequirement, ...],
    stage_requirement_ids: tuple[str, ...] = (), term_ids: tuple[str, ...] = (),
    time_method_id: str | None = None, coupling_ids: tuple[str, ...] = (), solver_id: str |
    None = None, backend_id: str | None = None, study_kind: str = 'stationary',
    schema_version: str = '1.0', term_method_bindings: tuple[tuple[str, str], ...] = (),
    splitting_id: str | None = None, preconditioner_id: str | None = None, resource_policy_id:
    str | None = None, requires_spatial_discretization: bool = True, coupling_mode: str =
    'fully_coupled', coupling_blocks: tuple[tuple[str, ...], ...] = (), coupling_strategy_id:
    str | None = None, nonlinear_solver_id: str | None = None, semantic_field_bindings:
    tuple[tuple[str, str, str], ...] = (), component_configurations: tuple[tuple[str, str],
    ...] = ()) -> None
```

Inspect the equation, field, term and method requirements supplied to numerical resolution.

Methods and properties.

```
to_dict() -> dict[str, object]
```

Return the complete provider-neutral, canonical serialization.

```
from_dict(payload: Mapping[str, object]) -> "'NumericalIR'"
```

Reconstruct this declaration from the supplied dictionary.

`content_sha256`. Read the content fingerprint of the resolved declaration or plan.

`Numerics` class

```
Numerics(*, spatial: object, linear_solver: object | None = None, nonlinear_solver: object |
    None = None, linearized_solver: object | None = None, time_integrator: object | None =
    None, step_limits: TimeStepLimits = <factory>, coupling: object | None = None,
    initial_guess: Mapping[FieldRef, object] | None = None, field_scales: Mapping[FieldRef,
    object] | None = None) -> None
```

Compose independent spatial, linear, nonlinear, time, and coupling selections. Initial guesses and scales are mappings keyed by owned field references.

`OperatorConditions` protocol

```
OperatorConditions(*args, **kwargs)
```

Structural port for a bound spatial operator's condition evaluation.

Methods and properties.

```
conditions(time: float) -> tuple
```

Return (essential constraints, boundary contributions, boundary loads).

`OperatorFacetTopology` protocol

```
OperatorFacetTopology(*args, **kwargs)
```

Structural port for an operator that exposes boundary or facet topology.

Methods and properties.

```
facet_owners(facet) -> tuple[int, ...]
```

Return the incident cell IDs used to orient the supplied facet.

`RequirementSet` class

```
RequirementSet(required: frozenset[str], preferred: frozenset[str] = frozenset()) -> None
```

Specify required and preferred feature identifiers for compatibility checks.

B.4 onesim.numerics.fdm

`BoundaryFluxDifference` class

```
BoundaryFluxDifference(*, derivative: object, boundary_flux: object = None, factory: object =
    None) -> None
```

Wrap a first derivative and its optional boundary flux to use physical exterior traces in conservative transport.

Methods and properties.

```
validate_geometry(geometry)
```

Check whether the supplied mesh geometry is compatible with the method.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, coordinates, periodic, law, boundaries, geometry=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_conservation(*, coordinates, periodic, law, boundaries, geometry=None)
```

Bind the conservative operator using the physical flux and geometry supplied by the model.

`CentralDifference` class

```
CentralDifference(kernel: str = 'numpy', factory: object | None = None) -> None
```

Use as gradient for a second-order centered first derivative on supported uniform samples.

Methods and properties.

```
validate_geometry(geometry)
```

Check whether the supplied mesh geometry is compatible with the method.

```
bind(*, coordinates, periodic)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`CentralLaplacian` class

```
CentralLaplacian(kernel: str = 'numpy', factory: object | None = None) -> None
```

Use as diffusion for a second-order scalar Laplacian on a uniform sample axis.

Methods and properties.

```
bind(*, coordinates, periodic)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`FiniteDifference` class

```
FiniteDifference(*, gradient: object | None = None, diffusion: object | None = None,
    reaction_sources: object | None = None, curl: object | None = None, cell_quadrature_order:
    int | None = None, face_quadrature_order: int | None = None, boundary_enforcement: object
    | None = None, stabilization: object | None = None, provider: object | None = None,
    stability: object | None = None) -> None
```

Select the named stencil, flux, boundary, curl or source collaborators required by the model's FD binding.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`GlobalLaxFriedrichsSplit` class

```
GlobalLaxFriedrichsSplit(factory: object = None) -> None
```

Supply a flux splitting based on a global wave-speed bound to WENOFluxDifference.

Methods and properties.

```
bind(*, law)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`HydrostaticFluxDifference` class

```
HydrostaticFluxDifference(*, flux: object, invariant_domain: object, dry_linearization: object
    = None) -> None
```

Select shallow-water hydrostatic flux differences with a matching Riemann flux, invariant domain and optional dry linearization.

Methods and properties.

```
validate(*, numerics, **unused)
```

Validate the supplied numerical context or input.

```
bind_operator(*, model, mesh, numerics, boundary_conditions, initial_conditions, inputs=None,
    **unused)
```

Construct the selected operator on the supplied geometry and context.

`PolynomialDifference` class

```
PolynomialDifference(*, accuracy_order: int = 2, factory: object | None = None) -> None
```

Use as gradient for a polynomial stencil at the requested accuracy order on a supported mesh.

Methods and properties.

`stencil_width`. Read the number of stencil samples required by the selected order.

```
validate_geometry(geometry)
```

Check whether the supplied mesh geometry is compatible with the method.

```
bind(*, coordinates, periodic, geometry=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`PolynomialLaplacian` class

```
PolynomialLaplacian(*, accuracy_order: int = 2, factory: object | None = None) -> None
```

Use as diffusion for a polynomial second-derivative stencil on supported coordinates.

Methods and properties.

```
bind(*, coordinates, periodic, geometry=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`SummationByPartsDifference` class

```
SummationByPartsDifference(*, accuracy_order: int = 2, factory: object = None) -> None
```

Select a first derivative with a summation-by-parts boundary structure for a compatible line binding.

Methods and properties.

```
bind(*, coordinates, periodic, length=None, **unused)
```

Bind this declaration to the supplied execution context and collaborators.

`WENO5Flux` class

```
WENO5Flux(factory: object = None) -> None
```

Supply fifth-order WENO flux reconstruction for the supported split-flux stencil.

Methods and properties.

```
bind()
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`WENOFluxDifference` class

```
WENOFluxDifference(flux_splitting: object, reconstruction: object, factory: object = None,
    diagnostic_gradient: object = None) -> None
```

Combine a flux splitter and a reconstruction in the conservative gradient port. A diagnostic gradient may supply a separate derivative for output.

Methods and properties.

```
bind_conservation(*, coordinates, periodic, law, boundaries, geometry=None)
```

Bind the conservative operator using the physical flux and geometry supplied by the model.

```
validate_geometry(geometry)
```

Check whether the supplied mesh geometry is compatible with the method.

```
bind(*, coordinates, periodic, law, boundaries)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

B.5 onesim.numerics.fem

`BrezziDouglasMarini` class

```
BrezziDouglasMarini(order: int = 1) -> None
```

Use as the H(div) flux space with complete vector polynomials of order at least one.

Methods and properties.

```
bind(*, mesh, quadrature_order=None)
```

Bind this declaration to the supplied execution context and collaborators.

`CIP` class

```
CIP(*, penalty: object = 0.1, factory: object | None = None) -> None
```

Select a continuous interior penalty on jumps of physical first derivatives.

Methods and properties.

```
bind_physical(*, operator)
```

Bind the numerical operation to the supplied physical model and geometry.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

`ConsistentMassStability` class

```
ConsistentMassStability(*, dense_threshold: int = 256, relative_tolerance: float = 1e-10) ->
    None
```

Compute a compatible generalized spectral bound from the free stiffness and consistent mass blocks. Configure dense threshold and eigensolver tolerance.

Methods and properties.

```
maximum_step(*, rate, state, time, declaration)
```

Compute the stability-based maximum step for the supplied state and rate.

`ConstraintAugmentation` class

```
ConstraintAugmentation(weight: float = 1.0) -> None
```

Add a weighted constraint contribution to a compatible primal block while preserving the exact constraint.

Methods and properties.

```
bind_mixed(*, operator)
```

Bind a mixed operator on its declared spaces.

`DGJumpPenalty` class

```
DGJumpPenalty(*, coefficient: float) -> None
```

Use a jump-based interior dissipation contribution with an explicit nonnegative coefficient.

Methods and properties.

```
bind(*, space, model, law, periodic)
```

Bind this declaration to the supplied execution context and collaborators.

`DGPolynomialLimiter` class

```
DGPolynomialLimiter(slope_limiter: object = None, invariant_domain: object = None, factory:
    object = None) -> None
```

Apply a slope reduction and optional convex admissibility scaling on a compatible DG stage.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, space, model, law, periodic)
```

Bind this declaration to the supplied execution context and collaborators.

`DiscontinuousLagrange` class

```
DiscontinuousLagrange(order: int = 1) -> None
```

Use as element for cell-local polynomials. Order zero gives a cellwise constant space.

Methods and properties.

```
bind_cell_space(*, mesh, quadrature_order=None)
```

Bind the discrete cell-supported space for the supplied mesh.

```
bind(*, mesh, quadrature_order=None, periodic=False)
```

Bind this declaration to the supplied execution context and collaborators.

`DissipativeMassStability` class

```
DissipativeMassStability(*, dense_threshold: int = 256, relative_tolerance: float = 1e-10) ->
    None
```

Resolve a spectral interval or an M-norm contractivity bound for the supported dissipative operator.

Methods and properties.

```
maximum_step(*, rate, state, time, declaration)
```

Compute the stability-based maximum step for the supplied state and rate.

`EqualOrderVelocityPressure` class

```
EqualOrderVelocityPressure(order: int = 1) -> None
```

Select equal-degree velocity and pressure spaces with a compatible stabilized formulation.

Methods and properties.

```
bind(*, mesh, quadrature_order=None)
```

Bind this declaration to the supplied execution context and collaborators.

`FiniteElement` class

```
FiniteElement(*, element: object, quadrature_order: int | None = None, provider: object | None
    = None, stabilization: object | None = None, convection: object | None = None, diffusion:
    object | None = None, reaction_sources: object | None = None, limiter: object | None =
    None, stability: object | None = None, initialization: object | None = None,
    boundary_enforcement: object | None = None) -> None
```

Choose an element space, integration rule, stabilization, boundary enforcement and compatible term components.

Methods and properties.

```
bind_space(mesh, *, factory=None, periodic=None)
```

Build the selected actual FE space through its explicitly selected owner.

`GalerkinAdvection` class

```
GalerkinAdvection(factory: object | None = None) -> None
```

Select the Galerkin advection contribution in an appropriate finite-element weak form.

Methods and properties.

```
bind(*, space, field_binding)
```

Bind this declaration to the supplied execution context and collaborators.

`GalerkinDiffusion` class

```
GalerkinDiffusion(*, factory: object = None) -> None
```

Select the Galerkin diffusion contribution where the physical residual exposes that port.

Methods and properties.

```
bind_volume(*, space)
```

Unweighted physical Cartesian gradient form on the selected volume space.

```
bind(*, space, coefficients, periodic, boundaries, boundary_enforcement=None)
```

Bind this declaration to the supplied execution context and collaborators.

`GLS` class

```
GLS(*, tau: object, factory: object | None = None, strong_residual: object | None = None) ->
    None
```

Select Galerkin least-squares stabilization with a scale tau and a compatible strong residual.

Methods and properties.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

`GradDiv` class

```
GradDiv(*, penalty: object = 1.0, factory: object = None) -> None
```

Select a penalty on the divergence of the velocity in a compatible mixed momentum assembly.

Methods and properties.

```
bind(*, space, balance, mesh_motion=None)
```

Bind this declaration to the supplied execution context and collaborators.

`HcurlNitsche` class

```
HcurlNitsche(penalty: float = 50.0) -> None
```

Use in boundary_enforcement for weak tangential traces with a dimensionless penalty.

Methods and properties.

```
bind_hcurl(*, space, model, mesh)
```

Bind the H(curl) operator on the supplied carrier.

`HydrostaticGraphViscosity` class

```
HydrostaticGraphViscosity(*, mass: object, invariant_domain: object, maximum_expansions: int =
    32, factory: object = None, dry_linearization: object = None) -> None
```

Combine graph viscosity with hydrostatic equilibrium and a shallow-water invariant domain.

Methods and properties.

```
bind_graph(*, space, model, law, boundaries, flux)
```

Build the graph-based operation on the supplied discrete support.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

`InterpolatedFluxDivergence` class

```
InterpolatedFluxDivergence(*, factory: object | None = None) -> None
```

Use a compatible continuous formulation that differentiates the interpolated physical flux.

Methods and properties.

```
bind(*, space)
```

Bind this declaration to the supplied execution context and collaborators.

`InvariantDomainGraphViscosity` class

```
InvariantDomainGraphViscosity(*, mass: object, invariant_domain: object, maximum_expansions:
    int = 32, factory: object = None) -> None
```

Choose a continuous-graph viscosity treatment using a selected mass operator and invariant domain.

Methods and properties.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_graph(*, space, model, law, boundaries, flux)
```

Build the graph-based operation on the supplied discrete support.

`Lagrange` class

```
Lagrange(order: int = 1) -> None
```

Use as element for a continuous nodal space. Order is total degree on simplices and per-coordinate degree on supported hexahedra.

`MixedContinuityContribution` class

```
MixedContinuityContribution(*, mass: object = None, velocity: object = None, pressure: object
    = None, load: object = None) -> None
```

Provide explicit mass, velocity, pressure or load pieces for the mixed continuity assembly.

`MixedDiffusionElement` class

```
MixedDiffusionElement(*, flux: object, scalar: object) -> None
```

Pair an H(div) flux space with a discontinuous scalar element for a mixed diffusion solve.

Methods and properties.

`order`. Read the nominal selected order.

```
bind(*, mesh, quadrature_order=None)
```

Bind this declaration to the supplied execution context and collaborators.

`MixedMomentumContribution` class

```
MixedMomentumContribution(*, mass: object = None, momentum: object = None, pressure: object =
    None, load: object = None) -> None
```

Provide explicit mass, momentum, pressure or load pieces for the mixed momentum assembly.

`Nedelec` class

```
Nedelec(order: int = 1) -> None
```

Use as element for first-family H(curl) edge elements. The positive public order starts at one.

`Nitsche` class

```
Nitsche(*, penalty: float = 10.0, symmetry: float = 1.0, factory: object | None = None) -> None
```

Enforce compatible diffusion or inflow traces weakly. Penalty and symmetry select the boundary form.

Methods and properties.

```
bind_physical(*, operator)
```

Bind the numerical operation to the supplied physical model and geometry.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

`OrderedFields` class

```
OrderedFields(*, lower: str, upper: str, minimum_separation: float) -> None
```

Constrain two named fields to remain ordered with a specified minimum separation.

Methods and properties.

```
bind(*, model, law)
```

Bind this declaration to the supplied execution context and collaborators.

```
admissible(values)
```

Test the supplied state against the invariant-domain rule.

```
maximum_scaling(mean, samples)
```

Compute the permitted scaling toward an admissible mean.

`PSPG` class

```
PSPG(*, time_scale: object, factory: object = None) -> None
```

Set the stabilization time scale for the supported pressure-stabilized mixed residual.

Methods and properties.

```
bind(*, space, balance, mesh_motion=None)
```

Bind this declaration to the supplied execution context and collaborators.

`RaviartThomas` class

```
RaviartThomas(order: int = 0) -> None
```

Use as the H(div) flux space with polynomial subdegree k. RT0 has order zero.

Methods and properties.

```
bind(*, mesh, quadrature_order=None)
```

Bind this declaration to the supplied execution context and collaborators.

`RowSumLumping` class

```
RowSumLumping(factory: object = None) -> None
```

Supply row-sum mass lumping to a compatible nodal or graph-viscosity method.

Methods and properties.

```
diagonal(*, space, consistent_mass)
```

Evaluate the diagonal entries used by the mass or operator contract.

`ScalarInterval` class

```
ScalarInterval(field: str, lower: float, upper: float) -> None
```

Constrain the named scalar field to an interval by mean-preserving scaling in a supported limiter.

Methods and properties.

```
coordinate_bounds(*, point_count)
```

Return the bounds used by the compatible limiter.

```
bind(*, model, law)
```

Bind this declaration to the supplied execution context and collaborators.

`tolerance`. Read the tolerance used in the bound check.

```
admissible(values)
```

Test the supplied state against the invariant-domain rule.

```
maximum_scaling(mean, samples)
```

Compute the permitted scaling toward an admissible mean.

```
source_step(values, source)
```

Forward-source bound for both faces of the selected scalar interval.

`StateLimiterPort` protocol

```
StateLimiterPort(*args, **kwargs)
```

Structural contract for an executable state limiter on the bound spatial state.

Methods and properties.

```
limit(values, *, time: float, stage_id: object)
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`SUPG` class

```
SUPG(*, tau: object, factory: object | None = None, strong_residual: object | None = None) ->
    None
```

Select streamline-upwind Petrov–Galerkin stabilization with tau and a supported strong residual.

Methods and properties.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

`SymmetricInteriorPenalty` class

```
SymmetricInteriorPenalty(*, penalty: float = 10.0, factory: object | None = None) -> None
```

Select symmetric interior-penalty diffusion for DG with an explicit penalty.

Methods and properties.

```
bind(*, space, coefficients, periodic, boundaries=None)
```

Bind this declaration to the supplied execution context and collaborators.

`TaylorHood` class

```
TaylorHood(order: int = 2) -> None
```

Select a mixed velocity–pressure space with pressure one polynomial degree lower than velocity.

Methods and properties.

```
bind(*, mesh, quadrature_order=None)
```

Bind this declaration to the supplied execution context and collaborators.

`VolumeCIP` class

```
VolumeCIP(*, diffusivity: object, factory: object | None = None) -> None
```

Penalize jumps of the normal derivative with an explicitly supplied diffusivity.

Methods and properties.

```
bind(*, space, balance)
```

Bind this declaration to the supplied execution context and collaborators.

B.6 onesim.numerics.field_transfer

`CellToQuadratureProjection` class

```
CellToQuadratureProjection(*, source_mesh: object | None = None, target_mesh: object | None =
    None, factory: object | None = None) -> None
```

Map native cell data to the receiving quadrature support through the bound evaluator.

Methods and properties.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`ConservativeVolumeTransfer` class

```
ConservativeVolumeTransfer(*, source_mesh: object | None = None, target_mesh: object | None =
    None, factory: object | None = None, absolute_tolerance: float = 1e-10,
    relative_tolerance: float = 1e-07, quadrature_relative_tolerance: float = 1e-09,
    quadrature_absolute_tolerance: float = 0.0, quadrature_order: int = 4,
    maximum_subdivisions: int = 256, maximum_depth: int = 10, positive: bool = True,
    intersection_roots: object | None = None, intersection_event_tolerance: float = 1e-08) ->
    None
```

Integrate an FE scalar density over physical FV overlaps. Configure coverage and quadrature tolerances for the receiving cells.

Methods and properties.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_runtime(*, evaluator, target, target_mesh, target_space, identifier, factory=None)
```

Bind the transfer for a concrete runtime source and receiver.

```
receiver_evaluator(*, evaluator, target_mesh, factory=None)
```

Provide the bound evaluator used by the target field.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`ElectricFieldToFaces` class

```
ElectricFieldToFaces(*, source_mesh: object | None = None, target_mesh: object | None = None,
    factory: object | None = None, periodic_trace: object | None = None) -> None
```

Provide the electric field on target faces, with an explicit periodic potential trace when needed.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_runtime(*, evaluator, target, target_mesh, target_space, identifier, factory=None)
```

Bind the transfer for a concrete runtime source and receiver.

```
receiver_evaluator(*, evaluator, target_mesh, factory=None)
```

Provide the bound evaluator used by the target field.

`FieldInterpolation` class

```
FieldInterpolation(*, source_mesh: object | None = None, target_mesh: object | None = None,
    factory: object | None = None) -> None
```

Interpolate a field to a compatible receiving support using the selected transfer factory.

Methods and properties.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`FiniteElementEvaluation` class

```
FiniteElementEvaluation(*, source_mesh: object | None = None, target_mesh: object | None =
    None, factory: object | None = None) -> None
```

Evaluate an FE field on the receiver's requested physical points, such as mechanical quadrature.

Methods and properties.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind_runtime(*, evaluator, target, target_mesh, target_space, identifier, factory=None)
```

Bind the transfer for a concrete runtime source and receiver.

`GlobalBroadcastTransfer` class

```
GlobalBroadcastTransfer(*, source_mesh: object | None = None, target_mesh: object | None =
    None, factory: object | None = None) -> None
```

Send an owned global scalar or vector to a compatible spatial input as a uniform value.

Methods and properties.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`GlobalValueTransfer` class

```
GlobalValueTransfer(*, source_mesh: object | None = None, target_mesh: object | None = None,
    factory: object | None = None) -> None
```

Send a global value to another compatible global input without a spatial sampling step.

Methods and properties.

```
receiver_evaluator(*, evaluator, target_mesh, factory=None)
```

Provide the bound evaluator used by the target field.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_runtime(*, evaluator, target, target_mesh, target_space, identifier, factory=None)
```

Bind the transfer for a concrete runtime source and receiver.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`operators` namespace

Transfer / operators.

`PeriodicPotentialTrace` class

```
PeriodicPotentialTrace(*, potential_jump: object = 0.0) -> None
```

Set the signed potential increase from the first periodic face to the second.

Methods and properties.

```
bind(*, evaluator, mesh)
```

Bind this declaration to the supplied execution context and collaborators.

`SpatialAverageTransfer` class

```
SpatialAverageTransfer(*, source_mesh: object | None = None, target_mesh: object | None =
    None, factory: object | None = None, quadrature_order: int = 4, quadrature: object | None
    = None) -> None
```

Integrate a native spatial field and divide by its physical measure using the selected quadrature.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, source, target, target_mesh, factory=None)
```

Bind this declaration to the supplied execution context and collaborators.

B.7 onesim.numerics.fvm

`BarthJespersenLimiter` class

```
BarthJespersenLimiter(epsilon: float = 1e-14) -> None
```

Limit reconstructed values to the local neighbor range on a supported multidimensional stencil.

Methods and properties.

```
factor(center: float, neighbors: tuple[float, ...], increment: float) -> float
```

limit slope.

```
limit_slope(center: float, left: float, right: float, slope: float) -> float
```

Apply the limiter to the given one-sided slopes.

```
factor_jvp(center, neighbors, increment, center_direction, neighbors_direction,
    increment_direction)
```

Derivative of the executed extremum/ratio branch at a cell face.

```
cell_factor(center, neighbors, increment)
```

Compute the reconstruction limiter factor for the given cell data.

```
cell_factor_jvp(center, neighbors, increment, center_direction, neighbors_direction,
    increment_direction)
```

Evaluate the directional derivative of the cell limiter factor.

```
limit(left_slope: float, right_slope: float) -> float
```

Limit both cell faces against the actual signed neighbor changes.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Evaluate the directional derivative of the limited result.

`BoundaryLeastSquaresGradient` class

```
BoundaryLeastSquaresGradient(*, factory: object = None) -> None
```

Reconstruct cell gradients with boundary data through the supported least-squares port.

Methods and properties.

```
bind(*, geometry)
```

Bind this declaration to the supplied execution context and collaborators.

`CellGradientReconstruction` class

```
CellGradientReconstruction(*, factory: object | None = None) -> None
```

Declare a native cell-gradient reconstruction component for compatible volume carriers.

Methods and properties.

```
bind_volume(*, geometry, gradient, limiter=None)
```

Bind this declaration to a supported volume mesh.

`CentralDifference` class

```
CentralDifference(factory: object | None = None) -> None
```

Select the central FV diffusion contribution on supported carriers. This is a different declaration from fdm.CentralDifference.

Methods and properties.

```
bind(*, geometry, gradient)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_scalar(*, geometry, gradient)
```

Bind the declaration for a scalar field.

`CentralElasticStress` class

```
CentralElasticStress(*, factory: object = None) -> None
```

Choose the central FV stress contribution for a compatible elasticity formulation.

Methods and properties.

```
bind_elasticity(*, geometry, gradient)
```

Bind the finite-volume elasticity collaborator.

`ConvexInvariantDomain` class

```
ConvexInvariantDomain(*, factory: object | None = None) -> None
```

Retract face states toward their unchanged cell mean to satisfy the bound invariant domain.

Methods and properties.

```
bind(*, admissible)
```

Bind this declaration to the supplied execution context and collaborators.

`DifferentiableSamples` class

```
DifferentiableSamples(values: object, jacobian: object) -> None
```

Carry sampled values and their Jacobian together in a differentiable FV assembly.

Methods and properties.

```
variables(values)
```

Create differentiable samples representing independent variables.

```
constant(value)
```

Create differentiable samples with a zero derivative.

```
unary(values, derivative)
```

Apply a unary operation with its derivative to the samples.

```
apply(matrix)
```

Apply the bound operation with the supplied values and collaborators.

```
sum()
```

Reduce the differentiable samples while propagating the corresponding derivatives.

`Eno2Reconstructor` class

```
Eno2Reconstructor() -> None
```

Select a smooth second-order scalar ENO stencil at the face.

Methods and properties.

```
bind(*, cell_widths, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
reconstruct_left_interfaces(values: tuple[float, ...]) -> tuple[float, ...]
```

Reconstruct the left state at each supported interface from cell samples.

`Eno3Reconstructor` class

```
Eno3Reconstructor() -> None
```

Select a smooth quadratic scalar ENO stencil at the face.

Methods and properties.

```
bind(*, cell_widths, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
reconstruct_left_interfaces(values: tuple[float, ...]) -> tuple[float, ...]
```

Reconstruct the left state at each supported interface from cell samples.

`FiniteIntegration` class

```
FiniteIntegration(*, hodge: object, topology: object = <factory>) -> None
```

Select a discrete incidence complex and metric Hodge maps for a compatible curl operator.

Methods and properties.

```
bind_complex(*, mesh, cell_quadrature_order, face_quadrature_order)
```

Build the discrete incidence complex and metric operators on the supplied mesh.

`FiniteVolume` class

```
FiniteVolume(*, face_quadrature_order: int = 3, cell_quadrature_order: int = 3, convection:
    object | None = None, reconstruction: object | None = None, limiter: object | None = None,
    positivity: object | None = None, stability: object | None = None, diffusion: object |
    None = None, curl: object | None = None, stabilization: object | None = None,
    boundary_enforcement: object | None = None, gradient: object | None = None,
    velocity_pressure_interpolation: object | None = None, particle_flux: object | None =
    None, energy_flux: object | None = None, reaction_sources: object | None = None,
    initialization: object | None = None, provider: object | None = None) -> None
```

Compose control-volume term operators, reconstruction, limiter, initialization, and cell and face quadrature.

Methods and properties.

```
option_applicability(*, context)
```

Dimension-dependent semantics, not a claim of provider execution.

`HydrostaticReconstruction` class

```
HydrostaticReconstruction(*, dry_linearization: object = None) -> None
```

Reconstruct depth and discharge consistently with the bed for a supported well-balanced method.

Methods and properties.

```
validate(*, numerics, **unused)
```

Validate the supplied numerical context or input.

```
bind_operator(*, model, mesh, numerics, boundary_conditions, initial_conditions, inputs=None,
    **unused)
```

Construct the selected operator on the supplied geometry and context.

`ImplicitLocalSources` class

```
ImplicitLocalSources(factory: object | None = None) -> None
```

Select the implicit local source treatment for models exposing a compatible reaction or source operator.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, model, geometry)
```

Bind this declaration to the supplied execution context and collaborators.

`LeastSquaresGradient` class

```
LeastSquaresGradient(factory: object | None = None) -> None
```

Choose least-squares reconstruction of cell gradients for supported face fluxes and field diagnostics.

Methods and properties.

```
bind(*, geometry)
```

Bind this declaration to the supplied execution context and collaborators.

`LinearUpwind` class

```
LinearUpwind(factory: object | None = None) -> None
```

Select a linear upwind reconstruction for the compatible FV convective operator.

Methods and properties.

```
bind(*, geometry, gradient, reconstruction=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_scalar_advection(*, geometry, gradient, reconstruction=None)
```

Construct the scalar advection flux collaborator for the selected carrier.

`MimeticCurl` class

```
MimeticCurl(*, factory: object = None) -> None
```

Choose the mimetic discrete curl route for supported electromagnetic models.

Methods and properties.

```
bind(*, shape, extents, wave_number_squared, relative_permittivity, source, field_id)
```

Bind this declaration to the supplied execution context and collaborators.

`MinmodLimiter` class

```
MinmodLimiter() -> None
```

Limit two scalar one-sided slopes with minmod to obtain a TVD slope on the supported stencil.

Methods and properties.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Selected generalized derivative; ties use the executed right branch.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`MonotoneUpwind` class

```
MonotoneUpwind(*, factory: object | None = None) -> None
```

Select scalar upwinding with a physical flux monotone in the declared direction.

Methods and properties.

```
bind(*, physical_flux, direction, wave_speed=None, physical_flux_jacobian=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
bind_conservation(*, law)
```

Bind the conservative operator using the physical flux and geometry supplied by the model.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`MonotonizedCentralLimiter` class

```
MonotonizedCentralLimiter() -> None
```

Use a monotonized central TVD slope with less flattening than minmod in smooth regions.

Methods and properties.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Selected generalized derivative, preserving the minimum's tie order.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`MUSCL` class

```
MUSCL(*, factory: object | None = None) -> None
```

Reconstruct conservative piecewise-linear face states. Select the limiter separately.

Methods and properties.

```
bind(*, cell_widths, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

`OrientedCellComplex` class

```
OrientedCellComplex() -> None
```

Represent oriented incidence between discrete geometric entities, independently of the metric.

Methods and properties.

```
bind(*, edge_space, face_space)
```

Bind this declaration to the supplied execution context and collaborators.

`OrthogonalHodge` class

```
OrthogonalHodge() -> None
```

Select orthogonal metric Hodge maps on a supported orthogonal carrier.

Methods and properties.

```
bind(*, mesh, cell_quadrature_order, face_quadrature_order)
```

Bind this declaration to the supplied execution context and collaborators.

`RhieChow` class

```
RhieChow(factory: object | None = None) -> None
```

Select collocated face-velocity interpolation with pressure correction in a supported flow model.

Methods and properties.

```
bind(*, geometry, gradient)
```

Bind this declaration to the supplied execution context and collaborators.

`Rusanov` class

```
Rusanov(*, factory: object | None = None) -> None
```

Select the local Lax–Friedrichs flux using the physical model's wave-speed estimate.

Methods and properties.

```
bind_conservation(*, law)
```

Bind the conservative operator using the physical flux and geometry supplied by the model.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bind(*, equation_of_state=None, physical_flux=None, wave_speed=None, direction=None)
```

Bind this declaration to the supplied execution context and collaborators.

`ScharfetterGummel` class

```
ScharfetterGummel(factory: object | None = None, gradient: object | None = None) -> None
```

Choose an exponential-fitting flux for drift–diffusion. Supply a gradient collaborator if the binding requires one.

Methods and properties.

```
bind(*, geometry, gradient=None, reconstruction=None, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

`SuperbeeLimiter` class

```
SuperbeeLimiter() -> None
```

Choose a compressive TVD limiter for a compatible scalar slope reconstruction.

Methods and properties.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Selected min/max branch, including the executed tie ordering.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`TvbLimiter` class

```
TvbLimiter(base_limiter: Callable[[float, float], float], tvb_constant: float = 0.0,
    mesh_width: float = 1.0, base_tangent: object = None) -> None
```

Wrap a base limiter with a mesh-width and curvature threshold for suppressing limiting near smooth extrema.

Methods and properties.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`supports_jvp`. Check availability of a Jacobian-vector product for the selected path.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Evaluate the directional derivative of the limited result.

`TwoPointDiffusion` class

```
TwoPointDiffusion(*, factory: object | None = None) -> None
```

Select a two-point diffusion flux based on adjacent cell values.

Methods and properties.

```
bind(*, geometry, coefficients, periodic)
```

Bind this declaration to the supplied execution context and collaborators.

`VanLeerLimiter` class

```
VanLeerLimiter() -> None
```

Use a harmonic TVD slope where the one-sided slopes agree in sign.

Methods and properties.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Exact smooth-branch derivative, zero on the selected inactive branch.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

`VenkatakrishnanLimiter` class

```
VenkatakrishnanLimiter(epsilon: float = 1e-08) -> None
```

Use smooth slope limiting with a regularizing epsilon.

Methods and properties.

```
factor(delta: float, variation: float) -> float
```

Evaluate the limiter or trace scaling factor.

```
limit_slope(left_slope: float, right_slope: float) -> float
```

Apply the limiter to the given one-sided slopes.

```
factor_jvp(delta, variation, delta_direction, variation_direction)
```

Differentiate the selected regularized rational factor.

```
cell_factor(center, neighbors, increment)
```

Compute the reconstruction limiter factor for the given cell data.

```
cell_factor_jvp(center, neighbors, increment, center_direction, neighbors_direction,
    increment_direction)
```

Evaluate the directional derivative of the cell limiter factor.

```
limit(left_slope: float, right_slope: float) -> float
```

Return the limited state or slope for the supplied carrier and admissibility policy.

```
limit_jvp(left_slope, right_slope, left_direction, right_direction)
```

Differentiate the signed available variation and selected factor.

`Weno3Reconstructor` class

```
Weno3Reconstructor(epsilon: float = 1e-12) -> None
```

Reconstruct a scalar left trace with WENO3 weights on the supported uniform line stencil.

Methods and properties.

```
bind(*, cell_widths, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
reconstruct_left_interfaces(values: tuple[float, ...]) -> tuple[float, ...]
```

Reconstruct the left state at each supported interface from cell samples.

`Weno5Reconstructor` class

```
Weno5Reconstructor(epsilon: float = 1e-12) -> None
```

Reconstruct a scalar left trace with WENO5 weights on the supported uniform line stencil.

Methods and properties.

```
bind(*, cell_widths, limiter=None)
```

Bind this declaration to the supplied execution context and collaborators.

```
reconstruct_left_interfaces(values: tuple[float, ...]) -> tuple[float, ...]
```

Reconstruct the left state at each supported interface from cell samples.

`WetDryActiveSet` class

```
WetDryActiveSet() -> None
```

Select the wet/dry active-set treatment for a compatible shallow-water linearization.

Methods and properties.

```
inactive_dry_nodes(values, rate)
```

Identify nodes treated as inactive by the dry-state rule.

```
bind_flux(flux, law)
```

Bind the face flux with the supplied physical and geometric collaborators.

```
linearize(*, residual, state, context)
```

Evaluate the local derivative or directional linearization of the operation.

```
linearization_law(law)
```

Return the selected linearization policy.

```
reconstructed_direction(state, star, direction, *, bed, high)
```

Reconstruct the supplied direction for the tangent calculation.

`WetDryTraceScaling` class

```
WetDryTraceScaling() -> None
```

Retract depth and discharge traces by one factor toward admissible means.

Methods and properties.

```
bind(*, admissible)
```

Bind this declaration to the supplied execution context and collaborators.

`WhitneyHodge` class

```
WhitneyHodge(edge_reconstruction: object | None = None, face_reconstruction: object | None =
    None) -> None
```

Use Whitney reconstruction for metric edge and face maps, with optional reconstruction collaborators.

Methods and properties.

```
bind(*, mesh, cell_quadrature_order, face_quadrature_order)
```

Bind this declaration to the supplied execution context and collaborators.

B.8 onesim.numerics.linear

`AlgebraicMultigrid` class

```
AlgebraicMultigrid(*, near_nullspace: object | None = None) -> None
```

Use a compatible multigrid preconditioner with optional near-nullspace information.

`BlockPreconditioner` class

```
BlockPreconditioner(*, solvers: tuple, blocks: tuple | None = None, sweep: str = 'diagonal')
    -> None
```

Use a diagonal or triangular sweep of inner solvers over an explicit or provider-owned block partition.

Methods and properties.

```
build(*, context=None)
```

Construct the bound executable component for the supplied context.

`ComplexSparseDirect` class

```
ComplexSparseDirect(*, preconditioner: object | None = None, scalar_type: str = 'complex',
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations:
    int = 1000, factory: object | None = None) -> None
```

Select the complex sparse direct route, for example for frequency-domain Maxwell phasors.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`ConjugateGradient` class

```
ConjugateGradient(*, preconditioner: object | None = None, scalar_type: str = 'real',
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations:
    int = 1000, factory: object | None = None) -> None
```

Solve a compatible positive-definite symmetric or Hermitian linear system. Check the operator and preconditioner guarantees.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`FlexibleGMRES` class

```
FlexibleGMRES(*, preconditioner: object | None = None, scalar_type: str = 'real',
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations:
    int = 1000, factory: object | None = None, restart: int | None = None) -> None
```

Use flexible right preconditioning, retaining the actual preconditioned search vectors.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`GMRES` class

```
GMRES(*, preconditioner: object | None = None, scalar_type: str = 'real', relative_tolerance:
    float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations: int = 1000, factory:
    object | None = None, restart: int | None = None) -> None
```

Select restarted or unrestarted GMRES with tolerances, scalar type, and an optional compatible preconditioner.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`ILU` class

```
ILU(*, fill_factor: float = 1.0, drop_tolerance: float = 0.0) -> None
```

Use incomplete sparse factorization with selectable fill and drop tolerance.

Methods and properties.

`method_id`. Read the registered method identity.

`RightGMRES` class

```
RightGMRES(*, preconditioner: object | None = None, scalar_type: str = 'real',
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations:
    int = 1000, factory: object | None = None, restart: int | None = None) -> None
```

Use fixed right preconditioning and check convergence on the physical residual.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`SchurPreconditioner` class

```
SchurPreconditioner(*, solvers: tuple, blocks: tuple | None = None, approximation: object |
    None = None) -> None
```

Use a leading-block solve and Schur-complement solve, with an optional approximation.

Methods and properties.

```
build(*, context=None)
```

Construct the bound executable component for the supplied context.

`SparseDirect` class

```
SparseDirect(*, preconditioner: object | None = None, scalar_type: str = 'real',
    relative_tolerance: float = 1e-08, absolute_tolerance: float = 1e-12, maximum_iterations:
    int = 1000, factory: object | None = None) -> None
```

Select assembled sparse factorization for the declared scalar type and available provider.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

B.9 onesim.numerics.nonlinear

`DampedNewton` class

```
DampedNewton(*, linear_solver: object | None = None, relaxation: float = 1.0, jacobian: object
    = 'analytic', factory: object | None = None, relative_tolerance: float = 1e-06,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30,
    maximum_relative_decrease: float = 1.0, line_search_reduction: float = 0.5,
    minimum_damping: float = 1e-06) -> None
```

Add Armijo backtracking and a minimum damping factor to the Newton solve.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`FixedPoint` class

```
FixedPoint(*, factory: object | None = None, relative_tolerance: float = 1e-05,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30, relaxation: float = 1.0)
    -> None
```

Iterate a compatible fixed-point map with relaxation, tolerances and a maximum iteration count.

Methods and properties.

```
bind(*, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`JacobianFreeNewtonKrylov` class

```
JacobianFreeNewtonKrylov(*, linear_solver: object | None = None, jacobian: object | None =
    None, factory: object | None = None, relative_tolerance: float = 1e-06,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30, relaxation: float = 1.0,
    maximum_relative_decrease: float = 1.0) -> None
```

Long name for JFNK. Supply the same derivative action, inner solver and nonlinear controls.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`JFNK` class

```
JFNK(*, linear_solver: object | None = None, jacobian: object | None = None, factory: object |
    None = None, relative_tolerance: float = 1e-06, absolute_tolerance: float = 1e-10,
    maximum_iterations: int = 30, relaxation: float = 1.0, maximum_relative_decrease: float =
    1.0) -> None
```

Select Newton–Krylov with a supported Jacobian action and explicit compatible Krylov solver.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`Newton` class

```
Newton(*, linear_solver: object | None = None, relaxation: float = 1.0, jacobian: object =
    'analytic', factory: object | None = None, relative_tolerance: float = 1e-06,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30,
    maximum_relative_decrease: float = 1.0) -> None
```

Solve a nonlinear residual using a selected Jacobian route and inner linear solver. Configure tolerances and relaxation.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`QuasiNewton` class

```
QuasiNewton(*, linear_solver: object | None = None, relaxation: float = 1.0, jacobian: object
    = 'analytic', factory: object | None = None, relative_tolerance: float = 1e-06,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30,
    maximum_relative_decrease: float = 1.0) -> None
```

Use inverse good-Broyden updates following the selected initial linear solve.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

`TrustRegionNewton` class

```
TrustRegionNewton(*, linear_solver: object | None = None, relaxation: float = 1.0, jacobian:
    object = 'analytic', factory: object | None = None, relative_tolerance: float = 1e-06,
    absolute_tolerance: float = 1e-10, maximum_iterations: int = 30,
    maximum_relative_decrease: float = 1.0, trust_radius: float = 1.0, minimum_damping: float
    = 1e-06) -> None
```

Limit the correction in solver coordinates with an adaptive trust radius.

Methods and properties.

```
bind(*, linear_solver=None, factory=None, context=None)
```

Bind this declaration to the supplied execution context and collaborators.

B.10 onesim.numerics.time

`AdaptiveTimeStep` class

```
AdaptiveTimeStep(*, initial: float, minimum: float, maximum: float, relative_tolerance: float
    = 0.001, absolute_tolerance: float = 1e-10, safety: float = 0.9, maximum_rejections: int =
    32, error_estimator: object | None = None) -> None
```

Pass as run(time_step=...) to choose the trial step range, error tolerances, safety and rejection limits.

Methods and properties.

```
next_step(step, normalized_error, *, order)
```

Compute the controller's proposed step after the supplied error and step history.

`BackwardEuler` class

```
BackwardEuler() -> None
```

Select first-order implicit Euler and supply a compatible nonlinear stage solver when required.

`BDF` class

```
BDF(*, order: int = 2, startup: object = <factory>, maximum_step_ratio: float = 2.0) -> None
```

Select backward differentiation order, startup policy and maximum step ratio. Keep its history for continuation.

`ClassicalRungeKutta4` class

```
ClassicalRungeKutta4() -> None
```

Select the four-stage explicit fourth-order RK method. The time step must resolve the spatial stability bound.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`ForwardEuler` class

```
ForwardEuler() -> None
```

Select first-order explicit Euler with separately declared step limits.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`TimeEventDefinition` class

```
TimeEventDefinition(event_id: str, function: Callable[[float, object], float], direction: int
    = 0, terminal: bool = False, reset: Callable[[float, object], object] | None = None,
    expression: str | None = None, observation_sha256: str | None = None, value_tolerance:
    float = 0.0) -> None
```

Declare a scalar semantic-state event, optional crossing direction, terminal behavior and reset.

Methods and properties.

```
to_mapping() -> dict[str, object]
```

Serialize event semantics without embedding executable Python.

`TimeExecutionBinding` class

```
TimeExecutionBinding(method_id: str, order: int, stage_configuration: object,
    startup_method_id: str | None = None, maximum_step_ratio: float | None = None,
    supports_stage_projection: bool = False, stage_projection_equation: str | None = None) ->
    None
```

Read the bound method, nominal order, stage configuration, startup policy and stage-projection capabilities.

`TimeMethod` class

```
TimeMethod(method_id: str) -> None
```

Select a registered time algorithm by method_id, then supply the stage solver required by its binding.

Methods and properties.

`order`. Read the nominal selected order.

`implicit`. Read whether the time method is implicit.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

`TimeStepLimits` class

```
TimeStepLimits(*, maximum_step: float | None = None, stability_safety_factor: float = 0.5,
    cfl_limit: float | None = None) -> None
```

Pass through Numerics.step_limits to bound the maximum step, stability safety factor and optional CFL.

Methods and properties.

```
to_mapping()
```

Return the mapping representation used for configuration and diagnostics.

```
bound(stability_step=None, *, courant_rate=None)
```

Combine independent bounds, scaling the spatial bound exactly once.

APPENDIX C

Exposed operator and service namespaces

These exports sit behind a module object. Use the indicated import, then access the named function or class on that object. The operator functions assemble or adapt provider-owned data and are intended for studies that explicitly own their spatial support, coupling and provider contracts. Their signatures and return annotations give the entry points for those studies.

C.1 onesim.numerics.field_transfer.operators

The executable field-transfer examples appear in Sections 9.8.2–9.8.5.

```
from onesim.numerics.field_transfer import operators
```

apply function

```
apply(transfer_operator: TransferOperator | TransferDescriptor, source: object, *, runtime:
    TransferRuntime | None = None, execution_profile: object | None = None) ->
    TransferExecutionResult
```

Apply a transfer through the common runtime.

conservative_remap function

```
conservative_remap(name: str, source_layout_id: str, target_layout_id: str, *, source_edges:
    Sequence[float], target_edges: Sequence[float], source_unit: str = '1', target_unit: str =
    '1', source_topology: str = 'cell', target_topology: str = 'cell', conservation_tolerance:
    float = 1e-10, provider_id: str | None = None) -> TransferOperator
```

Declare a conservative piecewise-constant cell remap.

identity function

```
identity(name: str, layout_id: str, *, unit: str = '1', topology: str = 'unspecified',
    conservative: bool = False, provider_id: str | None = None) -> TransferOperator
```

Declare a same-layout transfer.

interpolation function

```
interpolation(name: str, source_layout_id: str, target_layout_id: str, *, source_coordinates:
    Sequence[float], target_coordinates: Sequence[float], source_unit: str = '1', target_unit:
    str = '1', source_topology: str = 'unspecified', target_topology: str = 'unspecified',
    provider_id: str | None = None) -> TransferOperator
```

Declare a point/field interpolation between two layouts.

operator function

```
operator(name: str, descriptor: TransferDescriptor, *, provider_id: str | None = None,
    provider_version: str | None = None) -> TransferOperator
```

Create a named transfer without selecting a provider class.

prescribed_field function

```
prescribed_field(values: object, spatial_carrier: object, field_id: str, *, unit: str,
    value_kind: str, value_type: str = 'real', carrier_kind: str = 'cell', input_id: str |
    None = None, target_system_id: str | None = None, target_field_id: str | None = None,
    transfer: str | None = None, provider_id: str | None = None, coverage_policy: str =
    'centroids', extrapolation: str = 'reject', conservation_tolerance: float = 1e-10) ->
    SolvedSpatialFieldInput
```

Declare mesh-bound prescribed data, without a fictitious solved PDE.

Snapshot ownership, named target ports and explicitly selected transfers are the same shared backend contract as solved_field().

runtime function

```
runtime(*, linear=None, projection_kernel='auto', projection_parallel=False,
    execution_profile=None, linear_solver=None, numerical_registry=None)
```

Configure nodal projection's solver and independent sparse RHS execution.

`projection_kernel` selects auto/NumPy/Numba; auto uses Numba when installed. Explicit parallel execution requires Numba and is never silently ignored. These options do not override the linear solver node.

simplex_nodal_interpolation function

```
simplex_nodal_interpolation(name: str, source_mesh: object, target_mesh: object, *, unit: str
    = '1', provider_id: str | None = None) -> TransferOperator
```

Declare nonconservative 2D/3D nodal simplex interpolation.

The source needs ordered triangle/Tet4 connectivity and the target needs point coordinates. Every target point must be covered by a source simplex.

solved_field function

```
solved_field(result: object, field_id: str, *, unit: str, value_kind: str | None = None,
    value_type: str | None = None, component_count: int | None = None, input_id: str | None =
    None, target_system_id: str | None = None, target_field_id: str | None = None, transfer:
    str | None = None, provider_id: str | None = None, coverage_policy: str = 'centroids',
    extrapolation: str = 'reject', conservation_tolerance: float = 1e-10) ->
    SolvedSpatialFieldInput
```

Declare a qualified result field for later use on a Study target mesh.

The declaration retains source-result and mesh fingerprints but performs no mapping until `input.resolve(target_carrier)` is called. All methodology adapters can use the same field handoff contract. Point fields require the identical carrier; cross-mesh remaps currently require Tet4 cell support. 1D point fields accept explicit linear interpolation; 2D/3D nodal simplex fields accept explicit barycentric interpolation. Interpolation rejects uncovered target coordinates unless `extrapolation='clamp'` is explicitly requested. `target_system_id` binds the input to a named Study subsystem; an optional `target_field_id` names the receiving callback input, while `input_id` disambiguates multiple declared sources.

tetrahedral_cell_projection function

```
tetrahedral_cell_projection(name: str, source_mesh: object, target_mesh: object, *, unit: str
    = '1', provider_id: str | None = None, coverage_policy: str = 'centroids') ->
    TransferOperator
```

Declare a Tet4 cell-field projection with no outside extrapolation.

Accepts neutral mesh carriers or result `mesh_support` mappings. This centroid-sampling method is explicitly non-conservative; set `coverage_policy='vertices_and_centroids'` for strict node coverage. Use a separate volume-overlap remap when an integral conservation guarantee is required.

`tetrahedral_conservative_remap` function

```
tetrahedral_conservative_remap(name: str, source_mesh: object, target_mesh: object, *, unit:
    str = '1', conservation_tolerance: float = 1e-10, provider_id: str | None = None) ->
    TransferOperator
```

Declare exact Tet4 overlap remap, requiring equal polyhedral domains.

`validate` function

```
validate(transfer_operator: TransferOperator | TransferDescriptor, *, source: object | None =
    None, runtime: TransferRuntime | None = None) -> TransferValidation
```

Run the provider-neutral transfer preflight.

C.2 onesim.physics.electromagnetics.operators

The executable electromagnetic operator examples appear in Sections 6.7.6–6.7.8.

```
from onesim.physics.electromagnetics import operators
```

`maxwell_dae_heating` function

```
maxwell_dae_heating(base_operator, constraint, *, energy_field, energy_unit=None,
    power_field='absorbed_power_density', volumetric_heat_capacity=None,
    heat_capacity_unit='J/(m^3*K)')
```

Add native candidate-coupled Maxwell power to one complete DAE rate.

Infers physical energy-density units from provider result metadata, or accepts an explicit unit. Preserves all chemistry/transport/Poisson rows; energy conversion and sparse cross derivatives remain backend-owned.

`maxwell_heated_equation` function

```
maxwell_heated_equation(equation, *, energy_field, power_field='absorbed_power_density',
    electric_fields=('electric_real', 'electric_imaginary'), material_fields=(),
    volumetric_heat_capacity=None, heat_capacity_unit='J/(m^3*K)', provider_owned_mass=False)
```

Declare conservative native Maxwell work in one energy equation.

Retains the original chemistry/transport/external heating terms. Native work is an additional source in the energy state's declared units; W/m^3 power is derived, not another time-evolved degree of freedom. Runtime acknowledgement still requires an actual provider-owned source. This helper alone does not supply a coupled Study factory.

`nedelec_curl_curl_operator` function

```
nedelec_curl_curl_operator(request: object, configuration: object, **options: object) -> object
```

Prepare the complex Nedelec Maxwell pre-solve for `StationaryStudy`.

nedelec_maxwell_dae_constraint function

```
nedelec_maxwell_dae_constraint(*, coordinates, cells, carrier_id, source,
    curl_coefficient=1.0, mass_coefficient=1.0, essential_edge_ids=None,
    essential_edge_values=None, boundaries=(), material=None, material_fields=None,
    material_state=None, material_coefficient_scale=1.0,
     electric_real_field='electric_edge_real', electric_imaginary_field='electric_edge_imaginar
    y', field_id='electric_field', service=None, configuration=None, element_order=1,
    quadrature_order=None)
```

Native Tet4/Hex8 Maxwell constraints with complete H(curl) moments.

The common DAE solver owns the full material/edge candidate. This factory assembles but never invokes an inner Maxwell solve. Material coefficients and forcing are explicitly supplied weak-form data, not inferred units.

nedelec_maxwell_dae_equations function

```
nedelec_maxwell_dae_equations(*, domain='', material_fields=(),
    electric_real_field='electric_edge_real', electric_imaginary_field='electric_edge_imaginary
    ', surface_terms=False, essential_traces=False, material_input_fields=())
```

Declare executable Tet4 edge-integral Maxwell/PEC residuals.

Electric unknowns are oriented integrals in V; physical weak rows are V/m. This does not select the base transport method or infer material data.

nodal_curl_curl_operator function

```
nodal_curl_curl_operator(request: object, configuration: object, **options: object) -> object
```

Prepare the real nodal curl-curl FEM system for `StationaryStudy`.

periodic_maxwell_dae_constraint function

```
periodic_maxwell_dae_constraint(operator, *, material_fields,
    electric_real_field='electric_real', electric_imaginary_field='electric_imaginary')
```

Build backend-owned Maxwell constraint/codec derivatives for a DAE.

A dependency selects a scalar field by its ID, or a vector component by `{"field_id": "B", "component": 0}`. Multiple dependencies may select the same field/component; their analytic partials are added in the actual semantic codec. Component indices are zero-based, never inferred.

periodic_maxwell_dae_equations function

```
periodic_maxwell_dae_equations(*, domain='', material_fields=(),
    electric_real_field='electric_real', electric_imaginary_field='electric_imaginary')
```

Declare the two integrated native-edge Maxwell constraints.

Material fields name the actual cell-state dependencies (declared by their own equations). This describes the native periodic FV dialect; it does not select a provider or pretend to supply a coupled Study factory. The electric fields are algebraic real tangential edge values.

periodic_maxwell_dae_heating function

```
periodic_maxwell_dae_heating(base_operator, constraint, **settings)
```

Compatibility name for the common native Maxwell energy composition.

periodic_maxwell_energy_rate function

```
periodic_maxwell_energy_rate(base_operator, constraint, *, energy_field,
    power_field='absorbed_power_density', energy_unit=None, volumetric_heat_capacity=None,
    heat_capacity_unit='J/(m^3*K)')
```

Add native Maxwell absorption to an owned cell energy/temperature rate.

The common DAE adapter owns full implicit stages and codec ordering. The base operator owns transport/chemistry/cooling; this adapter never launches a separate Maxwell solve. Uniform periodic FV, physical-real cell rate only. Temperature requires explicit fixed volumetric heat capacity and its unit; energy-density states use their owned unit scale without another capacity.

periodic_maxwell_fvm_operator function

```
periodic_maxwell_fvm_operator(*, shape, extents, wave_number_squared, relative_permittivity,
    source, field_id='electric_field', material_state=None)
```

Prepare uniform periodic mimetic FVM Maxwell for StationaryStudy.

Extents are in metres; source is edge-sampled curl-curl forcing (V/m^3). Permittivity is a scalar or constant complex 3x3 Cartesian tensor. Native edge-diagonal arrays and MaterialPropertyEvaluator spatial scalar or tensor expressions are supported. The backend samples x/y/z at native edges or shared staggered edge-pair midpoints respectively. Explicit material_state binds real scalar parameters or native cell-centre fields for tensor IR. material_jvp includes their physical transfer. This does not substitute for STEP/PEC or a fully coupled ECR runtime.

periodic_maxwell_heated_equation function

```
periodic_maxwell_heated_equation(equation, **settings)
```

Compatibility name for the discretization-neutral heating declaration.

periodic_maxwell_material_coupling function

```
periodic_maxwell_material_coupling(coupling_id, *, target, source, operator, dependency)
```

Compose realified Maxwell and explicit native-cell material in Newton.

periodic_maxwell_power_coupling function

```
periodic_maxwell_power_coupling(coupling_id, *, target, source, operator, dependency,
    residual_measure)
```

Return native Maxwell absorption to an energy balance in common Newton.

Pair with periodic_maxwell_material_coupling in the opposite direction. The target cell balance must explicitly choose density (W/m^3) or cell_integral (W) residuals. No prescribed-power normalization is applied. Native peak phasors use exp(-i omega t). Tensor deposition may be signed locally; clipping is deliberately forbidden. This is periodic FV only.

C.3 onesim.physics.plasma.operators

The executable plasma operator examples appear in Sections 6.17.9–6.17.10.

```
from onesim.physics.plasma import operators
```

cell_transport_dae function

```
cell_transport_dae(base_operator)
```

Expose provider-owned cell transport and Poisson via common DAE methods.

Preserves chemistry and face laws. The selected provider must explicitly publish physical semidiscrete rates, algebraic balances and sparse derivatives; backward-Euler residuals are never guessed to be rates.

`periodic_cell_poisson_dae_constraint` function

```
periodic_cell_poisson_dae_constraint(shape, extents, *, charge_fields, **options)
```

Native periodic FV Gauss law with one explicit global mean gauge.

Charge fields are scalar SI number densities; mapping values are signed particle charges in coulombs. All cell balances remain in the coupled DAE. Nonneutral accepted states are rejected, never repaired by a background.

`periodic_cell_transport_dae` function

```
periodic_cell_transport_dae(base_operator, poisson_constraint)
```

Extend actual native periodic transport with its global Poisson gauge.

Retains provider chemistry/face laws and full sparse cross derivatives. A different native Poisson source is rejected, not silently replaced by the declared charge inventory. Initial snapshots name the global DOF.

`periodic_cell_transport_request` function

```
periodic_cell_transport_request(shape, extents, *, initial_fields, potential_field_index,
    permittivity, coefficients, volume_rates, **options)
```

Prepare native closed-periodic FV geometry without artificial walls.

The caller supplies physical coefficients and volume laws. The resulting cell-only Poisson block is singular; solving requires an explicit global gauge through the common DAE, not the legacy cell-only time solve.

`plasma_condition_binding` function

```
plasma_condition_binding(provider: object, problem_template: object, **options: object) ->
    object
```

Create the backend-owned IR-to-plasma condition adapter.

`plasma_operator` function

```
plasma_operator(provider: object, problem: object, *, final_time: float | None = None,
    field_ids: Sequence[str] = (), field_units: Sequence[str] = ()) -> object
```

Expose an explicit multispecies plasma provider through `Study`.

The provider owns chemistry and face-resolved plasma boundary semantics; the common runner owns time integration and result lifecycle. Requiring the provider and its typed problem as arguments keeps that boundary explicit and prevents a generic numerical factory from guessing a plasma state or mesh interpretation.

C.4 onesim.physics.heat_transfer.advanced_thermal

```
from onesim.physics.heat_transfer import advanced_thermal
```

`AdvancedThermalFeatureService` class

```
AdvancedThermalFeatureService()
```

Validate and lower every advanced Thermal feature through one path.

Selected methods.

```
validate(*, features: tuple[Mapping[str, object], ...], physics_ids: tuple[str, ...],
    domain_ids: tuple[str, ...], boundary_ids: tuple[str, ...], material_ids: tuple[str, ...],
    mesh_reference: Mapping[str, object] | None) -> AdvancedThermalValidation
```

Validate product scopes and normalize executable Thermal features.

`AdvancedThermalValidation` class

```
AdvancedThermalValidation(diagnostics: tuple[str, ...], normalized_features: tuple[dict[str,
    object], ...], provider_id: str = 'onesimulation.internal.advanced-thermal') -> None
```

Normalized advanced-thermal plan and deterministic diagnostics.

Selected methods.

```
to_mapping() -> dict[str, object]
```

Serialize the normalized validation result.

C.5 onesim.physics.heat_transfer.configuration

```
from onesim.physics.heat_transfer import configuration
```

`THERMAL_CONFIGURATION_SCHEMA` value

Identifier of the thermal configuration schema.

`THERMAL_MODEL_ALIASES` value

Model-name mapping used by the thermal configuration parser.

`ThermalConfiguration` class

```
ThermalConfiguration(provider: str, model: str, mesh: str, discretization:
    ThermalSpatialFamily | str, time_integration: str = 'classical_rk4', solver: str =
    'explicit_block', boundary_conditions: tuple[str, ...] = (), material_id: str = 'default',
    parameters: Mapping[str, object] = <factory>, version: int = 1, dimension: int = 3,
    coordinate_system: str = 'cartesian', material: Mapping[str, object] = <factory>,
    source_terms: tuple[str, ...] = (), source_data: Mapping[str, object] = <factory>,
    boundary_data: Mapping[str, object] = <factory>, time_model: str = 'transient', coupling:
    str | None = None, output_options: Mapping[str, object] = <factory>, units: Mapping[str,
    str] = <factory>, provenance: Mapping[str, str] = <factory>, capability_ids: tuple[str,
    ...] = ()) -> None
```

Select thermal physics separately from shared numerical providers.

Selected methods.

```
from_mapping(values: Mapping[str, object], *, unknown_fields: UnknownFieldPolicy | str =
    <UnknownFieldPolicy.REJECT: reject>) -> "'ThermalConfiguration'"
```

Build a configuration from the versioned public mapping schema.

```
to_numerical_configuration() -> NumericalConfiguration
```

```
to_simulation_configuration() -> SimulationConfiguration
```

Convert to the legacy problem-first envelope without data loss.

```
to_mapping() -> dict[str, object]
```

`UnknownFieldPolicy` class

```
UnknownFieldPolicy(*values)
```

Policy applied when a public Thermal mapping contains extra keys.

C.6 onesim.physics.heat_transfer.errors

```
from onesim.physics.heat_transfer import errors
```

`diagnostic_for` function

```
diagnostic_for(message: str, *, field: str | None = None) -> ThermalDiagnostic
```

Map a public validation/execution message to its stable error code.

`ThermalConfigurationError` class

```
ThermalConfigurationError(diagnostics: Iterable[ThermalDiagnostic]) -> None
```

Raised by the public execution boundary with structured diagnostics.

`ThermalDiagnostic` class

```
ThermalDiagnostic(code: ThermalErrorCode, message: str, field: str | None = None, hint: str |
    None = None) -> None
```

A thermal diagnostic with code, message and optional field and hint.

Selected methods.

```
to_mapping() -> dict[str, str]
```

Return code and message, plus non-None field and hint values.

`ThermalErrorCode` class

```
ThermalErrorCode(*values)
```

Stable identifiers for public thermal configuration and execution diagnostics.

C.7 onesim.physics.heat_transfer.heat_conduction

```
from onesim.physics.heat_transfer import heat_conduction
```

HeatConductionExecutionRequest class

```
HeatConductionExecutionRequest(problem: object, configuration: ThermalConfiguration,
    numerical_configuration: NumericalConfiguration, numerics: ThermalNumericsBinding,
    operators: ThermalHeatConductionOperatorBinding, fem_contract:
    ThermalFEMWeakDiffusionContract | None = None, fvm_contract:
    ThermalFVMConservativeFluxContract | None = None, fdm_reference:
    ThermalFDMStructuredReference | None = None, dg_contract: ThermalDGWeakDiffusionContract |
    None = None, spectral_element_contract: ThermalSpectralElementContract | None = None,
    nonlinear_jacobian: ThermalNonlinearJacobianContract | None = None,
    time_integrator_reference: ThermalTimeIntegratorReference | None = None) -> None
```

Complete provider-neutral request for `thermal.heat_conduction`.

Selected methods.

```
from_configuration(problem: object, configuration: ThermalConfiguration) ->
    "'HeatConductionExecutionRequest'"
```

```
to_mapping() -> dict[str, object]
```

```
resolve_spatial_execution_plan(registry: object, *, plan_id: str | None = None) ->
    NumericalExecutionPlan
```

Resolve the selected FEM/FVM contract through the shared registry.

HeatConductionProvider class

```
HeatConductionProvider(*args, **kwargs)
```

Public provider contract required by `run_heat_conduction`.

Selected methods.

```
execute(request: HeatConductionExecutionRequest) -> object
```

Execute the request using registered shared numerical providers.

C.8 onesim.physics.heat_transfer.heat_transport

```
from onesim.physics.heat_transfer import heat_transport
```

HeatTransportExecutionRequest class

```
HeatTransportExecutionRequest(problem: object, configuration: ThermalConfiguration,
    formulation: ThermalAdvectionDiffusionFormulation, physics_ir: ThermalPhysicsIR,
    fluid_field_transfer: ThermalFluidFieldTransfer) -> None
```

Separate transferred fluid input, thermal state, and material inputs.

Selected methods.

```
from_configuration(problem: object, configuration: ThermalConfiguration, velocity_transfer:
    ThermalVelocityTransfer, *, fluid_field_transfer: ThermalFluidFieldTransfer | None = None,
    sources: tuple[ThermalSource, ...] = (), boundary_conditions:
    tuple[ThermalBoundaryCondition, ...] = ()) -> "'HeatTransportExecutionRequest'"
```

```
to_mapping() -> dict[str, object]
```

C.9 onesim.physics.heat_transfer.public_api

```
from onesim.physics.heat_transfer import public_api
```

ThermalExecutionPlan class

```
ThermalExecutionPlan(configuration: ThermalConfiguration, numerical_configuration:
    NumericalConfiguration, numerics: ThermalNumericsBinding, capabilities:
    ThermalCapabilityReport, numerical_plan: NumericalExecutionPlan | None = None) -> None
```

Provider-neutral Thermal execution plan returned by :meth:plan.

Selected methods.

```
to_mapping() -> dict[str, object]
```

ThermalPublicApi class

```
ThermalPublicApi(provider_factory: Callable[..., object] | None = None, capability_matrix:
    ThermalCapabilityMatrix | None = None, numerical_registry: object | None = None, *,
    provider_registry: object | None = None) -> None
```

Expose all executable Thermal functionality through one public facade.

Selected methods.

```
bind_numerics(configuration: ThermalConfiguration) -> ThermalNumericsBinding
```

```
validate(problem: object | None, configuration: ThermalConfiguration | Mapping[str, object])
    -> tuple[ThermalDiagnostic, ...]
```

Return deterministic structured diagnostics without constructing a provider.

```
validate_or_raise(problem: object | None, configuration: ThermalConfiguration | Mapping[str,
    object]) -> ThermalConfiguration
```

Validate a public declaration before any provider is constructed.

```
validation_report(problem: object | None, configuration: ThermalConfiguration | Mapping[str,
    object]) -> ThermalCapabilityValidationReport
```

Return capability proof and all public validation diagnostics.

```
capabilities(configuration: ThermalConfiguration | Mapping[str, object]) ->
    ThermalCapabilityReport
```

Evaluate the configured physics/numerics tuple against the public matrix.

```
plan(configuration: ThermalConfiguration | Mapping[str, object], *, physics_ir:
    ThermalPhysicsIR | None = None) -> ThermalExecutionPlan
```

Build a provider-neutral plan; provider construction remains in `run`.

```
describe(configuration: ThermalConfiguration | Mapping[str, object] | None = None) ->
    dict[str, object]
```

Return a serializable public catalogue or one configuration description.

```
run(problem: object, configuration: ThermalConfiguration | Mapping[str, object]) -> object
```

```
run_heat_conduction(problem: object, configuration: ThermalConfiguration | Mapping[str,
    object]) -> object
```

Execute the explicitly supported `thermal.heat_conduction` variant.

```
diagnose_energy_balance(*, energy_flux: Mapping[str, float], total_energy: float,
    source_balance: float, total_energy_change: float, tolerance: float = 1e-10) ->
    ThermalEnergyBalanceReport
```

Normalize provider balance values into the public Thermal report.

```
run_heat_transport(problem: object, configuration: ThermalConfiguration | Mapping[str,
    object], velocity_transfer: object, fluid_field_transfer: ThermalFluidFieldTransfer | None
    = None) -> object
```

Execute `thermal.heat_transport` through the public transfer contract.

```
run_mapping(problem: object, configuration: Mapping[str, object]) -> object
```

Execute a versioned JSON-compatible Thermal configuration mapping.

C.10 onesim.physics.heat_transfer.validation_report

```
from onesim.physics.heat_transfer import validation_report
```

`ThermalCapabilityValidationReport` class

```
ThermalCapabilityValidationReport(configuration: ThermalConfiguration, capability:
    ThermalCapabilityReport, diagnostics: tuple[ThermalDiagnostic, ...] = ()) -> None
```

Combine configuration diagnostics with the matched capability proof.

Selected methods.

```
to_mapping() -> dict[str, object]
```