LayTracer

Fast two-point seismic ray tracing in 1-D layered media

Denis Anikiev

Version 0.3.1.dev1+g09ac0ec22

March 14, 2026

CONTENTS

LayTracer is an open-source Python package for computing ray paths, travel times, and amplitude attributes in horizontally layered (1D) velocity models with constant layer velocities. It is based on the dimensionless ray parameter method of Fang and Chen [2019], achieving rapid convergence.

Documentation: danikiev.github.io/LayTracer

Current Version: 0.3.1.dev1+g09ac0ec22 (*Changelog*)

Features:

- Fast two-point ray tracing via dimensionless ray parameter method
- Second-order Newton iteration for rapid convergence
- Refraction and reflection modes
- Inline computation of travel time, attenuation operator t^*, geometrical spreading, and reflection/transmission coefficients
- Efficient parallel computations via Joblib
- Standalone Matplotlib / Plotly visualisation
- Comprehensive Sphinx documentation with extensive theory available at danikiev.github.io/LayTracer

---

Citing LayTracer

To cite a particular version of LayTracer, please use the following format, e.g. for version 0.3.0:

Anikiev, D. (2026). *LayTracer: Fast two-point seismic ray tracing in layered media (0.3.0).* Zenodo. https://doi.org/10.5281/zenodo.19020694

To cite the collection of all versions of LayTracer, please use the following format:

Anikiev, D. (2026). *LayTracer: Fast two-point seismic ray tracing in layered media.* Zenodo. https://doi.org/10.5281/zenodo.18850919

This DOI represents all versions, and will always resolve to the latest one.

CHAPTER

ONE

GETTING STARTED

This chapter provides everything you need to begin using LayTracer. It includes *installation instructions* including *dependency management* and general description of the *package design*.

1.1 Requirements

LayTracer can be installed either with Conda or with standard Python `pip`. For a reproducible full environment, we recommend using the miniforge Conda implementation.

1.2 Installation

1.2.1 Dependencies

LayTracer leverages numerous Python packages. Below are the key dependencies:

- Python (3.8 to 3.12)
- NumPy (<2)
- SciPy
- Pandas
- psutil
- joblib
- Matplotlib
- Plotly
- cmcrameri

Optional dependencies used to build documentation include:

- Sphinx
- pydata-sphinx-theme
- sphinx-gallery
- sphinxcontrib-bibtex
- sphinx-design

- numpydoc

The authors are incredibly grateful to the developers of these packages.

1.2.2 Install using conda

The best way to install is by creating a new conda environment with all required packages:

```
conda env create -f environment.yml
```

Note: to speed up creation of the environment, use *mamba* instead of *conda*, which is a faster alternative.

Then activate the newly created environment:

```
conda activate laytracer
```

Finally, install the package:

```
pip install -e .
```

1.2.3 Install from PyPI (stable releases only)

If you want the latest published stable release, install directly from PyPI:

```
python -m pip install --upgrade pip
pip install laytracer
```

Use this mode for stable releases. For development work or unreleased changes, install from the repository with `pip install -e .`.

1.2.4 Install using pip only

If you prefer not to use Conda, you can install LayTracer into a standard virtual environment.

On Linux / macOS:

```
python -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
pip install -e .
```

On Windows (PowerShell):

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
pip install -e .
```

The pip-only approach is appropriate for package usage. For a pre-configured environment including documentation tooling, prefer the Conda workflow.

1.2.5 Install using script

For quick installation, you can use the specially designed installation scripts which implement all of the above mentioned steps.

On Windows, in miniforge prompt run:

```
install.bat
```

On Linux / macOS, from the repository root run:

```
chmod +x install.sh
./install.sh
```

1.2.6 Build documentation using script

To build and serve the documentation quickly, use the platform-specific scripts.

On Windows:

```
build-docs.bat
```

Build HTML + PDF on Windows:

```
build-docs.bat -pdf
```

On Linux / macOS:

```
chmod +x build-docs.sh
./build-docs.sh
```

Build HTML + PDF on Linux / macOS:

```
chmod +x build-docs.sh
./build-docs.sh -pdf
```

1.2.7 Build documentation using make

If you prefer explicit Sphinx/Make commands, you can build the docs directly from the `docs` folder.

On Linux / macOS:

```
cd docs
make html
```

Build HTML + PDF on Linux / macOS:

```
cd docs
make html
make latexpdf
```

On Windows:

```
cd docs
make.bat html
```

Build HTML + PDF on Windows:

```
cd docs
make.bat html
make.bat latexpdf
```

1.2.8 Uninstall

If you need to add/change packages, deactivate the environment first:

```
conda deactivate
```

Then remove the appropriate environment:

```
conda remove -n laytracer --all
```

1.3 Package Design

LayTracer is structured as a Python package with a layered, modular architecture designed for numerical reliability, API clarity, and maintainability.

1.3.1 High-level architecture

The implementation is separated into focused modules with clear responsibilities:

- `laytracer.model`
  - Defines model-level data containers (for example, `LayerStack`).
  - Converts tabular velocity/depth input into solver-ready layer stacks.
- `laytracer.solver`
  - Implements the core two-point ray tracing algorithm (dimensionless-parameter Newton solver).
  - Computes per-ray kinematics such as travel time and ray geometry.
- `laytracer.amplitude`
  - Implements amplitude-related physics (transmission/reflection coefficients, Brewster-angle analysis).
  - Provides reusable low-level functions independent of plotting/UI concerns.
- `laytracer.api`
  - Exposes high-level user workflows (for example, multi source-receiver tracing via `trace_rays`).
  - Handles batching, parallel execution, and result aggregation into user-facing containers.
- `laytracer.plot`
  - Provides visualization utilities (2-D profiles and 3-D interactive rendering).
  - Keeps visualization optional and decoupled from numerical core logic.

1.3.2 Public API surface

The package root (`laytracer.__init__`) re-exports the primary classes/functions so typical user workflows stay concise while internal module boundaries remain explicit.

1.3.3 Design principles

LayTracer follows several engineering principles:

- **Separation of concerns**: numerical solvers, physical coefficients, API orchestration, and plotting are isolated.
- **Composability**: low-level building blocks can be used independently in custom workflows.

- **Performance-aware implementation**: vectorized NumPy operations and optional parallel execution for survey-scale runs.
- **Reproducibility**: environment-driven dependency management and deterministic, test-backed numerical behavior.
- **Extensibility**: new physical attributes or workflow wrappers can be added without rewriting the solver core.

1.3.4 Typical execution flow

1. Build a layered model from input data (`build_layer_stack` or high-level API input).
2. Solve one or many source-receiver ray paths (`solve` / `trace_rays`).
3. Optionally compute attenuation/spreading/transmission attributes.
4. Visualize outputs using `laytracer.plot` helpers.

CHAPTER

TWO

METHODOLOGY

This chapter presents the theoretical foundations of the ray tracing algorithm implemented in LayTracer.

2.1 Two-point ray tracing problem

Given a horizontally layered velocity model with N constant-velocity layers and two points—a source S at (x_s, z_s) and a receiver R at (x_r, z_r)—the task is to find the ray path connecting them that satisfies **Snell's law** at every interface:

$$p = \frac{\sin\theta_k}{v_k} = \text{const}$$

where p is the **horizontal slowness** (ray parameter) and θ_k is the ray angle from vertical in layer k.

2.2 Dimensionless ray parameter

Fang and Chen [2019] introduce a dimensionless parameter

$$q = \sqrt{\frac{p^2}{1/v_{\text{max}}^2 - p^2}} = \frac{p\, v_{\text{max}}}{\sqrt{1 - p^2\, v_{\text{max}}^2}}$$

where $v_{\text{max}} = \max_k v_k$. The inverse relation is

$$p = \frac{q}{v_{\text{max}}\sqrt{1+q^2}}$$

Key advantages:

- $q \in [0, \infty)$ maps the full range of valid take-off angles
- The offset function $X(q)$ is well-behaved (monotonically increasing, smooth, avoids singularities near the critical angle)

2.3 Offset equation

The total horizontal distance (offset) X traversed by a ray through N layers is (Eq. 3 of Fang and Chen [2019]):

$$X(q) = \sum_{k=1}^{N} \frac{q\,\lambda_k\, h_k}{\sqrt{1 + (1-\lambda_k^2)\, q^2}}$$

where $\lambda_k = v_k / v_{\text{max}}$ and h_k is the thickness of the k-th traversed layer.

2.3.1 First derivative

$$\frac{\mathrm{d}X}{\mathrm{d}q} = \sum_{k=1}^{N} \frac{\lambda_k \, h_k}{\left[1 + (1 - \lambda_k^2)\, q^2\right]^{3/2}}$$

2.3.2 Second derivative

$$\frac{\mathrm{d}^2X}{\mathrm{d}q^2} = -3\, q \sum_{k=1}^{N} \frac{(1 - \lambda_k^2)\, \lambda_k \, h_k}{\left[1 + (1 - \lambda_k^2)\, q^2\right]^{5/2}}$$

2.4 Asymptotic initial estimate

Two linear asymptotes of $X(q)$ provide an efficient initial guess (see "Initial estimate of q" in Fang and Chen [2019]):

Near-field ($q \to 0$):

$$X \approx m_0 \, q, \qquad m_0 = \sum_k \lambda_k \, h_k$$

Far-field ($q \to \infty$):

$$X \approx m_\infty \, q + b_\infty$$

where

$$m_\infty = \sum_{k:\, \lambda_k = 1} h_k, \qquad b_\infty = \sum_{k:\, \lambda_k < 1} \frac{\lambda_k \, h_k}{\sqrt{1 - \lambda_k^2}}$$

The initial estimate q_0 is chosen from the appropriate asymptote based on whether the target offset X_R falls in the near-field or far-field regime.

2.5 Quadratic Newton iteration

The two-point problem $X(q) = X_R$ is solved by second-order Newton iteration (see Fang and Chen [2019]). At each step, $X(q)$ is expanded to second order about the current iterate q_i:

$$\tfrac{1}{2}\, X''(q_i)\, \Delta q^2 + X'(q_i)\, \Delta q + \left[X(q_i) - X_R\right] = 0$$

This quadratic equation in Δq yields two roots. The root minimising $|X(q_i + \Delta q) - X_R|$ is selected. Convergence is typically achieved within **2–3 iterations**.

2.6 Travel time

Once the ray parameter p is determined, the travel time through each layer is

$$\Delta t_k = \frac{h_k}{v_k^2 \, \eta_k}$$

where $\eta_k = \sqrt{1/v_k^2 - p^2}$ is the vertical slowness in layer k. The total travel time is $t = \sum_k \Delta t_k$.

2.7 Attenuation operator t^*

The attenuation operator (Aki and Richards [2002], Ch. 5) measures the cumulative dissipative loss of wave amplitude along the ray path. Since the spatial path length in layer k is $\Delta s_k = v_k \Delta t_k$, the spatial integral representing intrinsic absorption corresponds exactly to:

$$t^* = \sum_{k=1}^{N} \frac{\Delta t_k}{Q_k}$$

where Q_k is the quality factor in layer k. This gives the spectral decay $\exp(-\pi f\, t^*)$ at frequency f.

For a **vertical ray**: $t^* = \sum h_k/(v_k\, Q_k)$.

For **uniform** Q: $t^* = t/Q$.

2.8 Geometrical spreading

In a 1-D layered medium with cylindrical symmetry (3-D point source), the classical geometrical spreading factor L relates the solid angle of the ray tube at the source to its cross-sectional area at the receiver (Červený [2001], Aki and Richards [2002]):

$$L = \sqrt{\frac{X \cdot \cos\theta_s \cdot \cos\theta_r}{p} \left|\frac{\partial X}{\partial p}\right|}$$

where θ_s, θ_r are the ray angles at source and receiver. Equation above defines the **relative geometrical spreading** (see Červený [2001], Eq. 4.10.22), which measures the ray-tube geometrical divergence strictly from the ray curvature, without the source-point velocity multiplier $1/v_r$.

The derivative $\partial X/\partial p$ is computed analytically via the chain rule:

$$\frac{\partial X}{\partial p} = \frac{\mathrm{d}X}{\mathrm{d}q} \cdot \frac{\mathrm{d}q}{\mathrm{d}p}, \qquad \frac{\mathrm{d}q}{\mathrm{d}p} = \frac{v_{\max}}{(1 - p^2\, v_{\max}^2)^{3/2}}$$

2.9 Reflection and transmission coefficients

In layered media, wave amplitudes are modified at every crossed interface. LayTracer computes interface coefficients using the full angle-dependent P-SV Zoeppritz formulation. Two variants are available via the `transcoef_method` parameter:

- `"standard"` — displacement-amplitude coefficients (Zoeppritz); this is the default.
- `"normalized"` — energy-flux-normalized coefficients following Červený [2001], Eq. 5.3.10.

2.9.1 Angle-dependent P-SV formulation (welded solid-solid interface)

For horizontal slowness (ray parameter) p, the vertical slownesses are

$$\eta_{\alpha i} = \sqrt{\frac{1}{v_{Pi}^2} - p^2}, \qquad \eta_{\beta i} = \sqrt{\frac{1}{v_{Si}^2} - p^2},$$

and the auxiliary quantities

$$a = \rho_2\left(1 - 2v_{S2}^2 p^2\right) - \rho_1\left(1 - 2v_{S1}^2 p^2\right),$$
$$b = \rho_2\left(1 - 2v_{S2}^2 p^2\right) + 2\rho_1 v_{S1}^2 p^2,$$
$$c = \rho_1\left(1 - 2v_{S1}^2 p^2\right) + 2\rho_2 v_{S2}^2 p^2,$$
$$d = 2\left(\rho_2 v_{S2}^2 - \rho_1 v_{S1}^2\right).$$

Define the cosine-dependent intermediate terms

$$E = b\,\eta_{\alpha 1} + c\,\eta_{\alpha 2}, \quad F = b\,\eta_{\beta 1} + c\,\eta_{\beta 2}, \quad G = a - d\,\eta_{\alpha 1}\eta_{\beta 2}, \quad H = a - d\,\eta_{\alpha 2}\eta_{\beta 1},$$

and the system determinant

$$D = EF + GH\,p^2.$$

The complete 4×4 scattering matrix (Aki and Richards [2002], Eqs. 5.38–5.40) is computed by LayTracer. The eight independent P-SV coefficients are listed below.

Incident P-wave — reflection and transmission:

$$R_{PP} = \frac{(b\,\eta_{\alpha 1} - c\,\eta_{\alpha 2})\,F - (a + d\,\eta_{\alpha 1}\eta_{\beta 2})\,H\,p^2}{D},$$
$$R_{PS} = -\frac{2\,\eta_{\alpha 1}\,(ab + cd\,\eta_{\alpha 2}\eta_{\beta 2})\,p\,(v_{P1}/v_{S1})}{D},$$
$$T_{PP} = \frac{2\rho_1\,\eta_{\alpha 1}\,F\,(v_{P1}/v_{P2})}{D},$$
$$T_{PS} = \frac{2\rho_1\,\eta_{\alpha 1}\,H\,p\,(v_{P1}/v_{S2})}{D}.$$

Incident SV-wave — reflection and transmission:

$$R_{SP} = -\frac{2\,\eta_{\beta 1}\,(ab + cd\,\eta_{\alpha 2}\eta_{\beta 2})\,p\,(v_{S1}/v_{P1})}{D},$$
$$R_{SS} = -\frac{(b\,\eta_{\beta 1} - c\,\eta_{\beta 2})\,E - (a + d\,\eta_{\alpha 2}\eta_{\beta 1})\,G\,p^2}{D},$$
$$T_{SP} = -\frac{2\rho_1\,\eta_{\beta 1}\,G\,p\,(v_{S1}/v_{P2})}{D},$$
$$T_{SS} = \frac{2\rho_1\,\eta_{\beta 1}\,E\,(v_{S1}/v_{S2})}{D}.$$

For references and details on the derivation of these formulas, see Lay and Wallace [1995] (Table 3.1, note the sign error in the second term of b) and Aki and Richards [2002] (Equations 5.38–5.40).

For post-critical incidence the coefficients may become complex; for amplitude modelling the software uses $|T_l|$.

2.9.2 Energy-flux-normalized coefficients

The standard (displacement-amplitude) Zoeppritz coefficients $\bar{R}_{mn}$ do **not** conserve energy flux across an interface. For applications where energy conservation is essential (e.g. seismic-moment estimation in layered media), Červený [2001] (Eq. 5.3.10) defines **normalized** reflection/transmission coefficients:

$$\mathcal{R}_{mn} = \bar{R}_{mn}\sqrt{\frac{v_{\text{out}}\,\rho_{\text{out}}\,\cos\theta_{\text{out}}}{v_{\text{in}}\,\rho_{\text{in}}\,\cos\theta_{\text{in}}}},$$

where $\cos\theta = \sqrt{1 - v^2 p^2}$ and subscripts "in"/"out" refer to the incident and scattered wave, respectively (for reflections, "out" uses properties of the same side of the interface). For subcritical incidence at a lossless interface, the normalized coefficients ensure conservation of energy flux, with reflected and transmitted energy fractions summing to unity.

2.9.3 Critical angles

A **critical angle** occurs when the transmitted wave in a faster medium becomes evanescent. For an incident P-wave crossing into a layer where $v_{P2} > v_{P1}$, the P-critical angle is

$$\theta_c^P = \arcsin\left(\frac{v_{P1}}{v_{P2}}\right).$$

Similarly, when $v_{S2} > v_{P1}$ an S-to-P critical angle exists:

$$\theta_c^S = \arcsin\left(\frac{v_{P1}}{v_{S2}}\right).$$

For an incident SV-wave the same logic applies with v_{S1} in the numerator. Beyond the critical angle the corresponding vertical slowness becomes imaginary, the coefficient becomes complex, and total reflection occurs for that mode.

2.9.4 Brewster angles

By analogy with optics, a **Brewster angle** is an incidence angle at which a reflection or transmission coefficient passes through zero or a deep minimum. In electromagnetic theory, Brewster's angle is the incidence angle at which $R_p = 0$ for p-polarised light at a dielectric interface. In elastodynamics, the same phenomenon occurs: certain combinations of elastic parameters produce incidence angles where one of the P-SV scattering coefficients vanishes.

Unlike critical angles, which depend only on velocity ratios, Brewster angles depend on *all six* elastic parameters (v_{P1}, v_{S1}, ρ_1, v_{P2}, v_{S2}, ρ_2). Physically, they arise from destructive interference between the P and SV displacement potentials at the welded interface: the two potential contributions to a particular scattered mode cancel exactly, driving that coefficient to zero.

For example, the reflected P coefficient R_{PP} may vanish at an angle well below the critical angle. At this Brewster angle the incident energy is partitioned entirely into the transmitted P-wave and the mode-converted waves, with no same-mode reflection.

LayTracer provides the function `find_brewster_angles` which numerically detects these minima in the computed coefficient curves by searching for local minima of $|C(\theta)|$ whose value falls below a user-specified threshold.

2.10 Extension to 3-D layered media

All the formulae above are derived in a **2-D vertical plane** containing both the source and the receiver. Because horizontally layered media possess *cylindrical symmetry* about the vertical axis through the source, any source–receiver pair in three-dimensional space can be **reduced to an equivalent 2-D problem** without loss of generality (Červený [2001], Ch. 3; Aki and Richards [2002], Ch. 4).

2.10.1 Coordinate projection

Let the source position be $\mathbf{s} = (x_s, y_s, z_s)$ and the receiver position $\mathbf{r} = (x_r, y_r, z_r)$. The **epicentral distance** (horizontal distance) is

$$\Delta = \sqrt{(x_r - x_s)^2 + (y_r - y_s)^2}$$

and the **unit direction vector** from source to receiver in the horizontal plane is

$$\hat{\mathbf{u}} = \frac{1}{\Delta}\begin{pmatrix} x_r - x_s \\ y_r - y_s \end{pmatrix}, \qquad \Delta > 0$$

(for vanishing Δ the azimuth is arbitrary, and we default to $\hat{\mathbf{u}} = (1,\, 0)$).

The 2-D ray tracing problem is then solved exactly as described in the preceding sections, with the epicentral distance Δ playing the role of the target offset X_R and the depth coordinates z_s, z_r determining the traversed layers.

2.10.2 Back-projection to 3-D

Once the 2-D ray path $\{(x_k^{(2D)},\ z_k)\}_{k=0}^{M}$ has been computed, it is mapped back to 3-D Cartesian coordinates via

$$\begin{aligned} X_k &= x_s + x_k^{(2D)}\ \hat{u}_x, \\ Y_k &= y_s + x_k^{(2D)}\ \hat{u}_y, \\ Z_k &= z_k. \end{aligned}$$

This simply *sweeps* the 2-D ray along the source–receiver azimuth.

2.10.3 Validity of amplitude attributes

Because the medium properties depend only on depth, every quantity computed from the 2-D ray remains valid in 3-D:

- **Ray parameter** p — determined solely by Snell's law across horizontal interfaces.
- **Travel time** — sum of vertical-slowness contributions Δt_k, unchanged by azimuth.
- **Attenuation operator** t^* — depends only on Δt_k and Q_k.
- **Geometrical spreading** — the formula incorporates the epicentral distance X to capture the 3-D cylindrical divergence of the ray-tube out of the incidence plane (Červený [2001], §4.10).
- **Transmission coefficients** — depend on ray parameter and layer impedances, not on azimuth.

Thus, the layered-media solver need only operate in the 2-D ray plane; the full 3-D solution is recovered by geometry alone.

CHAPTER

THREE

EXAMPLES

Gallery of examples demonstrating the LayTracer package capabilities, including ray tracing, reflection & transmission coefficients, geometric spreading, attenuation dependencies, and visualisation of rays in layered media.

3.1 01. Basic ray tracing

This example demonstrates two-point ray tracing through a simple 3-layer velocity model using LayTracer in 2D and 3D cases.

3.1.1 Setup

```
import laytracer as lt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
```

3.1.2 Define velocity model

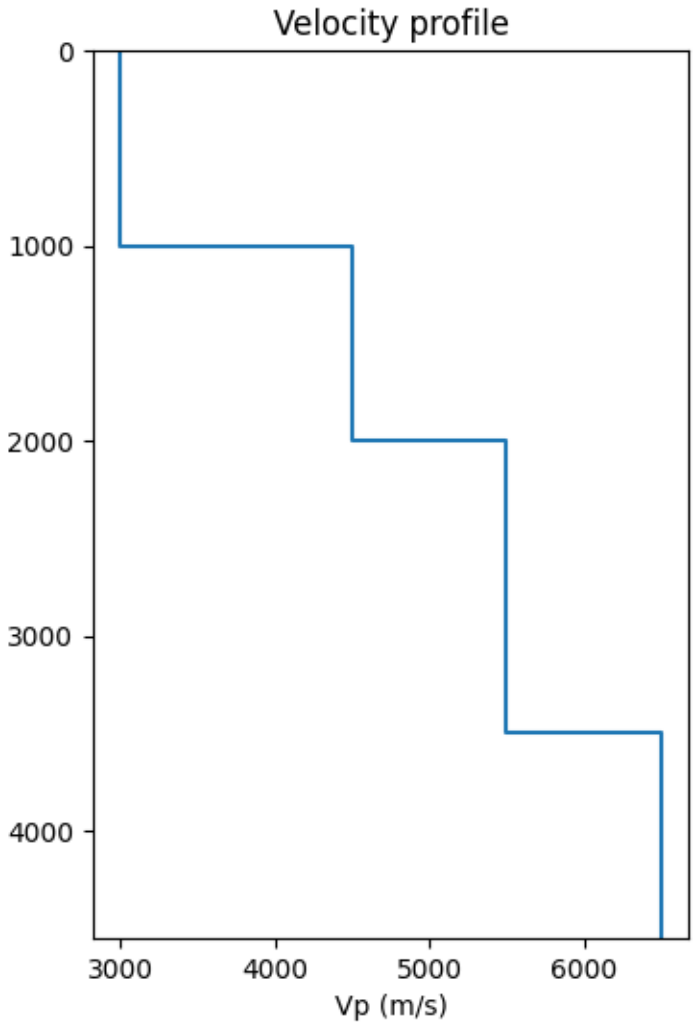
A 3-layer model with increasing velocity with depth.

```
vel_df = pd.DataFrame({
    "Depth": [0.0, 1000.0, 2000.0, 3500.0],
    "Vp":    [3000.0, 4500.0, 5500.0, 6500.0],
    "Vs":    [1500.0, 2250.0, 2750.0, 3250.0],
    "Rho":   [2200.0, 2500.0, 2700.0, 2900.0],
    "Qp":    [200.0,  400.0,  600.0,  800.0],
    "Qs":    [100.0,  200.0,  300.0,  400.0],
})

print(vel_df)
```

```
    Depth      Vp      Vs     Rho     Qp     Qs
0     0.0  3000.0  1500.0  2200.0  200.0  100.0
1  1000.0  4500.0  2250.0  2500.0  400.0  200.0
2  2000.0  5500.0  2750.0  2700.0  600.0  300.0
3  3500.0  6500.0  3250.0  2900.0  800.0  400.0
```

3.1.3 Plot velocity profile

```
ax = lt.plot.velocity_profile(vel_df, param="Vp")
plt.show()
```



3.1.4 Trace a single ray in 2-D

Trace a P-wave from a source at depth 3000 m to a receiver at the surface.

```
stack = lt.build_layer_stack(vel_df, z_src=3000.0, z_rcv=0.0)

# Use trace_rays for 2D tracing as well
res = lt.trace_rays(
    sources=[0.0, 0.0, 3000.0],
    receivers=[5000.0, 0.0, 0.0],
    velocity_df=vel_df,
```

(continues on next page)

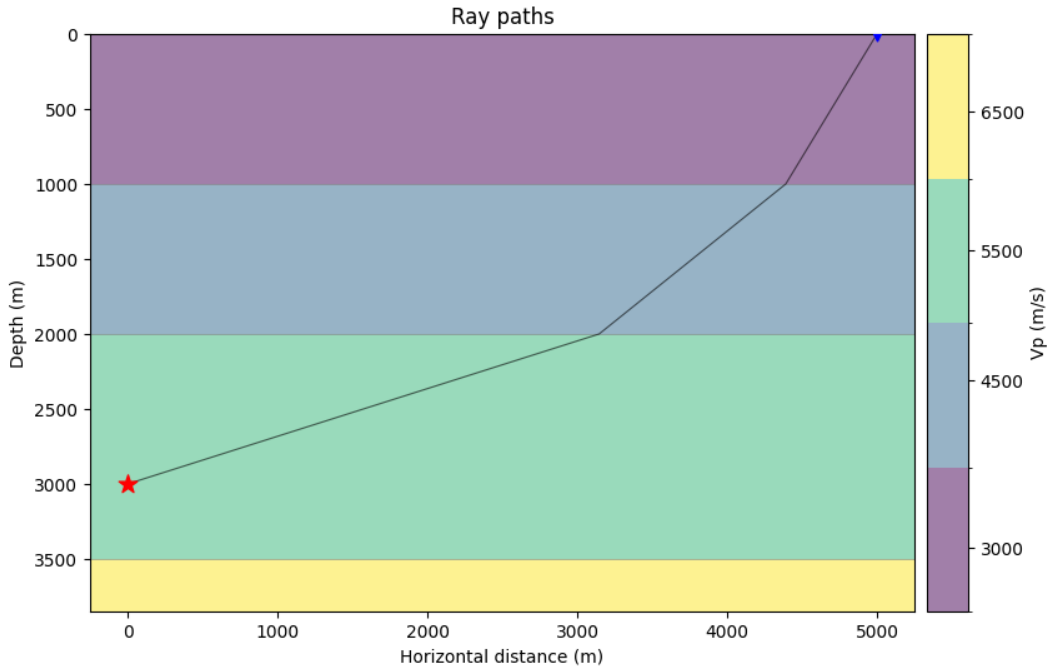
(continued from previous page)

```
    source_phase="P",
)

print(f"Travel time:    {res.travel_times[0]:.4f} s")
print(f"Ray parameter:  {res.ray_parameters[0]:.6e} s/m")
```

```
Travel time:    1.3453 s
Ray parameter:  1.732757e-04 s/m
```

3.1.5 Plot the 2-D ray

```
ax = lt.plot.rays_2d(
    vel_df,
    rays=res.rays,
    sources=np.array([[0.0, 0.0, 3000.0]]),
    receivers=np.array([[5000.0, 0.0, 0.0]]),
    vel_type="Vp",
    add_colorbar=True,
    model_alpha=0.5,
    discrete_colorbar=True,
)
plt.show()
```



3.1.6 Trace multiple rays in 3-D

Use `lt.trace_rays` to trace from one source to multiple receivers arranged in a circle.

```
src = np.array([0.0, 0.0, 3000.0])

# Receivers on surface in a circle of radius 5000 m
n_rcv = 12
angles = np.linspace(0, 2 * np.pi, n_rcv, endpoint=False)
rcvs = np.column_stack([
    5000.0 * np.cos(angles),
    5000.0 * np.sin(angles),
    np.zeros(n_rcv),
])

result = lt.trace_rays(
    sources=src,
    receivers=rcvs,
    velocity_df=vel_df,
    source_phase="P",
)

print(f"Number of rays: {len(result.rays)}")
print(f"Travel times:   {result.travel_times}")
```

```
Number of rays: 12
Travel times:   [1.34534574 1.34534574 1.34534574 1.34534574 1.34534574 1.34534574
 1.34534574 1.34534574 1.34534574 1.34534574 1.34534574 1.34534574]
```

3.1.7 Plot 3-D rays

```
fig = lt.plot.rays_3d(
    vel_df,
    rays=result.rays,
    sources=src,
    receivers=rcvs,
)
fig
plt.show()
```

Total running time of the script: (0 minutes 0.261 seconds)

3.2 02. Paper examples

Reproducing three characteristic figures from paper:

Fang, X., & Chen, X. (2019). A fast and robust two-point ray tracing method in layered media with constant or linearly varying layer velocity. Geophysical Prospecting, 67(7), 1811–1824. https://doi.org/10.1111/1365-2478.12799 [Fang and Chen, 2019].

3.2.1 Setup

```
import laytracer as lt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = -1
```

3.2.2 Reproduce Figure 9

Here we reproduce Figure 9 from Fang and Chen [2019].

Figure 9 illustrates the ray paths for reflected waves in a five-layered model with mixed constant and gradient velocity layers. The model consists of:

- Layer 1 (0-100m): Gradient velocity $v = 4z + 1800$
- Layer 2 (100-200m): Constant velocity $v = 2400$
- Layer 3 (200-300m): Gradient velocity $v = z + 2400$
- Layer 4 (300-400m): Constant velocity $v = 2700$
- Layer 5 (400-500m): Gradient velocity $v = 1.5z + 2250$

The figure demonstrates the method's capability to handle complex models with both constant and gradient layers. It shows rays with reflection angles of 1°, 30°, 50°, and (in the paper) 89° at the bottom interface (500m depth).

```
print("Reproducing Figure 9...")

layers = []
# Layer 1 (Gradient)
layers.append(lt.model.discretize_gradient_layer(0, 100, lambda z: 4*z + 1800))
# Layer 2 (Constant)
layers.append(pd.DataFrame({"Depth": [100.0], "Vp": [2400.0], "Vs": [2400.0/1.732],
↪"Rho": [2500.0]}))
# Layer 3 (Gradient)
layers.append(lt.model.discretize_gradient_layer(200, 300, lambda z: z + 2400))
# Layer 4 (Constant)
layers.append(pd.DataFrame({"Depth": [300.0], "Vp": [2700.0], "Vs": [2700.0/1.732],
↪"Rho": [2500.0]}))
# Layer 5 (Gradient) - We need this to go down to 500m
layers.append(lt.model.discretize_gradient_layer(400, 500, lambda z: 1.5*z + 2250))

# Add dummy half-space at 500m so it's a valid interface
layers.append(pd.DataFrame({"Depth": [500.0], "Vp": [2000.0], "Vs": [2000.0/1.732],
↪"Rho": [2500.0]}))

vel_df = pd.concat(layers, ignore_index=True)

src = np.array([0.0, 0.0, 0.0])

# Target reflection angles at bottom (500m)
# Reduced set to avoid numerical instability with grazing rays in discretized model
angles_deg = np.array([1, 30, 50, 85, 89])
```

(continues on next page)

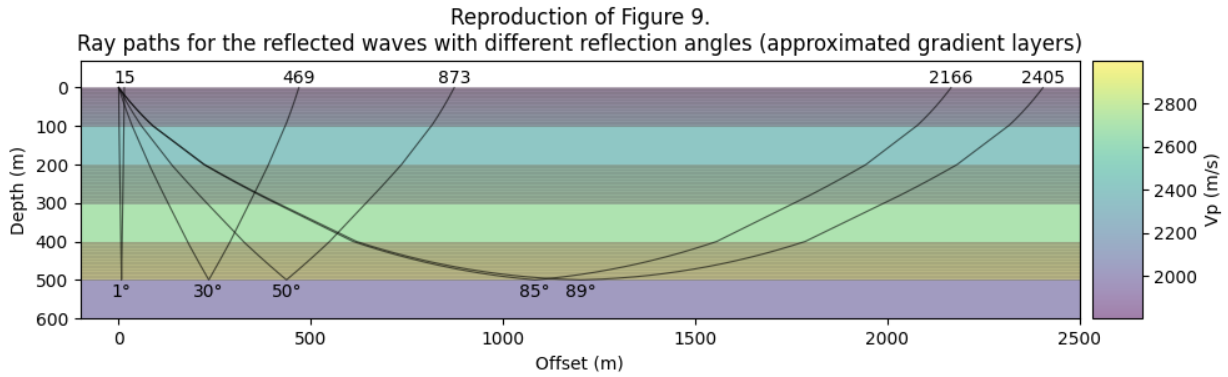
(continued from previous page)

```
v_ref = 3000.0
p_targets = np.sin(np.deg2rad(angles_deg)) / v_ref

# Calculate target offsets
stack = lt.build_layer_stack(vel_df, 0.0, 500.0)
h = stack.h
v = stack.vp
vmax = np.max(v)
lmd = v / vmax
q_vals = lt.solver.q_from_p(p_targets, vmax)

offsets_half = []
for q in q_vals:
    x = lt.solver.offset(q, h, lmd)
    offsets_half.append(x)

offsets_total = np.array(offsets_half) * 2.0

receivers = np.zeros((len(offsets_total), 3))
receivers[:, 0] = offsets_total

print(f"Tracing rays for {len(receivers)} receivers...")
try:
    results = lt.trace_rays(
        sources=src,
        receivers=receivers,
        velocity_df=vel_df,
        source_phase="P",
        reflection=[(500.0, "P")]
    )
    print("Figure 9 Tracing complete.")
except Exception as e:
    print(f"Error tracing rays in Fig 9: {e}")
    raise e

fig, ax = plt.subplots(figsize=(10, 5))
lt.plot.rays_2d(
    vel_df=vel_df,
    rays=results.rays,
    vel_type="Vp",
    ax=ax,
    xlim=(-100, 2500),
    ylim=(600, -70),
    plot_model=True,
    add_colorbar=True,
    model_alpha=0.5
)
for i, x in enumerate(offsets_total):
    ax.text(x, 0, f"{offsets_total[i]:.0f}", ha='center', va='bottom')
    ax.text(x/2, 510, f"{angles_deg[i]}°", ha='center', va='top')

ax.set_title("Reproduction of Figure 9.\nRay paths for the reflected waves with␣
```

(continues on next page)

(continued from previous page)

```
↪different reflection angles (approximated gradient layers)")
ax.set_xlabel("Offset (m)")
ax.set_ylabel("Depth (m)")
fig.tight_layout()
plt.show()
```




```
Reproducing Figure 9...
Tracing rays for 5 receivers...
Figure 9 Tracing complete.
```

3.2.3 Reproduce Figure 10

Here we reproduce Figure 10 from Fang and Chen [2019].

Figure 10 presents a random realization of a 10-layered model used for Monte Carlo simulations to test robustness. In the paper's experimental setup:

- The model contains 10 layers with random thicknesses (uniform distribution corresponding to ~18-189 m).
- Layer velocities vary randomly between 1500 and 3000 m/s.
- The source is located at 10 m depth.
- Rays are traced to receivers at offsets of 500, 1000, and 2000 m.

This setup tests the q-method's stability against random velocity fluctuations and layer thickness variations.

```
print("Reproducing Figure 10...")
np.random.seed(42)
n_layers = 10
total_depth = 1000.0
h_raw = np.random.uniform(18, 189, n_layers)
h_vals = h_raw * (total_depth / np.sum(h_raw))
```

(continues on next page)

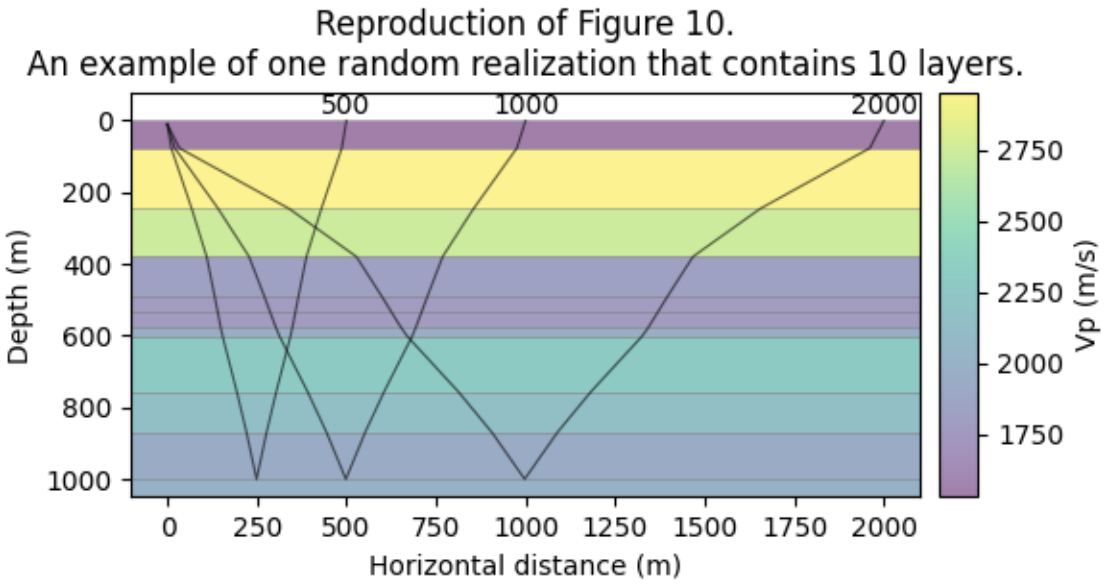
(continued from previous page)

```
depths_top = np.concatenate(([0], np.cumsum(h_vals[:-1])))
v_vals = np.random.uniform(1500, 3000, n_layers)

df_data = {
    "Depth": depths_top,
    "Vp": v_vals,
    "Vs": v_vals / 1.732,
    "Rho": 2500.0
}
vel_df = pd.DataFrame(df_data)

# Add dummy half-space
dummy_row = pd.DataFrame({
    "Depth": [1000.0],
    "Vp": [2000.0],
    "Vs": [2000.0/1.732],
    "Rho": [2500.0]
})
vel_df = pd.concat([vel_df, dummy_row], ignore_index=True)

src = np.array([0.0, 0.0, 10.0])
targets = np.array([500.0, 1000.0, 2000.0])
receivers = np.zeros((len(targets), 3))
receivers[:, 0] = targets

try:
    results = lt.trace_rays(
        sources=src,
        receivers=receivers,
        velocity_df=vel_df,
        source_phase="P",
        reflection=[(1000.0, "P")]
    )
    print("Figure 10 Tracing complete.")
except Exception as e:
    print(f"Error tracing rays in Fig 10: {e}")
    raise e

fig, ax = plt.subplots(figsize=(6, 8))
lt.plot.rays_2d(
    vel_df=vel_df,
    rays=results.rays,
    vel_type="Vp",
    ax=ax,
    ylim=(1050, -80),
    plot_model=True,
    add_colorbar=True,
    model_alpha=0.5
)
for i, x in enumerate(targets):
    ax.text(x, 0, f"{targets[i]:.0f}", ha='center', va='bottom')
```

(continues on next page)

(continued from previous page)

```
ax.set_title("Reproduction of Figure 10.\nAn example of one random realization that␣
↪contains 10 layers.")
fig.tight_layout()
plt.show()
```




```
Reproducing Figure 10...
Figure 10 Tracing complete.
```

3.2.4 Reproduce Figure 15c

Here we reproduce Figure 15c from Fang and Chen [2019].

Figure 15c compares the q-method with the method of Kim and Baag (2002) using "Model I" from their paper. This serves as a crustal model benchmark with six constant-velocity layers:

- Layer 1 (0-5 km): 5.5 km/s
- Layer 2 (5-10 km): 5.8 km/s
- Layer 3 (10-15 km): 6.2 km/s
- Layer 4 (15-22 km): 6.6 km/s
- Layer 5 (22-32 km): 7.2 km/s
- Layer 6 (32-42 km): 7.9 km/s
- Layer 7 (>42 km): 8.0 km/s

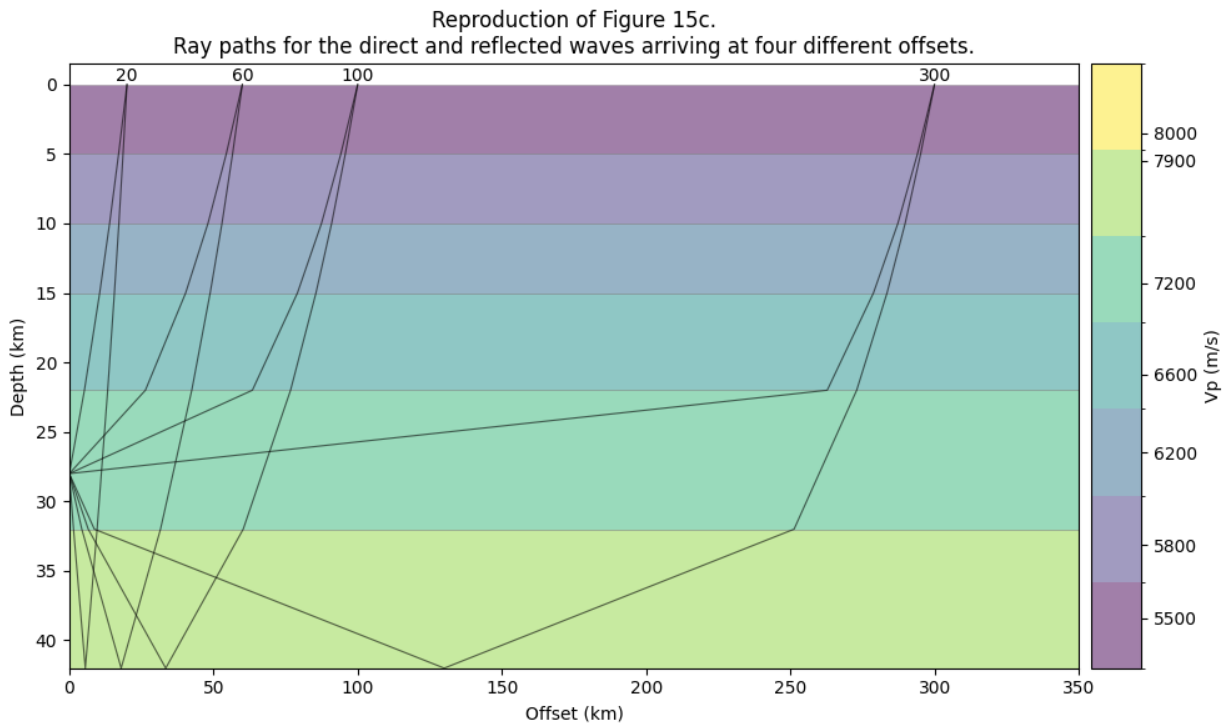
The figure displays ray paths for both direct and reflected waves arriving at offsets of 20, 60, 100, and 300 km.

```
print("Reproducing Figure 15c...")
depths = np.array([0.0, 5.0, 10.0, 15.0, 22.0, 32.0, 42.0]) * 1000.0
vp = np.array([5.5, 5.8, 6.2, 6.6, 7.2, 7.9, 8.0]) * 1000.0

vel_df = pd.DataFrame({
    "Depth": depths,
    "Vp": vp,
    "Vs": vp / 1.732,
    "Rho": 2500.0
})

offsets_km = np.array([20, 60, 100, 300])
offsets_m = offsets_km * 1000.0
src = np.array([0.0, 0.0, 28000.0])
receivers = np.zeros((len(offsets_m), 3))
receivers[:, 0] = offsets_m

try:
    res_refl = lt.trace_rays(
        sources=src,
        receivers=receivers,
        velocity_df=vel_df,
        source_phase="P",
        reflection=[(42000.0, "P")]
    )
    res_refr = lt.trace_rays(
        sources=src,
        receivers=receivers,
        velocity_df=vel_df,
        source_phase="P"
    )

    print("Figure 15c Tracing complete.")
except Exception as e:
    print(f"Error tracing rays in Fig 15c: {e}")
```

(continues on next page)

(continued from previous page)

```
    raise e

# Concatenate rays from both results
res_rays = res_refl.rays + res_refr.rays

fig, ax = plt.subplots(figsize=(10, 6))
lt.plot.rays_2d(
    vel_df=vel_df,
    rays=res_rays,
    vel_type="Vp",
    ax=ax,
    xlim=(0, 350),
    ylim=(42, -1.5),
    plot_model=True,
    equal_scale=False,
    add_colorbar=True,
    discrete_colorbar=True,
    model_alpha=0.5,
    unit="km"
)
for i, x in enumerate(offsets_km):
    ax.text(x, 0, f"{offsets_km[i]:.0f}", ha='center', va='bottom')

ax.set_title("Reproduction of Figure 15c.\nRay paths for the direct and reflected␣
↪waves arriving at four different offsets.")
ax.set_xlabel("Offset (km)")
ax.set_ylabel("Depth (km)")
fig.tight_layout()
plt.show()
```




```
Reproducing Figure 15c...
Figure 15c Tracing complete.
```

Total running time of the script: (0 minutes 0.575 seconds)

3.3 03. Reflection & transmission

Reproduction of the classic P-SV reflection & transmission test case from Charles J. Ammon's MATLAB Exercise L3 (PDF) (Lay and Wallace [1995], Figure 3.28).

For an incident P-wave the system unknowns are $[R_{PP},\ R_{PS},\ T_{PP},\ T_{PS}]$. For an incident SV-wave the unknowns are $[R_{SP},\ R_{SS},\ T_{SP},\ T_{SS}]$.

3.3.1 Setup

```
import laytracer as lt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
```

3.3.2 Model

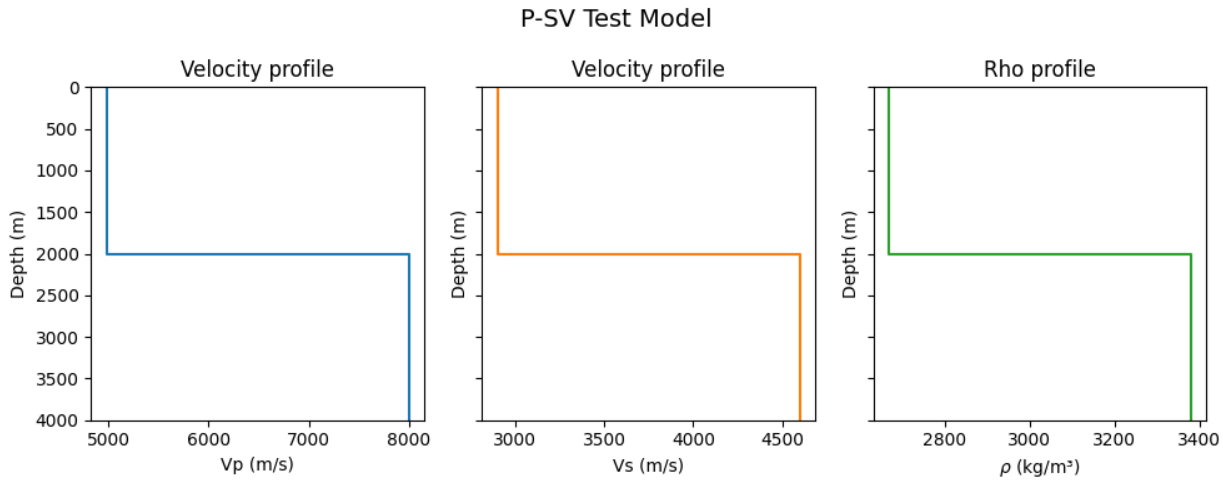
Medium parameters (Km/s and g/cm^3)

```
mi_vp, mi_vs, mi_rho = 4.98, 2.9, 2.667    # incident
mt_vp, mt_vs, mt_rho = 8.00, 4.6, 3.38     # transmitted
```

(continues on next page)

(continued from previous page)

```
# Create a DataFrame for visualization (using SI units m/s, kg/m^3)
model_psv = pd.DataFrame({
    "Depth": [0.0, 2000.0],  # Arbitrary interface depth at 2km
    "Vp":    [mi_vp * 1000, mt_vp * 1000],
    "Vs":    [mi_vs * 1000, mt_vs * 1000],
    "Rho":   [mi_rho * 1000, mt_rho * 1000],
})

# Plot the velocity model
fig, axes = plt.subplots(1, 3, figsize=(10, 4), sharey=True)
lt.plot.velocity_profile(model_psv, param="Vp", ax=axes[0], ylim=(4000, 0))
lt.plot.velocity_profile(model_psv, param="Vs", ax=axes[1], color="tab:orange",␣
↪ylim=(4000, 0))
lt.plot.velocity_profile(model_psv, param="Rho", ax=axes[2], color="tab:green",␣
↪ylim=(4000, 0))

fig.suptitle("P-SV Test Model", fontsize=14)
fig.tight_layout()
plt.show()

# Ray-parameter sweep: p from 0 to 1/Vp_incident
n_p = 200
p_vec = np.linspace(0, 1.0 / mi_vp, n_p + 1)

# Compute all 8 R/T coefficients
RT = lt.psv_rt_coefficients(
    p=p_vec,
    vp1=mi_vp, vs1=mi_vs, rho1=mi_rho,
    vp2=mt_vp, vs2=mt_vs, rho2=mt_rho,
)
```



3.3.3 Incident P-wave coefficients

For an incident P-wave the ray parameter sweeps from 0 to $1/V_P$ (grazing P incidence), covering the full $0 - -90°$ range.

Critical angle (dashed red line):

- Transmitted P becomes evanescent at $\theta_c^{T(P)} = \arcsin(V_P^{(1)}/V_P^{(2)}) \approx 38.5°$. Beyond this angle $|R_{PP}| \to 1$ (total reflection). There is no transmitted-SV critical angle because $V_P^{(1)} > V_S^{(2)}$ for this model.

Brewster angles (dotted purple lines):

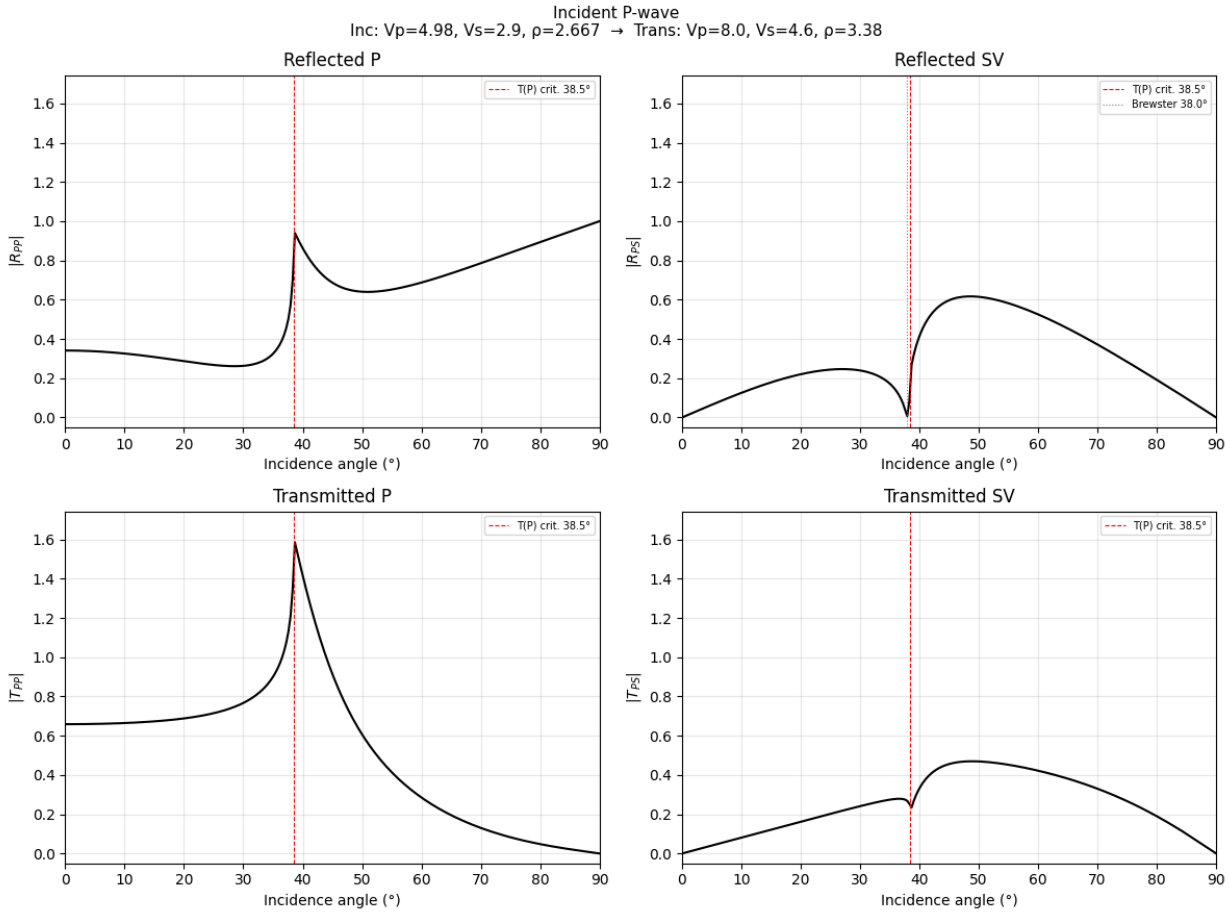
- $|R_{PS}|$ has a near-zero at $37.9°$, just before the critical angle. This is the P-to-SV mode-conversion null, analogous to the optical Brewster angle. Its position depends on all six elastic parameters, not just the velocity ratio.

```
# Incidence angle (P-wave): :math:`\theta` = \arcsin{p \cdot V_p}`
angle_P = np.rad2deg(np.arcsin(np.clip(p_vec * mi_vp, -1, 1)))
crit_P = np.rad2deg(np.arcsin(mi_vp / mt_vp))    # transmitted P critical

# Detect Brewster angles for all P-incident coefficients
brew_P = lt.find_brewster_angles(RT, angle_P, keys=["Rpp", "Rps", "Tpp", "Tps"])

# Shared y-limit across all four P-incident panels
p_keys = ["Rpp", "Rps", "Tpp", "Tps"]
ymax_P = max(np.nanmax(np.abs(RT[k])) for k in p_keys) * 1.1
ymax_P = max(ymax_P, 0.5)

fig, axes = plt.subplots(2, 2, figsize=(12, 9))
fig.suptitle(
    "Incident P-wave\n"
    f"Inc: Vp={mi_vp}, Vs={mi_vs}, ρ={mi_rho}  →  "
    f"Trans: Vp={mt_vp}, Vs={mt_vs}, ρ={mt_rho}",
    fontsize=11,
)

labels = [
    (0, 0, "Rpp", r"$|R_{PP}|$",  "Reflected P"),
    (0, 1, "Rps", r"$|R_{PS}|$",  "Reflected SV"),
    (1, 0, "Tpp", r"$|T_{PP}|$",  "Transmitted P"),
    (1, 1, "Tps", r"$|T_{PS}|$",  "Transmitted SV"),
]

for row, col, key, ylabel, title in labels:
    ax = axes[row, col]
    ax.plot(angle_P, np.abs(RT[key]), "k-", lw=1.5)
    ax.axvline(crit_P, color="r", ls="--", lw=0.8,
               label=f"T(P) crit. {crit_P:.1f}°")
    # Brewster lines for this coefficient
    for ba in brew_P.get(key, []):
        ax.axvline(ba, color="tab:purple", ls=":", lw=0.8,
                   label=f"Brewster {ba:.1f}°")
    ax.set_xlim(0, 90)
    ax.set_ylim(-0.05, ymax_P)
    ax.set_xlabel("Incidence angle (°)")
    ax.set_ylabel(ylabel)
    ax.set_title(title)
```

(continues on next page)

(continued from previous page)

```
    ax.legend(fontsize=7, loc="upper right")
    ax.grid(True, alpha=0.3)

fig.tight_layout()
plt.show()
```



3.3.4 Normalized P-wave coefficients

Energy-flux-normalized coefficients account for the impedance and directional cosine contrast across the interface. They are useful for amplitude-preserving modelling because the product of normalized transmission coefficients along a ray is the displacement-amplitude transfer factor that conserves energy flux.

The normalization follows Červený [2001] Eq. 5.3.10:

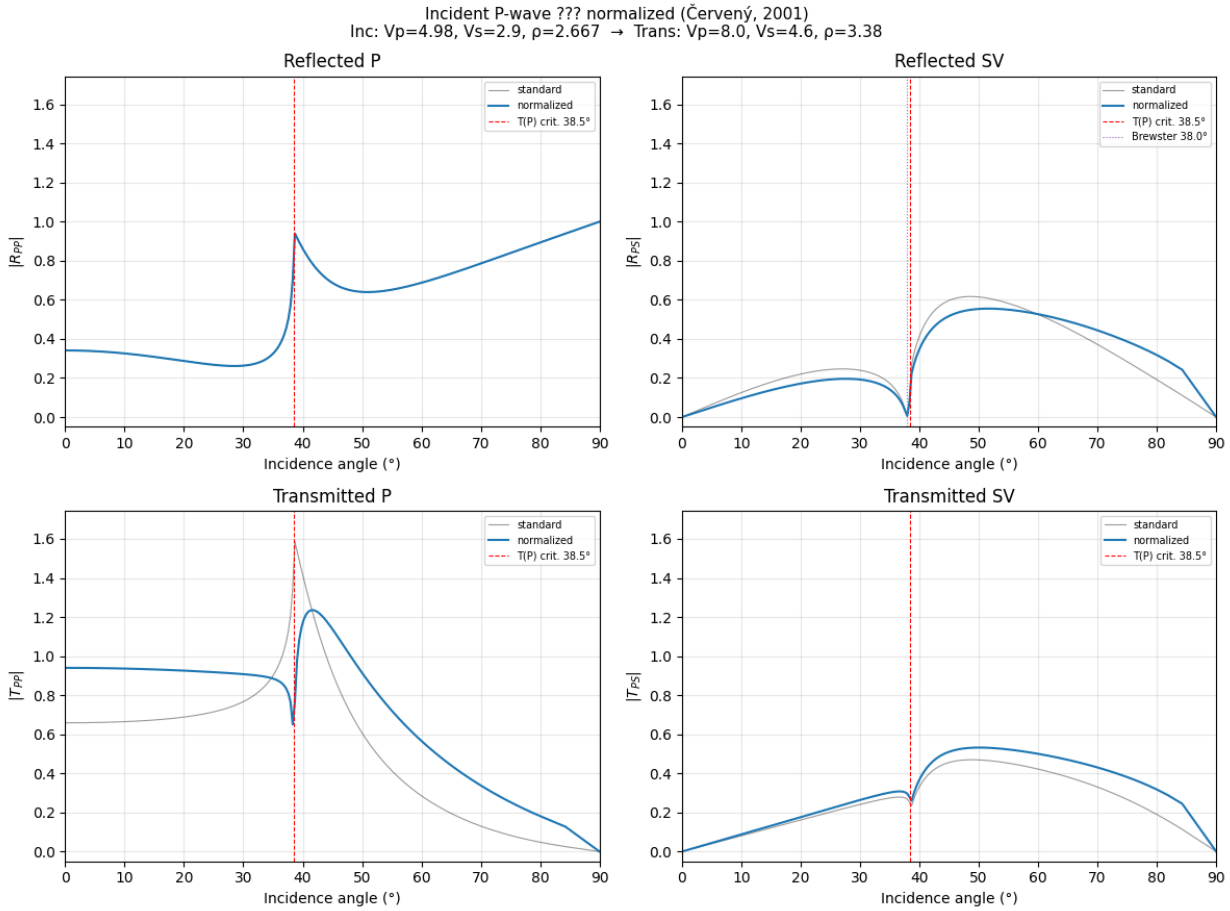
$$R_{mn}^{\text{norm}} = \bar{R}_{mn} \sqrt{\frac{V_{\text{out}}\,\rho_{\text{out}}\,\cos\theta_{\text{out}}}{V_{\text{in}}\,\rho_{\text{in}}\,\cos\theta_{\text{in}}}}$$

```
# Mapping: key -> (v_in, rho_in, v_out, rho_out)
norm_map_P = {
    "Rpp": (mi_vp, mi_rho, mi_vp, mi_rho),
    "Rps": (mi_vp, mi_rho, mi_vs, mi_rho),
    "Tpp": (mi_vp, mi_rho, mt_vp, mt_rho),
```

(continues on next page)

(continued from previous page)

```
    "Tps": (mi_vp, mi_rho, mt_vs, mt_rho),
}

RT_norm_P = {}
for key, (vi, ri, vo, ro) in norm_map_P.items():
    RT_norm_P[key] = lt.normalize_rt_coefficient(
        RT[key], p_vec, vi, ri, vo, ro,
    )

ymax_Pn = max(
    np.nanmax(np.abs(RT_norm_P[k])) for k in p_keys
) * 1.1
ymax_Pn = max(ymax_Pn, 0.5)

fig, axes = plt.subplots(2, 2, figsize=(12, 9))
fig.suptitle(
    "Incident P-wave ??? normalized (Červený, 2001)\n"
    f"Inc: Vp={mi_vp}, Vs={mi_vs}, ρ={mi_rho}  →  "
    f"Trans: Vp={mt_vp}, Vs={mt_vs}, ρ={mt_rho}",
    fontsize=11,
)

for row, col, key, ylabel, title in labels:
    ax = axes[row, col]
    ax.plot(angle_P, np.abs(RT[key]), "k-", lw=0.8, alpha=0.4, label="standard")
    ax.plot(angle_P, np.abs(RT_norm_P[key]), "tab:blue", lw=1.5, label="normalized")
    ax.axvline(crit_P, color="r", ls="--", lw=0.8,
               label=f"T(P) crit. {crit_P:.1f}°")
    for ba in brew_P.get(key, []):
        ax.axvline(ba, color="tab:purple", ls=":", lw=0.8,
                   label=f"Brewster {ba:.1f}°")
    ax.set_xlim(0, 90)
    ax.set_ylim(-0.05, max(ymax_P, ymax_Pn))
    ax.set_xlabel("Incidence angle (°)")
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.legend(fontsize=7, loc="upper right")
    ax.grid(True, alpha=0.3)

fig.tight_layout()
plt.show()
```



3.3.5 Ray diagrams (P-incidence)

We visualize the ray paths for typical situations using *lt.plot.rays_2d*. The interface is at 2000 m.

```
def plot_ray_situation(angle, wave_type, title, ax):
    # 1. Setup background (velocity model) and axes
    # We pass empty rays list first just to set up the plot environment
    lt.plot.rays_2d(
        model_psv, rays=[], ax=ax, vel_type="Vp",
        xlim=(-100, 6000), ylim=(4000, 0),
        plot_model=True,
        add_colorbar=True,
        model_alpha=0.5,
        discrete_colorbar=True,
    )

    # 2. Compute Offset for the given angle to define receiver position
    # The example wants to visualize SPECIFIC angles.
    # trace_rays solves the Two-Point problem (Fixed Receiver).
    # To plot a ray for a specific angle, we first find where it lands.
    # Or we can keep using manual shooting logic?
    # No, the goal is to demonstrate the NEW engine.
```

(continues on next page)

(continued from previous page)

```
    # We calculate geometric offset for the flat layers given angle
    v_inc = mi_vp * 1000 if wave_type == "P" else mi_vs * 1000
    p_target = np.sin(np.deg2rad(angle)) / v_inc

    # Check critical angles before tracing
    # If p > 1/V_layer, it's evanescent.
    # LayTracer solver handles non-evanescent rays.
    # We manually check evanescence for the legs we want to plot.

    source = np.array([0.0, 0.0, 0.0])
    z_int = 2000.0
    z_bot = 4000.0

    # Helper to trace and plot one ray variant
    def run_trace(rcv_z, reflection_arg=None, refraction_arg=None, label="", color="",
↪ style=""):
        # Calculate theoretical horizontal offset for this p
        # We assume simplified straight rays for this calc (constant layer blocks)

        # We need the path legs to calculate X(p_target).
        # We can use lt.offset() if we build the stack manually,
        # OR just simple trig since model is constant layers.

        # Legs depend on reflection/refraction.
        dx = 0.0

        # LEG 1: 0 -> 2000
        # Check P-wave layer 0
        v0 = mi_vp * 1000 if wave_type == "P" else mi_vs * 1000
        if p_target * v0 >= 1.0: return # Evanescent at start
        dx += 2000.0 * p_target * v0 / np.sqrt(1.0 - (p_target*v0)**2)

        is_refl = (reflection_arg is not None)

        if is_refl:
            # LEG 2: 2000 -> 0 (Up)
            # Phase determined by reflection arg "P" or "S"
            ph_up = reflection_arg[0][1]
            v1 = mi_vp * 1000 if ph_up == "P" else mi_vs * 1000
            if p_target * v1 >= 1.0: return # Evanescent reflection
            dx += 2000.0 * p_target * v1 / np.sqrt(1.0 - (p_target*v1)**2)
            z_end = 0.0
        else:
            # LEG 2: 2000 -> 4000 (Down)
            # Phase determined by refraction arg "P" or "S" (or default P/S if None?)
            # trace_rays defaults transmission to same phase if not specified.
            # But here we want to test conversions explicitly.
            # If refraction_arg is set, use it.
            ph_down = refraction_arg[0][1] if refraction_arg else wave_type
            v1 = mt_vp * 1000 if ph_down == "P" else mt_vs * 1000

            # Check critical angle for transmission
```

(continues on next page)

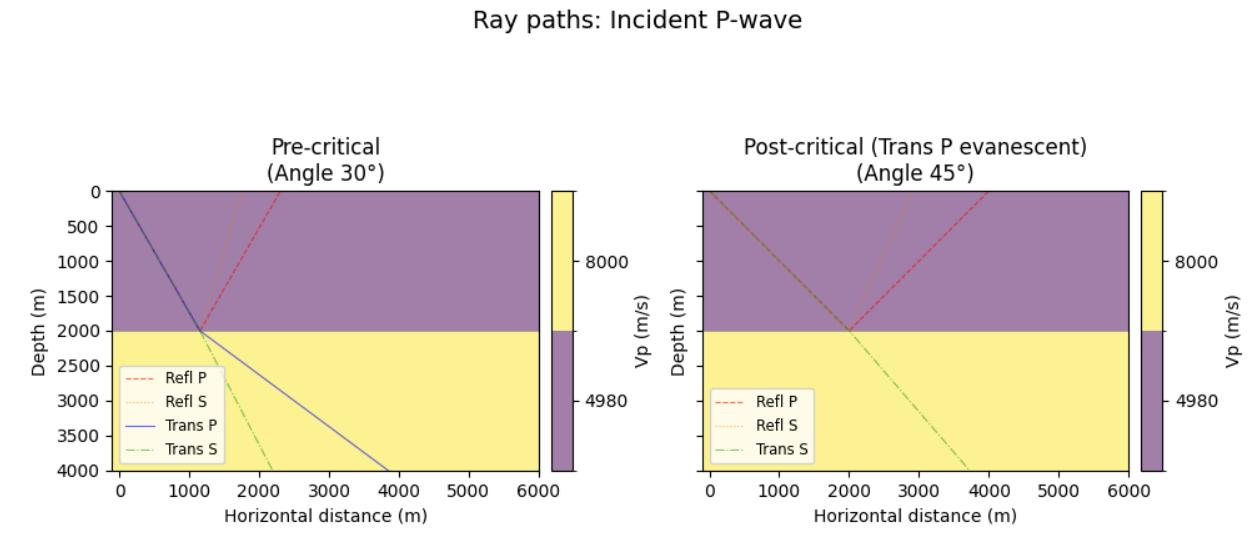
(continued from previous page)

```
            if p_target * v1 >= 1.0: return # Critical/Evanescent

            dx += (z_bot - z_int) * p_target * v1 / np.sqrt(1.0 - (p_target*v1)**2)
            z_end = z_bot

        receiver = np.array([dx, 0.0, z_end])

        # RUN THE SOLVER
        try:
            res = lt.trace_rays(
                sources=source,
                receivers=receiver,
                velocity_df=model_psv,
                source_phase=wave_type,
                reflection=reflection_arg,
                refraction=refraction_arg,
                requested={"travel_times", "rays", "ray_parameters"}
            )

            if res.rays and len(res.rays) > 0 and res.rays[0] is not None:
                lt.plot.rays_2d(
                    model_psv,
                    rays=res.rays,
                    ax=ax,
                    ray_color=color,
                    plot_model=False,
                    linestyle=style,
                    label=label,
                    xlim=(-100, 6000), ylim=(4000, 0)
                )
        except Exception:
            pass # Solver might fail if we messed up bounds, ignore for plot

    # 1. Reflected P
    run_trace(0.0, reflection_arg=[(2000.0, "P")], label="Refl P", color="r", style="-
↪-")

    # 2. Reflected S
    run_trace(0.0, reflection_arg=[(2000.0, "S")], label="Refl S", color="tab:orange",
↪ style=":")

    # 3. Transmitted P
    # Note: refraction arg is only needed if MODE CONSTANT changes.
    # P->P is default transmission.
    # But to be explicit we can convert.
    if wave_type == "P":
        run_trace(4000.0, refraction_arg=None, label="Trans P", color="b", style="-")
        run_trace(4000.0, refraction_arg=[(2000.0, "S")], label="Trans S", color=
↪"tab:green", style="-.")
    else:
        # Incident S
        run_trace(4000.0, refraction_arg=[(2000.0, "P")], label="Trans P", color="b",␣
```

(continues on next page)

(continued from previous page)

```
↪style="-")
        run_trace(4000.0, refraction_arg=None, label="Trans S", color="tab:green",␣
↪style="-.") # S->S

    # Incident ray is not plotted separately because trace_rays returns the FULL path.
    # The previous manual code overlaid legs.
    # The new code plots full V-shapes.
    # This might look slightly different (lines overlapping on the incident leg).
    # That is acceptable and actually more physically correct (showing the full ray).

    ax.legend(loc="lower left", fontsize="small")
    ax.set_title(f"{title}\n(Angle {angle}°)")


# P-incidence scenarios
scenarios_p = [
    (30, "Pre-critical"),
    (45, "Post-critical (Trans P evanescent)"),
]

fig, axes = plt.subplots(1, 2, figsize=(10, 5), sharey=True)
for i, (ang, name) in enumerate(scenarios_p):
    plot_ray_situation(ang, "P", name, axes[i])

fig.suptitle("Ray paths: Incident P-wave", fontsize=14)
fig.tight_layout()
plt.show()
```



3.3.6 Incident SV-wave coefficients

For an incident SV-wave the ray parameter sweeps from 0 to $1/V_S$ (grazing SV incidence), covering the full $0 - -90°$ range.

Critical angles (coloured lines) - three distinct thresholds:

- $\theta_c^{T(P)} = \arcsin(V_S^{(1)}/V_P^{(2)}) \approx 21.3°$ - transmitted P goes evanescent (blue dotted)
- $\theta_c^{R(P)} = \arcsin(V_S^{(1)}/V_P^{(1)}) \approx 35.6°$ - reflected P goes evanescent (red dashed)
- $\theta_c^{T(SV)} = \arcsin(V_S^{(1)}/V_S^{(2)}) \approx 39.1°$ - transmitted SV goes evanescent (green dash-dot); beyond this angle all energy is reflected as SV ($|R_{SS}| = 1$).

The reflected SV wave is always real (same medium, same velocity).

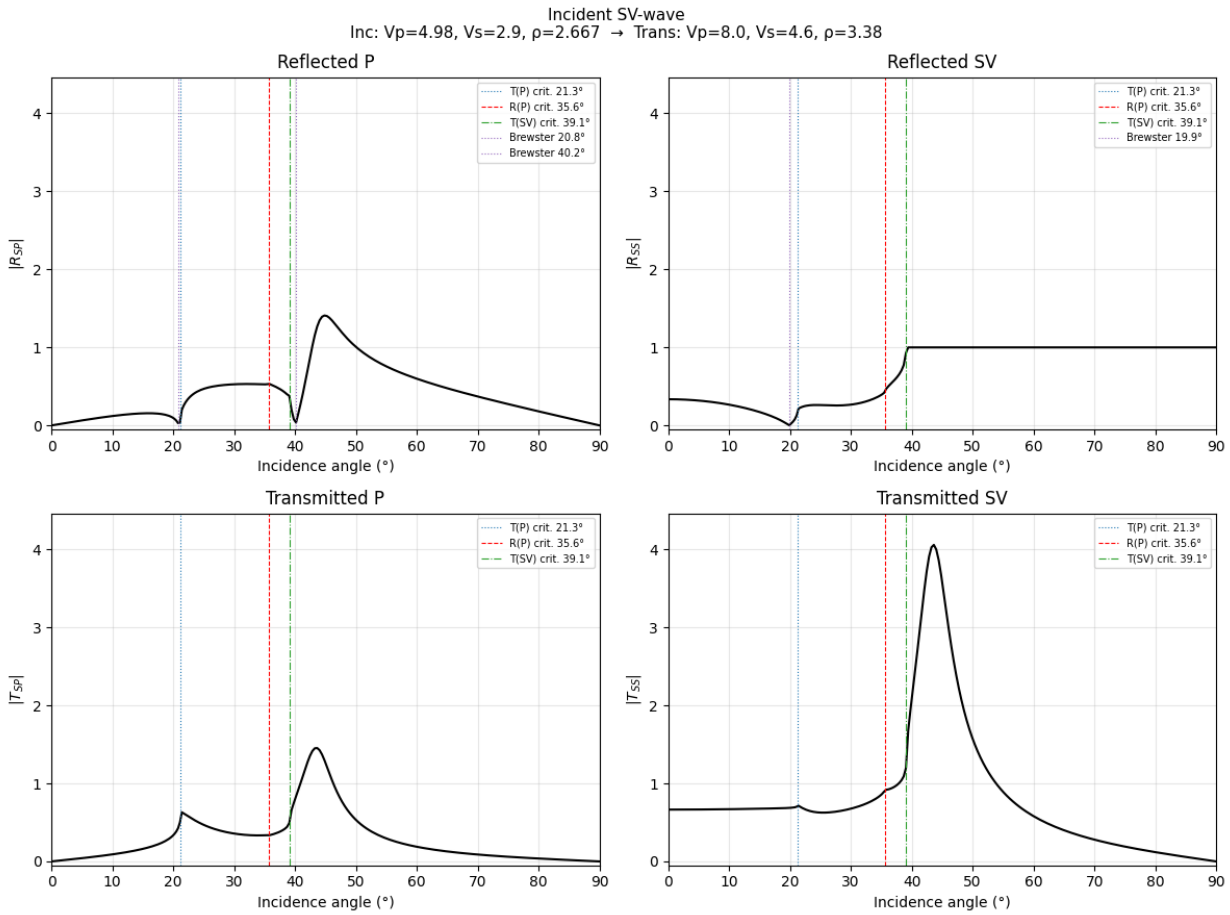
Brewster angles (purple dotted lines) - the near-zeros of $|R_{SP}|$ near 21° and 40°, and of $|R_{SS}|$ near 20°, are mode-conversion nulls governed by the full elastic contrast.

```
p_vec_sv = np.linspace(0, 1.0 / mi_vs, n_p + 1)

RT_sv = lt.psv_rt_coefficients(
    p=p_vec_sv,
    vp1=mi_vp, vs1=mi_vs, rho1=mi_rho,
    vp2=mt_vp, vs2=mt_vs, rho2=mt_rho,
)

# Incidence angle (SV-wave):  :math:`\theta = \arcsin(p \cdot V_s)`
angle_SV = np.rad2deg(np.arcsin(np.clip(p_vec_sv * mi_vs, -1, 1)))

# Critical angles
crit_tp = np.rad2deg(np.arcsin(mi_vs / mt_vp))    # transmitted P
crit_rp = np.rad2deg(np.arcsin(mi_vs / mi_vp))    # reflected P
crit_ts = np.rad2deg(np.arcsin(mi_vs / mt_vs))    # transmitted SV

# Detect Brewster angles for all SV-incident coefficients
brew_SV = lt.find_brewster_angles(
    RT_sv, angle_SV, keys=["Rsp", "Rss", "Tsp", "Tss"],
)

fig, axes = plt.subplots(2, 2, figsize=(12, 9))
fig.suptitle(
    "Incident SV-wave\n"
    f"Inc: Vp={mi_vp}, Vs={mi_vs}, ρ={mi_rho}  →  "
    f"Trans: Vp={mt_vp}, Vs={mt_vs}, ρ={mt_rho}",
    fontsize=11,
)

labels_sv = [
    (0, 0, "Rsp", r"$|R_{SP}|$",  "Reflected P"),
    (0, 1, "Rss", r"$|R_{SS}|$",  "Reflected SV"),
    (1, 0, "Tsp", r"$|T_{SP}|$",  "Transmitted P"),
    (1, 1, "Tss", r"$|T_{SS}|$",  "Transmitted SV"),
]

# Shared y-limit across all four SV-incident panels
```

(continues on next page)

(continued from previous page)

```
sv_keys = ["Rsp", "Rss", "Tsp", "Tss"]
ymax_SV = max(np.nanmax(np.abs(RT_sv[k])) for k in sv_keys) * 1.1
ymax_SV = max(ymax_SV, 0.5)

for row, col, key, ylabel, title in labels_sv:
    ax = axes[row, col]
    ax.plot(angle_SV, np.abs(RT_sv[key]), "k-", lw=1.5)
    ax.axvline(crit_tp, color="tab:blue", ls=":", lw=0.8,
               label=f"T(P) crit. {crit_tp:.1f}°")
    ax.axvline(crit_rp, color="r", ls="--", lw=0.8,
               label=f"R(P) crit. {crit_rp:.1f}°")
    ax.axvline(crit_ts, color="tab:green", ls="-.", lw=0.8,
               label=f"T(SV) crit. {crit_ts:.1f}°")
    # Brewster lines for this coefficient
    for ba in brew_SV.get(key, []):
        ax.axvline(ba, color="tab:purple", ls=":", lw=0.8,
                   label=f"Brewster {ba:.1f}°")
    ax.set_xlim(0, 90)
    ax.set_ylim(-0.05, ymax_SV)
    ax.set_xlabel("Incidence angle (°)")
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.legend(fontsize=7, loc="upper right")
    ax.grid(True, alpha=0.3)

fig.tight_layout()
plt.show()
```



3.3.7 Normalized SV-wave coefficients (Červený, 2001)

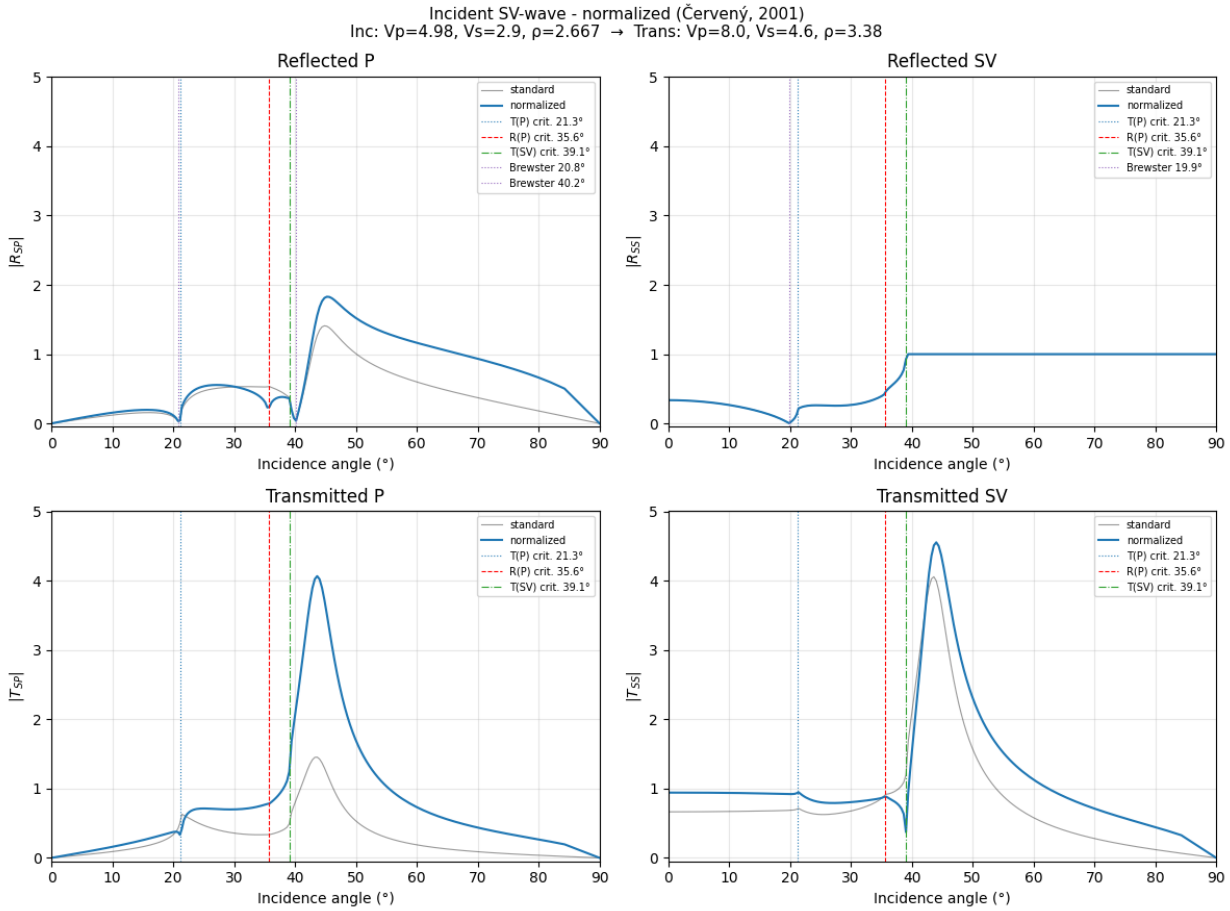
Same energy-flux normalization applied to the SV-incident coefficients. The three critical-angle markers are preserved.

```
# Mapping: key -> (v_in, rho_in, v_out, rho_out)
norm_map_SV = {
    "Rsp": (mi_vs, mi_rho, mi_vp, mi_rho),
    "Rss": (mi_vs, mi_rho, mi_vs, mi_rho),
    "Tsp": (mi_vs, mi_rho, mt_vp, mt_rho),
    "Tss": (mi_vs, mi_rho, mt_vs, mt_rho),
}

RT_norm_SV = {}
for key, (vi, ri, vo, ro) in norm_map_SV.items():
    RT_norm_SV[key] = lt.normalize_rt_coefficient(
        RT_sv[key], p_vec_sv, vi, ri, vo, ro,
    )

ymax_SVn = max(
    np.nanmax(np.abs(RT_norm_SV[k])) for k in sv_keys
) * 1.1
ymax_SVn = max(ymax_SVn, 0.5)
```

(continues on next page)

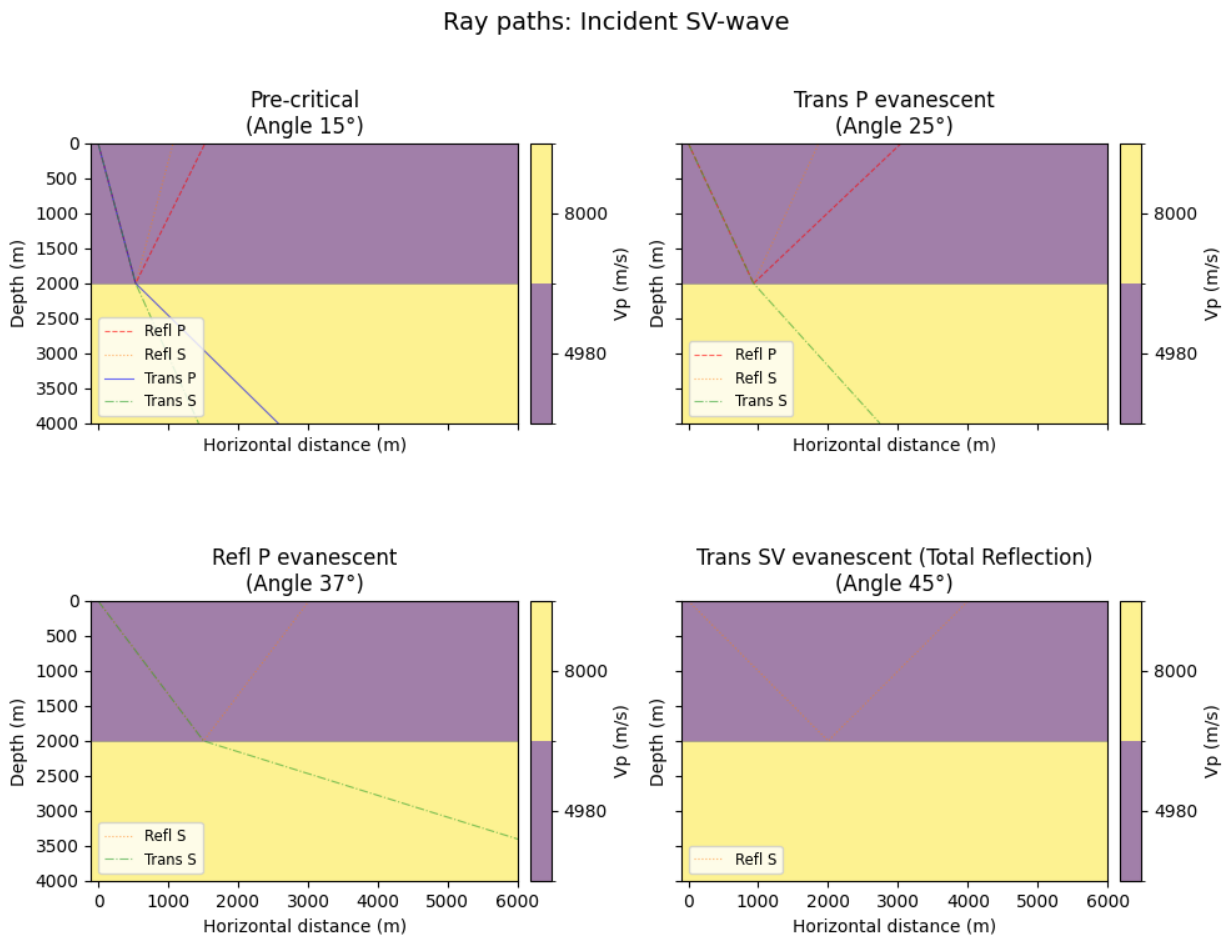
(continued from previous page)

```
fig, axes = plt.subplots(2, 2, figsize=(12, 9))
fig.suptitle(
    "Incident SV-wave - normalized (Červený, 2001)\n"
    f"Inc: Vp={mi_vp}, Vs={mi_vs}, ρ={mi_rho}  →  "
    f"Trans: Vp={mt_vp}, Vs={mt_vs}, ρ={mt_rho}",
    fontsize=11,
)

for row, col, key, ylabel, title in labels_sv:
    ax = axes[row, col]
    ax.plot(angle_SV, np.abs(RT_sv[key]), "k-", lw=0.8, alpha=0.4, label="standard")
    ax.plot(angle_SV, np.abs(RT_norm_SV[key]), "tab:blue", lw=1.5, label="normalized")
    ax.axvline(crit_tp, color="tab:blue", ls=":", lw=0.8,
               label=f"T(P) crit. {crit_tp:.1f}°")
    ax.axvline(crit_rp, color="r", ls="--", lw=0.8,
               label=f"R(P) crit. {crit_rp:.1f}°")
    ax.axvline(crit_ts, color="tab:green", ls="-.", lw=0.8,
               label=f"T(SV) crit. {crit_ts:.1f}°")
    for ba in brew_SV.get(key, []):
        ax.axvline(ba, color="tab:purple", ls=":", lw=0.8,
                   label=f"Brewster {ba:.1f}°")
    ax.set_xlim(0, 90)
    ax.set_ylim(-0.05, max(ymax_SV, ymax_SVn))
    ax.set_xlabel("Incidence angle (°)")
    ax.set_ylabel(ylabel)
    ax.set_title(title)
    ax.legend(fontsize=7, loc="upper right")
    ax.grid(True, alpha=0.3)

fig.tight_layout()
plt.show()
```



3.3.8 Ray diagrams (SV-incidence)

```
# SV-incidence scenarios
scenarios_sv = [
    (15, "Pre-critical"),
    (25, "Trans P evanescent"),
    (37, "Refl P evanescent"),
    (45, "Trans SV evanescent (Total Reflection)"),
]

fig, axes = plt.subplots(2, 2, figsize=(10, 8), sharey=True, sharex=True)
axes = axes.flatten()

for i, (ang, name) in enumerate(scenarios_sv):
    plot_ray_situation(ang, "S", name, axes[i])

fig.suptitle("Ray paths: Incident SV-wave", fontsize=14)
fig.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 3.151 seconds)

3.4 04. Amplitude analysis

This example demonstrates the computation of amplitude-related quantities alongside ray tracing: the attenuation operator t^*, geometrical spreading, and transmission coefficients.

3.4.1 Setup

```
import laytracer as lt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
```

3.4.2 Define velocity model

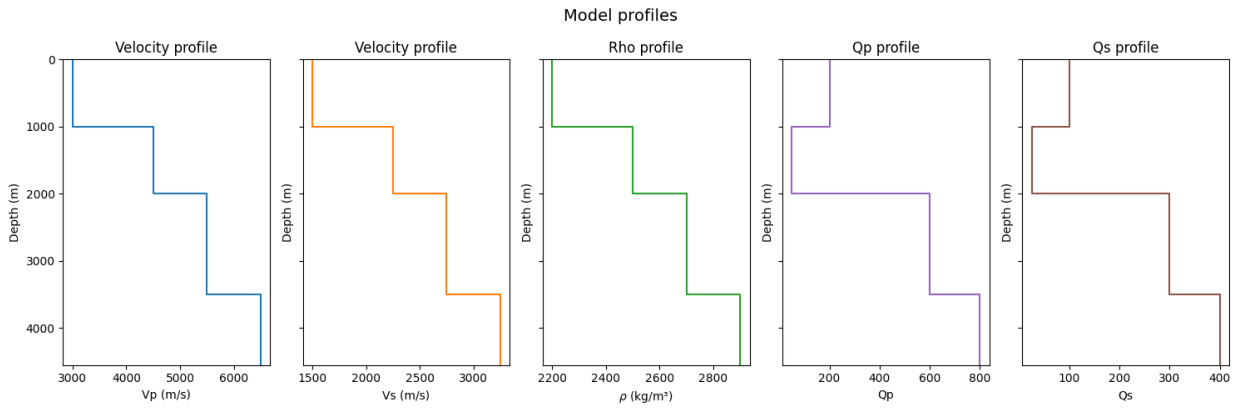
```
vel_df = pd.DataFrame({
    "Depth": [0.0, 1000.0, 2000.0, 3500.0],
    "Vp":    [3000.0, 4500.0, 5500.0, 6500.0],
    "Vs":    [1500.0, 2250.0, 2750.0, 3250.0],
    "Rho":   [2200.0, 2500.0, 2700.0, 2900.0],
    "Qp":    [200.0,  50.0,   600.0,  800.0],
    "Qs":    [100.0,  25.0,   300.0,  400.0],
})

print(vel_df)
```

```
    Depth      Vp      Vs      Rho     Qp     Qs
0     0.0  3000.0  1500.0  2200.0  200.0  100.0
1  1000.0  4500.0  2250.0  2500.0   50.0   25.0
2  2000.0  5500.0  2750.0  2700.0  600.0  300.0
3  3500.0  6500.0  3250.0  2900.0  800.0  400.0
```

3.4.3 Plot velocity model

Visualise the parameters of the model (P-wave, S-wave, density, attenuation) that will be used for the amplitude calculations.

```
fig, axes = plt.subplots(1, 5, figsize=(15, 5), sharey=True)

lt.plot.velocity_profile(vel_df, param="Vp", ax=axes[0])
lt.plot.velocity_profile(vel_df, param="Vs", ax=axes[1], color="tab:orange")
lt.plot.velocity_profile(vel_df, param="Rho", ax=axes[2], color="tab:green")
lt.plot.velocity_profile(vel_df, param="Qp", ax=axes[3], color="tab:purple")
lt.plot.velocity_profile(vel_df, param="Qs", ax=axes[4], color="tab:brown")

fig.suptitle("Model profiles", fontsize=14)
fig.tight_layout()
plt.show()
```



3.4.4 Trace rays with amplitude computation

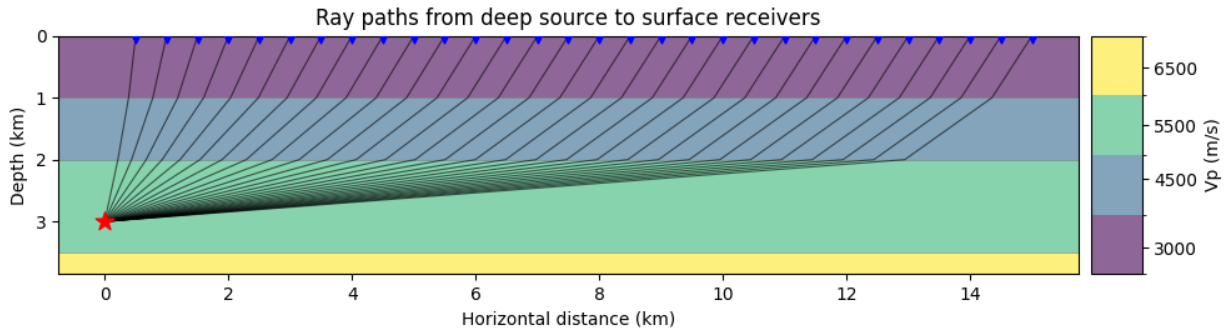
Trace P-waves from a deep source to receivers at varying offsets, requesting t^*, relative geometrical spreading, and transmission.

```
src = np.array([0.0, 0.0, 3000.0])

offsets = np.arange(500, 15001, 500)
rcvs = np.column_stack([offsets, np.zeros_like(offsets), np.zeros_like(offsets)])

result = lt.trace_rays(
    sources=src,
    receivers=rcvs,
    velocity_df=vel_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="standard",
)
```

3.4.5 Plot ray paths

Before analysing the amplitudes, let's visualize the ray paths from the source to the receivers. We overlay the rays on the P-wave velocity model to observe their trajectories.

```
fig, ax = plt.subplots(figsize=(10, 6))
lt.plot.rays_2d(
    vel_df,
    rays=result.rays,
    sources=src,
    receivers=rcvs,
    ax=ax,
    vel_type="Vp",
    plot_model=True,
    add_colorbar=True,
    model_alpha=0.6,
    discrete_colorbar=True,
    unit="km",
)
ax.set_title("Ray paths from deep source to surface receivers")
fig.tight_layout()
plt.show()
```



3.4.6 Plot amplitude quantities vs offset

Here we analyze the variation of different amplitude-related quantities as a function of receiver offset:

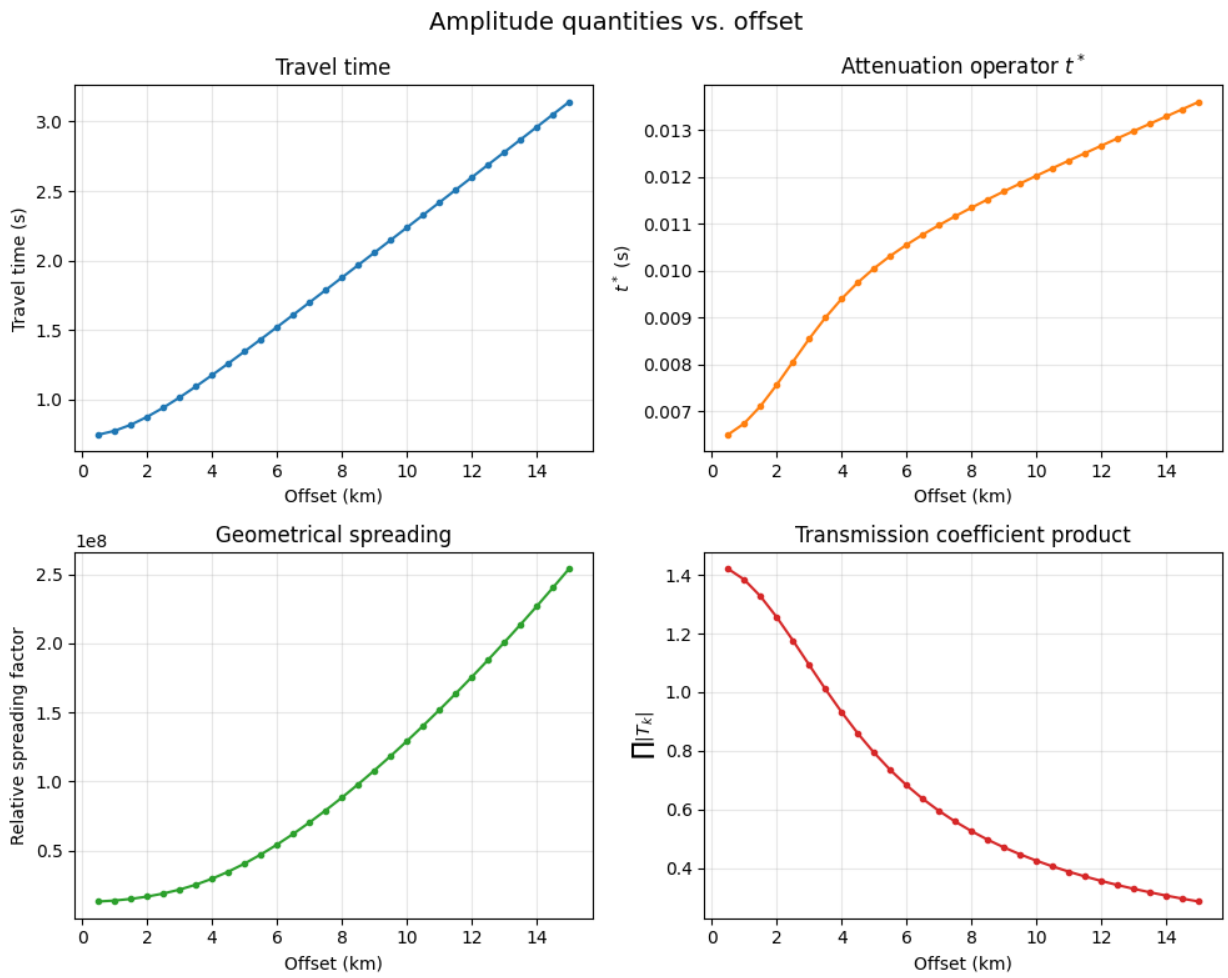
- **Travel time**: Increases smoothly with offset. The curvature is governed by the velocity structure (moveout equation).
- **Attenuation operator** t^*: Calculated as the path integral $t^* = \int_{\text{ray}} \frac{dt}{Q(s)}$. It represents cumulative anelastic decay. Rays traveling further horizontally spend more time traversing the highly attenuating layer (Q=50) between 1-2 km depth, accumulating higher t^*.
- **Geometrical spreading**: Measures the spatial divergence of the energetic ray tube. It generally grows with propagation distance, but velocity contrasts distort wavefronts, causing focusing or defocusing effects.
- **Transmission coefficient product**: The cumulative product of Zoeppritz transmission coefficients $\prod |T_k|$ across all crossed interfaces. Notice how the transmission efficiency drops sharply at larger offsets as the rays become more grazing, converting more energy into reflected modes.

```
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

axes[0, 0].plot(offsets / 1000, result.travel_times, "o-", markersize=3)
axes[0, 0].set_xlabel("Offset (km)")
axes[0, 0].set_ylabel("Travel time (s)")
axes[0, 0].set_title("Travel time")
axes[0, 0].grid(True, alpha=0.3)

axes[0, 1].plot(offsets / 1000, result.tstar, "o-", markersize=3, color="tab:orange")
axes[0, 1].set_xlabel("Offset (km)")
axes[0, 1].set_ylabel(r"$t^*$ (s)")
axes[0, 1].set_title(r"Attenuation operator $t^*$")
axes[0, 1].grid(True, alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
if result.spreading is not None:
    valid = result.spreading > 0
    axes[1, 0].plot(
        offsets[valid] / 1000, result.spreading[valid],
        "o-", markersize=3, color="tab:green",
    )
axes[1, 0].set_xlabel("Offset (km)")
axes[1, 0].set_ylabel("Relative spreading factor")
axes[1, 0].set_title("Geometrical spreading")
axes[1, 0].grid(True, alpha=0.3)

axes[1, 1].plot(
    offsets / 1000, result.trans_product,
    "o-", markersize=3, color="tab:red",
)
axes[1, 1].set_xlabel("Offset (km)")
axes[1, 1].set_ylabel(r"$\prod |T_k|$")
axes[1, 1].set_title("Transmission coefficient product")
axes[1, 1].grid(True, alpha=0.3)

fig.suptitle("Amplitude quantities vs. offset", fontsize=14)
fig.tight_layout()
plt.show()
```



3.4.7 Compare standard vs energy-flux-normalized transmission

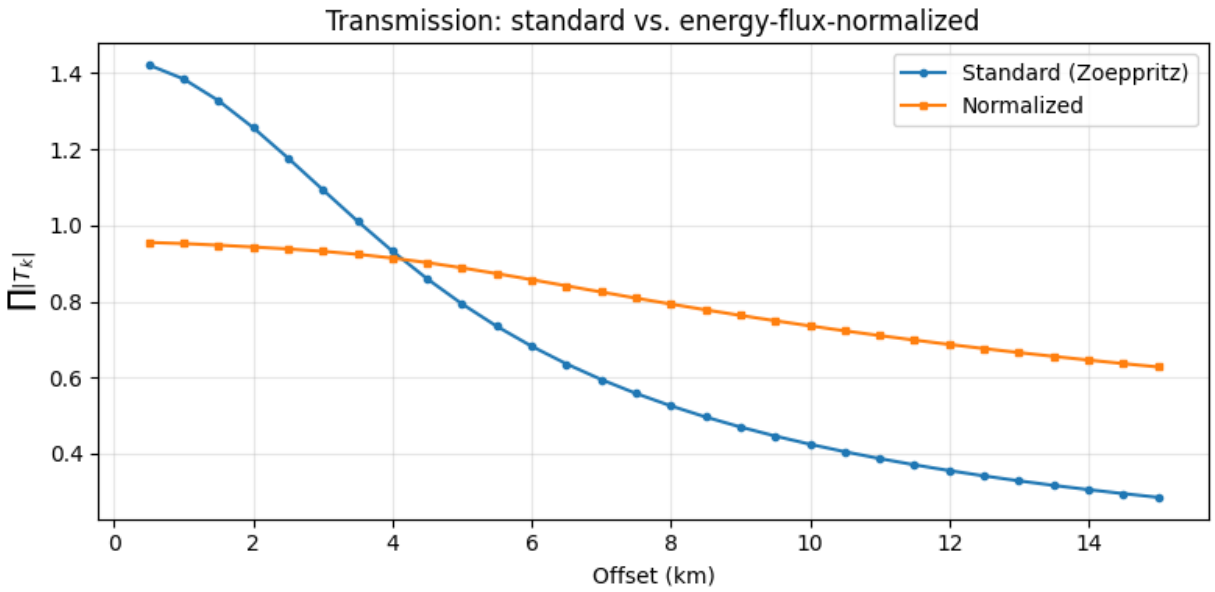
The `"normalized"` method applies the Červený [2001] Eq. 5.3.10 energy-flux normalization to the Zoeppritz coefficients. Normalized coefficients conserve energy flux across each interface, whereas standard (displacement-amplitude) coefficients do not.

```
result_normalized = lt.trace_rays(
    sources=src,
    receivers=rcvs,
    velocity_df=vel_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="normalized",
)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(offsets / 1000, result.trans_product, "o-", label="Standard (Zoeppritz)",␣
↪markersize=3)
ax.plot(offsets / 1000, result_normalized.trans_product, "s-", label="Normalized",␣
↪markersize=3)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel("Offset (km)")
ax.set_ylabel(r"$\prod |T_k|$")
ax.set_title("Transmission: standard vs. energy-flux-normalized")
ax.legend()
ax.grid(True, alpha=0.3)
fig.tight_layout()
plt.show()
```



3.4.8 Advanced Spreading Analysis

This section compares how different velocity structures (discrete channel vs. continuous gradient) distort the wavefront and influence geometrical spreading.

1. **Low-Velocity Channel**: Discrete layers funnelling rays.
2. **Continuous Gradient**: Smoothly varying velocity (discretized).

In flat 1D media, even with strong refraction, geometrical spreading for reflections remains growing and caustic-free.

Comparative Analysis of the Results:

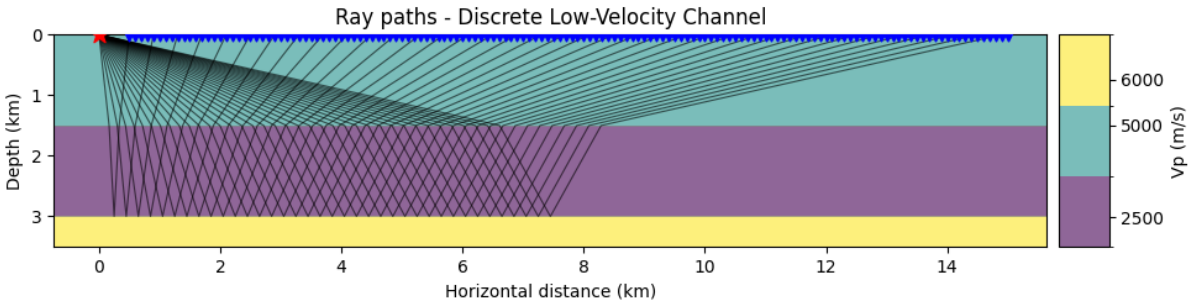
1. **Initial Magnitude**: The Gradient model (Blue) starts with a higher spreading factor than the Channel model (Green). This is because the average velocity in the gradient (5000 $\rightarrow$ 3500 m/s) is higher than in the channel (5000 $\rightarrow$ 2500 m/s). Higher average velocity leads to faster initial ray-tube expansion.
2. **Curvature & Rate of Change**:
   - The **Discrete Channel (Green)** follows a predictable parabolic growth. The refraction is "lumped" at a single interface, after which the rays travel straight.
   - The **Continuous Gradient (Blue)** stays "flatter" for mid-offsets but then undergoes an aggressive "upturn" at large offsets (>12 km). This happens because the continuous refraction makes the horizontal offset $x(p)$ extremely sensitive to changes in take-off angle as rays become grazing.
3. **Monotonicity**: Importantly, neither curve shows a "dip" or singularity. In 1D flat media, dx/dp remains positive for reflections, meaning we see no caustics, only varying rates of wavefield divergence.

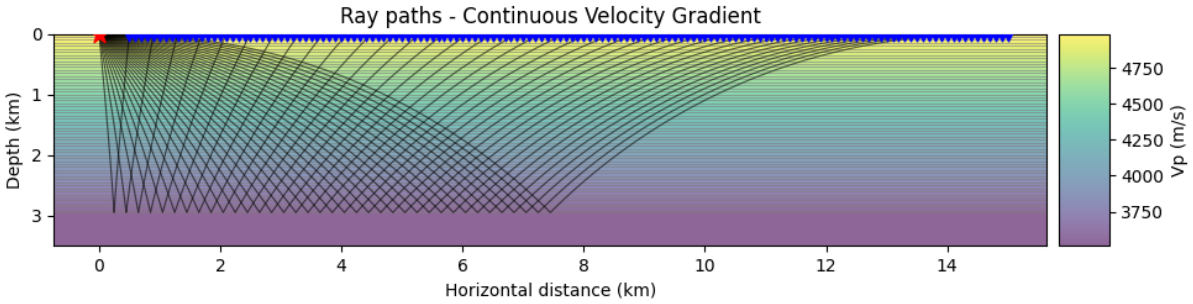
```
from laytracer.model import discretize_gradient_layer

# --- 1. Define Models & Trace Rays ---

# Discrete Channel Model
refr_df = pd.DataFrame({
    "Depth": [0.0, 1500.0, 3000.0],
    "Vp":    [5000.0, 2500.0, 6000.0],
    "Vs":    [2880.0, 1440.0, 3460.0],
    "Rho":   [2700.0, 2200.0, 2800.0],
    "Qp":    [300.0,  300.0,  300.0, ],
    "Qs":    [150.0,  150.0,  150.0, ],
})

# Continuous Gradient Model (approximated by 50m layers)
def v_func(z):
    return 5000.0 - 0.5 * z
grad_df = discretize_gradient_layer(0.0, 3000.0, v_func, dz=50.0)

src_p = np.array([0.0, 0.0, 0.0])
offsets = np.arange(500, 15001, 100)
rcvs_p = np.column_stack([offsets, np.zeros_like(offsets), np.zeros_like(offsets)])

# Trace Category 1: Channel
res_refr = lt.trace_rays(
    sources=src_p, receivers=rcvs_p, velocity_df=refr_df,
    source_phase="P", reflection=[(3000.0, "P")],
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"}, transcoef_method="standard"
)

# Trace Category 2: Gradient (reflect off last interface ~3km)
z_reflect = grad_df["Depth"].iloc[-1]
res_grad = lt.trace_rays(
    sources=src_p, receivers=rcvs_p, velocity_df=grad_df,
    source_phase="P", reflection=[(z_reflect, "P")],
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"}, transcoef_method="standard"
)

# --- 2. Advanced Visualisation ---

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 14), sharex=True)

# Panel 1: Channel Rays
lt.plot.rays_2d(
    refr_df, rays=res_refr.rays[::4],
    sources=src_p, receivers=rcvs_p, ax=ax1,
    vel_type="Vp", model_alpha=0.6,
    plot_model=True, add_colorbar=True, discrete_colorbar=True, unit="km",
)
ax1.set_title("Ray paths - Discrete Low-Velocity Channel")
ax1.set_ylim(3.5, 0)
```

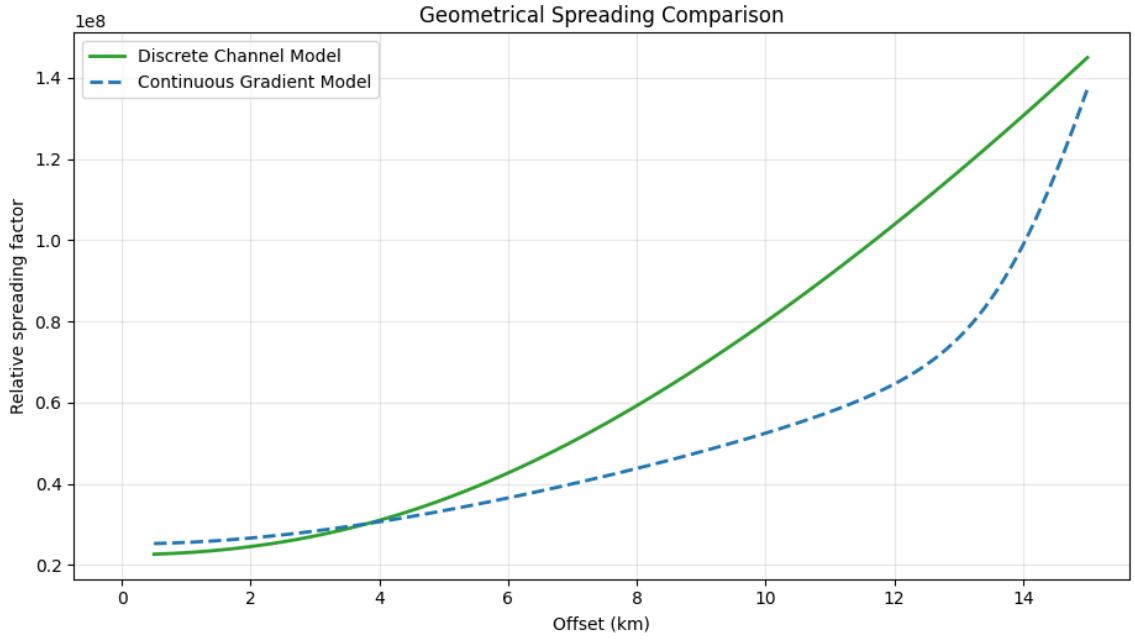
(continues on next page)

(continued from previous page)

```
ax1.tick_params(labelbottom=True) # Forces x-labels to show despite sharex=True

# Panel 2: Gradient Rays
lt.plot.rays_2d(
    grad_df, rays=res_grad.rays[::4],
    sources=src_p, receivers=rcvs_p, ax=ax2,
    vel_type="Vp", model_alpha=0.6,
    plot_model=True, add_colorbar=True, discrete_colorbar=False, unit="km",
)
ax2.set_title("Ray paths - Continuous Velocity Gradient")
ax2.set_ylim(3.5, 0)
ax2.tick_params(labelbottom=True)


# Panel 3: Spreading Comparison
valid_f = res_refr.spreading > 0
valid_g = res_grad.spreading > 0

ax3.plot(
    offsets[valid_f] / 1000, res_refr.spreading[valid_f],
    "-", color="tab:green", label="Discrete Channel Model", linewidth=2
)
ax3.plot(
    offsets[valid_g] / 1000, res_grad.spreading[valid_g],
    "--", color="tab:blue", label="Continuous Gradient Model", linewidth=2
)

ax3.set_xlabel("Offset (km)")
ax3.set_ylabel("Relative spreading factor")
ax3.set_title("Geometrical Spreading Comparison")
ax3.legend()
ax3.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```


Ray paths - Discrete Low-Velocity Channel
Depth (km)
Horizontal distance (km)
Vp (m/s)
6000
5000
2500


Ray paths - Continuous Velocity Gradient
Depth (km)
Horizontal distance (km)
Vp (m/s)
4750
4500
4250
4000
3750


Geometrical Spreading Comparison
1e8
Discrete Channel Model
Continuous Gradient Model
Relative spreading factor
Offset (km)

Total running time of the script: (0 minutes 2.618 seconds)

3.5 05. Homogeneous equivalence quality check

A two-layer model whose layers share **identical** elastic parameters must reproduce the closed-form homogeneous-medium solution exactly. This example computes travel time, t^*, geometrical spreading, the transmission-coefficient product, and their combined deterministic factor for three cases:

$$C = \frac{T}{L}$$

where $T = \prod |T_k|$ is the transmission-coefficient product and L is the relative geometrical spreading.

Interpretation of each factor:

- t^* captures attenuation effects (intrinsic and scattering) along the ray path.
- T captures interface effects (energy partition at each crossing). If an interface is physically invisible (identical elastic parameters above and below), its transmission magnitude is 1, so it should not alter amplitudes.
- L captures ray-tube divergence/convergence (pure geometry and kinematics) through the relative geometrical spreading factor.
- $C = T/L$ is therefore the deterministic, frequency-independent part of amplitude scaling.

Including attenuation, a common form is

$$A(f) \propto \frac{T}{L} \exp(-\pi f t^*)$$

so this example checks separately that both attenuation (t^*) and deterministic scaling (T/L) remain unchanged when replacing a true homogeneous medium with an equivalent two-layer representation.

The expected physics for this benchmark is strict equivalence:

- identical travel times and ray parameters,
- identical t^*, spreading, T, and T/L,
- overlapping ray geometries and offset-dependent curves,
- only machine-precision numerical differences.

It is therefore a powerful quality check for the internal consistency of the code, and a regression test to catch any future changes that might break this fundamental equivalence. The three approaches to compute the same physical quantities are:

- (a) Analytical formulas for a homogeneous medium.
- (b) LayTracer with a single-layer (homogeneous) model.
- (c) LayTracer with a two-layer model where both layers have the same Vp, Vs, ρ, Qp, Qs.

All three must agree, confirming internal consistency of the code.

3.5.1 Setup

```
import laytracer as lt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
```

3.5.2 Common parameters

We use a single set of elastic constants and a fixed source-receiver geometry throughout the example.

```
VP = 5000.0        # P-wave velocity   (m/s)
VS = VP / 1.732    # S-wave velocity   (m/s)
RHO = 2700.0       # density           (kg/m³)
QP = 500.0         # P-wave quality factor
QS = 250.0         # S-wave quality factor

src = np.array([0.0, 0.0, 500.0])       # source at 500 m depth
rcv = np.array([5000.0, 0.0, 2500.0])   # receiver at 2500 m depth

epic = np.sqrt((rcv[0] - src[0]) ** 2 + (rcv[1] - src[1]) ** 2)
dz = abs(rcv[2] - src[2])
dist = np.sqrt(epic ** 2 + dz ** 2)

print(f"Epicentral distance: {epic:.1f} m")
print(f"Depth offset:        {dz:.1f} m")
print(f"Straight-ray length: {dist:.4f} m")
```

```
Epicentral distance: 5000.0 m
Depth offset:        2000.0 m
Straight-ray length: 5385.1648 m
```

3.5.3 (a) Analytical homogeneous solution

In a homogeneous medium a ray is a straight line of length $R = \sqrt{X^2 + \Delta z^2}$, giving:

$$
\begin{aligned}
t &= R / V_P \\
p &= X / (V_P \cdot R) \\
t^* &= t / Q_P \\
L &= R \cdot V_P \\
\prod T &= 1 \quad \text{(no interfaces)}
\end{aligned}
$$

```
tt_a  = dist / VP
p_a   = epic / (VP * dist)
ts_a  = tt_a / QP
L_a   = dist * VP
T_a   = 1.0
C_a   = T_a / L_a                # combined deterministic amplitude factor

print(f"\n--- Analytical ---")
print(f"Travel time:      {tt_a:.8f} s")
print(f"Ray parameter:    {p_a:.10e} s/m")
print(f"t*:               {ts_a:.10e} s")
print(f"Spreading:        {L_a:.4f}")
print(f"Trans. product:   {T_a:.6f}")
print(f"Combined (T/L):   {C_a:.10e}")
```

```
--- Analytical ---
Travel time:      1.07703296 s
Ray parameter:    1.8569533818e-04 s/m
t*:               2.1540659229e-03 s
Spreading:        26925824.0357
Trans. product:   1.000000
Combined (T/L):   3.7139067635e-08
```

3.5.4 (b) Homogeneous model via LayTracer

A single-layer DataFrame - the simplest possible model.

```
homo_df = pd.DataFrame({
    "Depth": [0.0],
    "Vp":    [VP],
    "Vs":    [VS],
    "Rho":   [RHO],
    "Qp":    [QP],
    "Qs":    [QS],
})

res_h = lt.trace_rays(
    sources=src,
    receivers=rcv,
    velocity_df=homo_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="standard",
)

tt_h  = float(res_h.travel_times[0])
p_h   = float(res_h.ray_parameters[0])
ts_h  = float(res_h.tstar[0])
L_h   = float(res_h.spreading[0])
T_h   = float(res_h.trans_product[0])
C_h   = T_h / L_h

print(f"\n--- Homogeneous code ---")
print(f"Travel time:      {tt_h:.8f} s")
print(f"Ray parameter:    {p_h:.10e} s/m")
print(f"t*:               {ts_h:.10e} s")
print(f"Spreading:        {L_h:.4f}")
print(f"Trans. product:   {T_h:.6f}")
print(f"Combined (T/L):   {C_h:.10e}")
```

```
--- Homogeneous code ---
Travel time:      1.07703296 s
Ray parameter:    1.8569533818e-04 s/m
t*:               2.1540659229e-03 s
Spreading:        26925824.0357
Trans. product:   1.000000
```

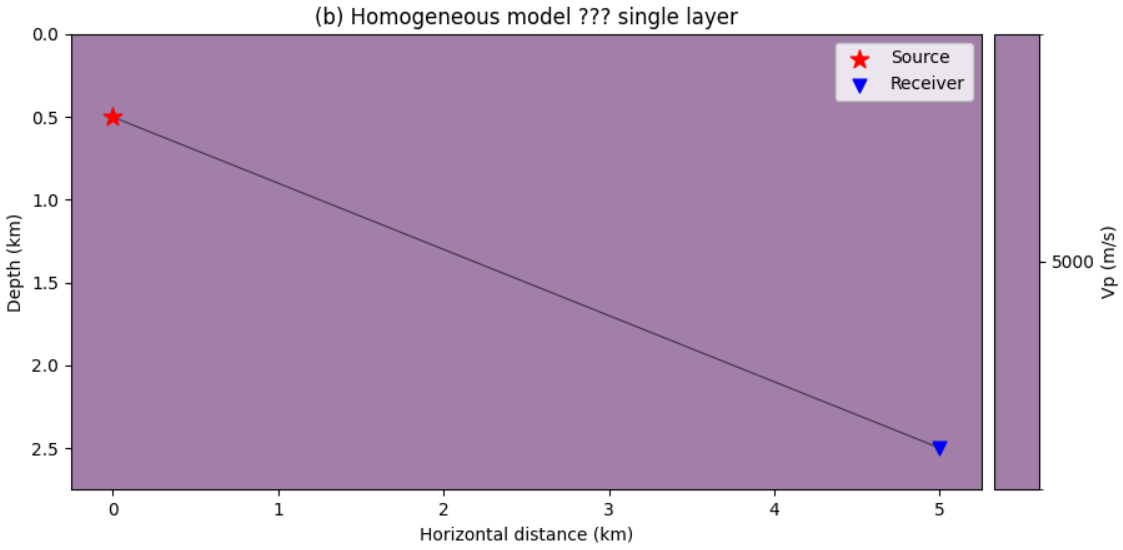
(continues on next page)

(continued from previous page)

```
Combined (T/L):   3.7139067635e-08
```

Plot the ray through the homogeneous model

```
ax = lt.plot.rays_2d(
    homo_df,
    rays=res_h.rays,
    sources=np.atleast_2d(src),
    receivers=np.atleast_2d(rcv),
    vel_type="Vp",
    add_colorbar=True,
    model_alpha=0.5,
    discrete_colorbar=True,
    unit="km",
)
ax.set_title("(b) Homogeneous model ??? single layer")
ax.legend(loc="upper right")
plt.show()
```



3.5.5 (c) Two-layer model with identical parameters

The interface at 1500 m splits the medium into two layers, but both share the same properties. A correct implementation should return a transmission coefficient of 1.0 at that interface and identical travel time, t^*, and spreading.

```
layered_df = pd.DataFrame({
    "Depth": [0.0, 1500.0],
    "Vp":    [VP, VP],
```

(continues on next page)

(continued from previous page)

```
    "Vs":    [VS, VS],
    "Rho":   [RHO, RHO],
    "Qp":    [QP, QP],
    "Qs":    [QS, QS],
})

res_l = lt.trace_rays(
    sources=src,
    receivers=rcv,
    velocity_df=layered_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="standard",
)

tt_l  = float(res_l.travel_times[0])
p_l   = float(res_l.ray_parameters[0])
ts_l  = float(res_l.tstar[0])
L_l   = float(res_l.spreading[0])
T_l   = float(res_l.trans_product[0])
C_l   = T_l / L_l

print(f"\n--- Layered code (identical layers) ---")
print(f"Travel time:      {tt_l:.8f} s")
print(f"Ray parameter:    {p_l:.10e} s/m")
print(f"t*:               {ts_l:.10e} s")
print(f"Spreading:        {L_l:.4f}")
print(f"Trans. product:   {T_l:.6f}")
print(f"Combined (T/L):   {C_l:.10e}")
```

```
--- Layered code (identical layers) ---
Travel time:      1.07703296 s
Ray parameter:    1.8569533818e-04 s/m
t*:               2.1540659229e-03 s
Spreading:        26925824.0357
Trans. product:   1.000000
Combined (T/L):   3.7139067635e-08
```

Plot the ray through the two-layer model

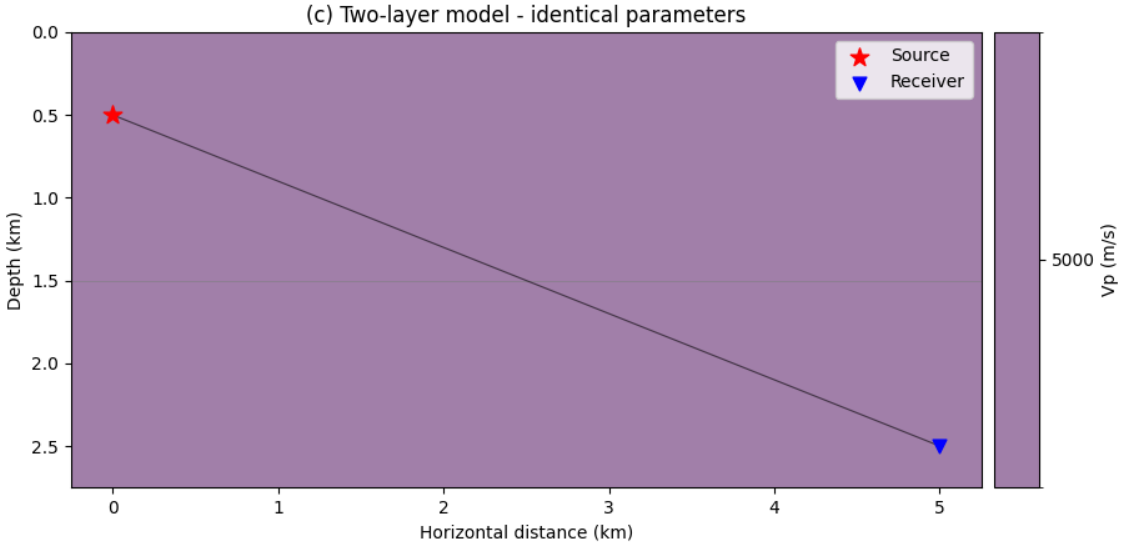
The interface at 1500 m is visible, but the ray is identical to the homogeneous case (a straight line) because the two layers share the same velocity.

```
ax = lt.plot.rays_2d(
    layered_df,
    rays=res_l.rays,
    sources=np.atleast_2d(src),
    receivers=np.atleast_2d(rcv),
    vel_type="Vp",
    add_colorbar=True,
    model_alpha=0.5,
```

(continues on next page)

(continued from previous page)

```
    discrete_colorbar=True,
    unit="km",
)
ax.set_title("(c) Two-layer model - identical parameters")
ax.legend(loc="upper right")
plt.show()
```



3.5.6 Comparison table

Collect all results into a DataFrame for easy visual inspection.

```
labels = [
    "Travel time (s)",
    "Ray parameter (s/m)",
    "t* (s)",
    "Spreading",
    "Trans. product",
    "Combined (T/L)",
]

vals_a = [tt_a, p_a, ts_a, L_a, T_a, C_a]
vals_h = [tt_h, p_h, ts_h, L_h, T_h, C_h]
vals_l = [tt_l, p_l, ts_l, L_l, T_l, C_l]

comparison = pd.DataFrame({
    "(a) Analytical": vals_a,
    "(b) Homo code":  vals_h,
    "(c) Layered code": vals_l,
```

(continues on next page)

(continued from previous page)

```
}, index=labels)

print("\n" + comparison.to_string())
```

```
                      (a) Analytical  (b) Homo code  (c) Layered code
Travel time (s)         1.077033e+00   1.077033e+00      1.077033e+00
Ray parameter (s/m)     1.856953e-04   1.856953e-04      1.856953e-04
t* (s)                  2.154066e-03   2.154066e-03      2.154066e-03
Spreading               2.692582e+07   2.692582e+07      2.692582e+07
Trans. product          1.000000e+00   1.000000e+00      1.000000e+00
Combined (T/L)          3.713907e-08   3.713907e-08      3.713907e-08
```

3.5.7 Relative errors

Quantify the mismatch between each code result and the analytical reference. Values should be at the machine-precision level.

```
rel_err_h = np.abs((np.array(vals_h) - np.array(vals_a))
                   / np.where(np.array(vals_a) != 0, vals_a, 1.0))
rel_err_l = np.abs((np.array(vals_l) - np.array(vals_a))
                   / np.where(np.array(vals_a) != 0, vals_a, 1.0))

err_df = pd.DataFrame({
    "Homo vs Analytical": rel_err_h,
    "Layered vs Analytical": rel_err_l,
}, index=labels)

print("\nRelative errors:")
print(err_df.to_string(float_format="{:.2e}".format))
```

```
Relative errors:
                      Homo vs Analytical  Layered vs Analytical
Travel time (s)                 0.00e+00              6.18e-16
Ray parameter (s/m)             0.00e+00              2.92e-16
t* (s)                          0.00e+00              6.04e-16
Spreading                       0.00e+00              0.00e+00
Trans. product                  0.00e+00              0.00e+00
Combined (T/L)                  0.00e+00              0.00e+00
```

3.5.8 Accuracy check

Assert that every quantity agrees across all three approaches to better than 10^{-10} relative tolerance. This is a hard pass/fail gate that can catch regressions.

```
TOL = 1e-10

def _check(name, ref, val, tol=TOL):
    """Return (name, ref, val, rel_err, status)."""
    if ref == 0:
        err = abs(val)
    else:
```

(continues on next page)

(continued from previous page)

```
        err = abs((val - ref) / ref)
    status = "PASS" if err <= tol else "FAIL"
    return (name, ref, val, err, status)

checks = []
# (b) homo code vs analytical
checks.append(_check("tt   (b) vs (a)", tt_a, tt_h))
checks.append(_check("p    (b) vs (a)", p_a,  p_h))
checks.append(_check("t*   (b) vs (a)", ts_a, ts_h))
checks.append(_check("L    (b) vs (a)", L_a,  L_h))
checks.append(_check("T    (b) vs (a)", T_a,  T_h))
checks.append(_check("C    (b) vs (a)", C_a,  C_h))
# (c) layered code vs analytical
checks.append(_check("tt   (c) vs (a)", tt_a, tt_l))
checks.append(_check("p    (c) vs (a)", p_a,  p_l))
checks.append(_check("t*   (c) vs (a)", ts_a, ts_l))
checks.append(_check("L    (c) vs (a)", L_a,  L_l))
checks.append(_check("T    (c) vs (a)", T_a,  T_l))
checks.append(_check("C    (c) vs (a)", C_a,  C_l))
# (c) layered code vs (b) homo code
checks.append(_check("tt   (c) vs (b)", tt_h, tt_l))
checks.append(_check("p    (c) vs (b)", p_h,  p_l))
checks.append(_check("t*   (c) vs (b)", ts_h, ts_l))
checks.append(_check("L    (c) vs (b)", L_h,  L_l))
checks.append(_check("T    (c) vs (b)", T_h,  T_l))
checks.append(_check("C    (c) vs (b)", C_h,  C_l))

check_df = pd.DataFrame(
    checks, columns=["Check", "Reference", "Value", "Rel. error", "Status"]
)

print(f"\nAccuracy checks (tolerance = {TOL:.0e}):\n")
print(check_df.to_string(index=False, float_format="{:.6e}".format))

n_fail = (check_df["Status"] == "FAIL").sum()
print(f"\nResult: {len(checks) - n_fail}/{len(checks)} checks passed.")
if n_fail:
    print(">>> SOME CHECKS FAILED - investigate! <<<")
else:
    print("All checks passed - homogeneous equivalence confirmed.")
```

```
Accuracy checks (tolerance = 1e-10):

          Check    Reference        Value   Rel. error Status
tt   (b) vs (a) 1.077033e+00 1.077033e+00 0.000000e+00   PASS
p    (b) vs (a) 1.856953e-04 1.856953e-04 0.000000e+00   PASS
t*   (b) vs (a) 2.154066e-03 2.154066e-03 0.000000e+00   PASS
L    (b) vs (a) 2.692582e+07 2.692582e+07 0.000000e+00   PASS
T    (b) vs (a) 1.000000e+00 1.000000e+00 0.000000e+00   PASS
C    (b) vs (a) 3.713907e-08 3.713907e-08 0.000000e+00   PASS
tt   (c) vs (a) 1.077033e+00 1.077033e+00 6.184897e-16   PASS
p    (c) vs (a) 1.856953e-04 1.856953e-04 2.919304e-16   PASS
```

(continues on next page)

(continued from previous page)

```
t*   (c) vs (a) 2.154066e-03 2.154066e-03 6.039939e-16   PASS
L    (c) vs (a) 2.692582e+07 2.692582e+07 0.000000e+00   PASS
T    (c) vs (a) 1.000000e+00 1.000000e+00 0.000000e+00   PASS
C    (c) vs (a) 3.713907e-08 3.713907e-08 0.000000e+00   PASS
tt   (c) vs (b) 1.077033e+00 1.077033e+00 6.184897e-16   PASS
p    (c) vs (b) 1.856953e-04 1.856953e-04 2.919304e-16   PASS
t*   (c) vs (b) 2.154066e-03 2.154066e-03 6.039939e-16   PASS
L    (c) vs (b) 2.692582e+07 2.692582e+07 0.000000e+00   PASS
T    (c) vs (b) 1.000000e+00 1.000000e+00 0.000000e+00   PASS
C    (c) vs (b) 3.713907e-08 3.713907e-08 0.000000e+00   PASS

Result: 18/18 checks passed.
All checks passed - homogeneous equivalence confirmed.
```

3.5.9 Offset-dependent equivalence

Repeat the comparison for a range of offsets, similarly to example 04, and show that homogeneous and identical-layered models produce overlapping curves for all amplitude-related quantities.

```
offsets = np.arange(500.0, 15001.0, 500.0)
src_off = np.array([src[0], src[1], rcv[2]])
rcvs = np.column_stack([
    offsets,
    np.zeros_like(offsets),
    np.full_like(offsets, src[2]),
])

res_h_off = lt.trace_rays(
    sources=src_off,
    receivers=rcvs,
    velocity_df=homo_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="standard",
)

res_l_off = lt.trace_rays(
    sources=src_off,
    receivers=rcvs,
    velocity_df=layered_df,
    source_phase="P",
    requested={"travel_times", "rays", "ray_parameters", "tstar", "spreading", "trans_
↪product"},
    transcoef_method="standard",
)
```

Offset-dependent ray paths

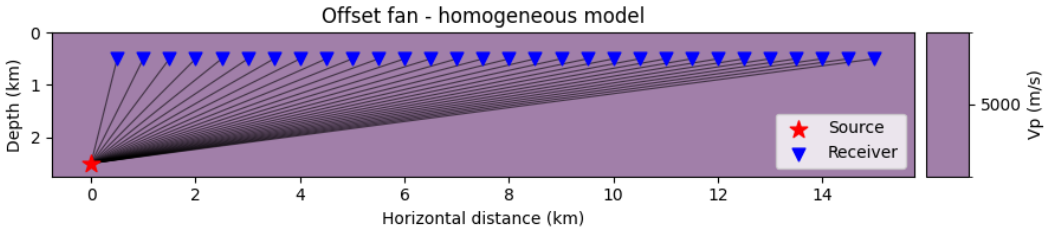
Plot the full fan of rays for all offsets in both models.

```
ax = lt.plot.rays_2d(
    homo_df,
    rays=res_h_off.rays,
    sources=np.atleast_2d(src_off),
    receivers=rcvs,
    vel_type="Vp",
    add_colorbar=True,
    model_alpha=0.5,
    discrete_colorbar=True,
    unit="km",
)
ax.set_title("Offset fan - homogeneous model")
ax.legend(loc="lower right")
plt.show()

ax = lt.plot.rays_2d(
    layered_df,
    rays=res_l_off.rays,
    sources=np.atleast_2d(src_off),
    receivers=rcvs,
    vel_type="Vp",
    add_colorbar=True,
    model_alpha=0.5,
    discrete_colorbar=True,
    unit="km",
)
ax.set_title("Offset fan - two-layer identical model")
ax.legend(loc="lower right")
plt.show()

combined_h_off = res_h_off.trans_product / res_h_off.spreading
combined_l_off = res_l_off.trans_product / res_l_off.spreading

curve_relerr = pd.DataFrame({
    "Travel time": np.abs((res_l_off.travel_times - res_h_off.travel_times) / res_h_
↪off.travel_times),
    "t*": np.abs((res_l_off.tstar - res_h_off.tstar) / res_h_off.tstar),
    "Relative Spreading": np.abs((res_l_off.spreading - res_h_off.spreading) / res_h_
↪off.spreading),
    "Trans. product": np.abs((res_l_off.trans_product - res_h_off.trans_product) / np.
↪where(res_h_off.trans_product != 0, res_h_off.trans_product, 1.0)),
    "Combined (T/L)": np.abs((combined_l_off - combined_h_off) / combined_h_off),
}, index=offsets.astype(int))

print("\nMax relative mismatch over offsets (Layered vs Homo):")
print(curve_relerr.max().to_string(float_format="{:.2e}".format))

fig, axes = plt.subplots(3, 2, figsize=(12, 10), sharex=True)
axes = axes.ravel()
```

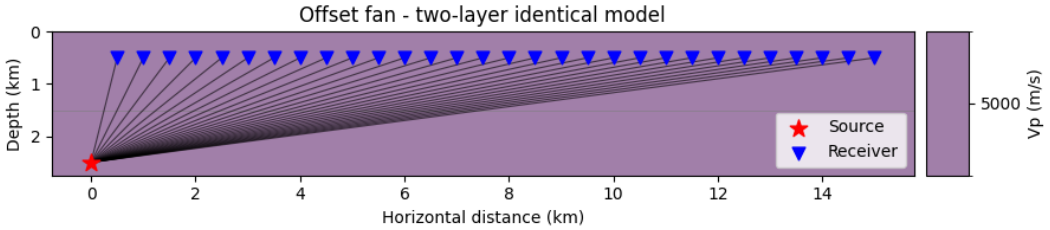
(continues on next page)

(continued from previous page)

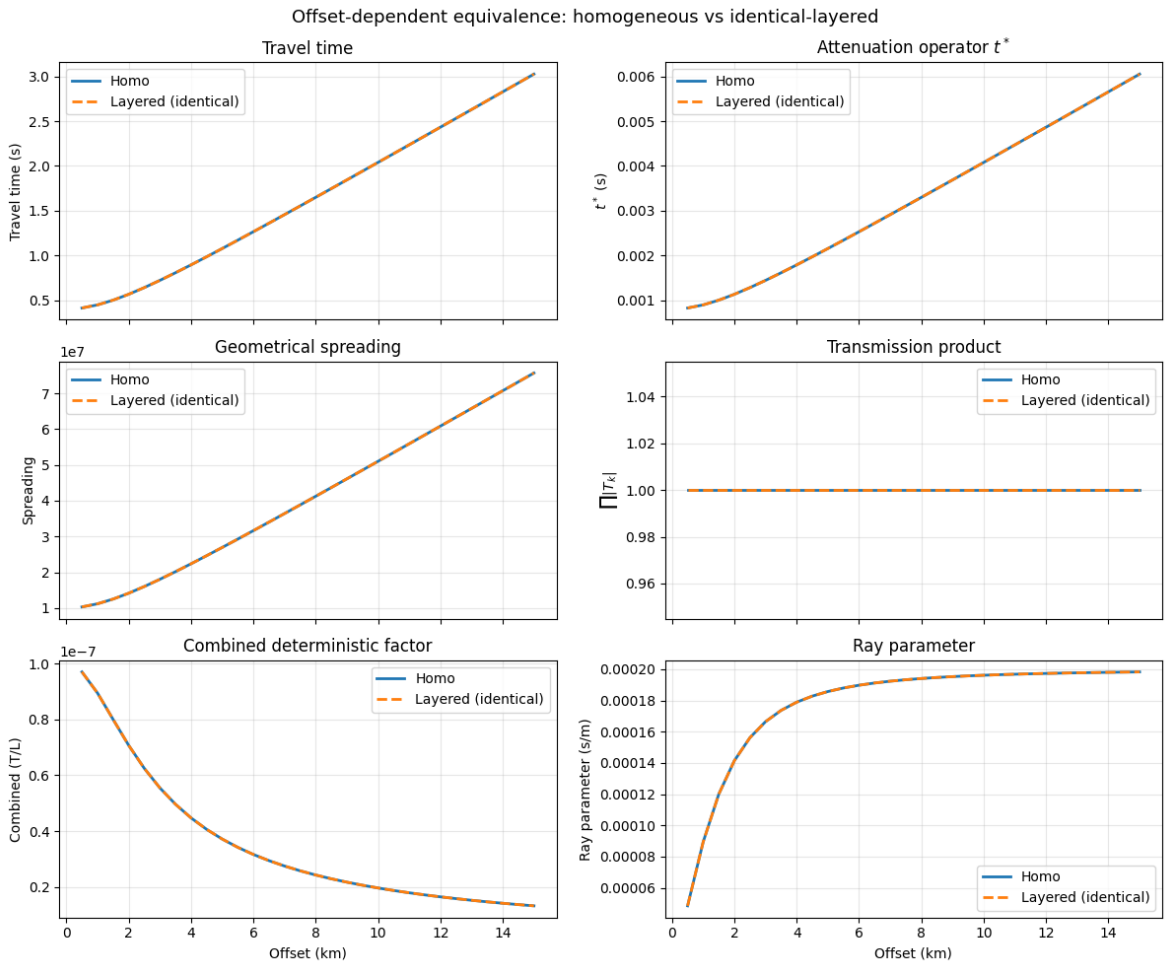
```
km = offsets / 1000.0

axes[0].plot(km, res_h_off.travel_times, "-", linewidth=2, label="Homo")
axes[0].plot(km, res_l_off.travel_times, "--", linewidth=2, label="Layered (identical)
↪")
axes[0].set_ylabel("Travel time (s)")
axes[0].set_title("Travel time")
axes[0].grid(True, alpha=0.3)

axes[1].plot(km, res_h_off.tstar, "-", linewidth=2, label="Homo")
axes[1].plot(km, res_l_off.tstar, "--", linewidth=2, label="Layered (identical)")
axes[1].set_ylabel(r"$t^*$ (s)")
axes[1].set_title(r"Attenuation operator $t^*$")
axes[1].grid(True, alpha=0.3)

axes[2].plot(km, res_h_off.spreading, "-", linewidth=2, label="Homo")
axes[2].plot(km, res_l_off.spreading, "--", linewidth=2, label="Layered (identical)")
axes[2].set_ylabel("Spreading")
axes[2].set_title("Geometrical spreading")
axes[2].grid(True, alpha=0.3)

axes[3].plot(km, res_h_off.trans_product, "-", linewidth=2, label="Homo")
axes[3].plot(km, res_l_off.trans_product, "--", linewidth=2, label="Layered␣
↪(identical)")
axes[3].set_ylabel(r"$\prod |T_k|$")
axes[3].set_title("Transmission product")
axes[3].grid(True, alpha=0.3)

axes[4].plot(km, combined_h_off, "-", linewidth=2, label="Homo")
axes[4].plot(km, combined_l_off, "--", linewidth=2, label="Layered (identical)")
axes[4].set_ylabel("Combined (T/L)")
axes[4].set_title("Combined deterministic factor")
axes[4].set_xlabel("Offset (km)")
axes[4].grid(True, alpha=0.3)

axes[5].plot(km, res_h_off.ray_parameters, "-", linewidth=2, label="Homo")
axes[5].plot(km, res_l_off.ray_parameters, "--", linewidth=2, label="Layered␣
↪(identical)")
axes[5].set_ylabel("Ray parameter (s/m)")
axes[5].set_title("Ray parameter")
axes[5].set_xlabel("Offset (km)")
axes[5].grid(True, alpha=0.3)

for ax in axes:
    ax.legend(loc="best")

fig.suptitle("Offset-dependent equivalence: homogeneous vs identical-layered",␣
↪fontsize=13)
fig.tight_layout()
plt.show()
```


Offset fan - homogeneous model
Depth (km)
0
1
2
Source
Receiver
5000
Vp (m/s)
0
2
4
6
8
10
12
14
Horizontal distance (km)

-


Offset fan - two-layer identical model
Depth (km)
0
1
2
Source
Receiver
5000
Vp (m/s)
0
2
4
6
8
10
12
14
Horizontal distance (km)

-




```
Max relative mismatch over offsets (Layered vs Homo):
Travel time          8.64e-15
t*                   8.74e-15
Relative Spreading   4.43e-15
Trans. product       2.22e-16
Combined (T/L)       4.33e-15
```

3.5.10 Conclusion

This extended quality test now validates equivalence at two levels:

1. **Single source-receiver pair**: analytical homogeneous formulas, homogeneous code, and identical-layered code agree for travel time, ray parameter, t^*, geometrical spreading, transmission product, and combined factor.
2. **Offset sweep**: homogeneous and identical-layered models produce overlapping ray fans and overlapping offset-dependent curves for all computed quantities, with relative mismatches at machine precision.

Therefore, inserting an interface between layers with identical elastic parameters introduces no spurious kinematic or amplitude effects in LayTracer, as required by the underlying physics.

Total running time of the script: (0 minutes 0.990 seconds)

CHAPTER

FOUR

API REFERENCE

LayTracer — fast two-point ray tracing in 1-D layered media.

4.1 Public API

4.1.1 Model

LayerStack(h, vp, vs[, rho, qp, qs])	Layers traversed by a ray between source and receiver depths.
ModelArrays(depths, vp, vs[, rho, qp, qs])	Pre-extracted numpy arrays from a velocity DataFrame.
build_layer_stack(vel_model, z_src, z_rcv)	Extract the layer stack between source and receiver depths.

4.1.2 Solver

solve(h, v, segments, interactions, ...[, ...])	Solve the two-point ray tracing problem for an arbitrary path.
RayResult(travel_time, ray_path, ray_parameter)	Result of a single two-point ray trace.

4.1.3 Multi-ray

trace_rays(sources, receivers, velocity_df)	Trace rays for all source-receiver pairs.
TraceResult(travel_times[, rays, ...])	Container for multi-ray tracing results.

4.1.4 Amplitude

transmission_normal(v1, rho1, v2, rho2)	Normal-incidence displacement transmission coefficient.
psv_rt_coefficients(p, vp1, vs1, rho1, vp2, ...)	Compute all eight P-SV reflection/transmission coefficients.
find_brewster_angles(rt_coefficients, angles)	Find Brewster-like angles (deep minima) in R/T coefficient curves.
normalize_rt_coefficient(R_bar, p, v_in, ...)	Apply Červený (2001) normalization to a displacement R/T coefficient.

4.1.5 Visualisation

`plot`	Standalone visualisation helpers for LayTracer.

4.2 Model

class laytracer.**LayerStack**(*h*, *vp*, *vs*, *rho=None*, *qp=None*, *qs=None*)

Layers traversed by a ray between source and receiver depths.

Parameters

h
[numpy.ndarray] Layer thicknesses (m), shape `(N,)`. Ordered from the shallowest traversed depth to the deepest.

vp
[numpy.ndarray] P-wave velocities (m/s), shape `(N,)`.

vs
[numpy.ndarray] S-wave velocities (m/s), shape `(N,)`.

rho
[numpy.ndarray or None] Densities (kg/m^3), shape `(N,)`, or *None*.

qp
[numpy.ndarray or None] P-wave quality factors, shape `(N,)`, or *None*.

qs
[numpy.ndarray or None] S-wave quality factors, shape `(N,)`, or *None*.

property n_layers

Number of layers in the stack.

v(*vel_type='Vp'*)

Return the velocity array for the requested wave type.

Parameters

vel_type
[str] `'Vp'` or `'Vs'`.

q_factor(*vel_type='Vp'*)

Return the Q-factor array for the requested wave type.

class laytracer.**ModelArrays**(*depths*, *vp*, *vs*, *rho=None*, *qp=None*, *qs=None*)

Pre-extracted numpy arrays from a velocity DataFrame.

Use *from_dataframe* to construct once and pass to *build_layer_stack* to avoid repeated DataFrame column access when tracing many rays.

classmethod from_dataframe(*vel_df*)

Extract arrays once from *vel_df*.

laytracer.**build_layer_stack**(*vel_model*, *z_src*, *z_rcv*)

Extract the layer stack between source and receiver depths.

The returned *LayerStack* contains the layers traversed by a ray connecting *z_src* and *z_rcv*, with partial thicknesses at the source and receiver layers. Layers are always ordered from the shallowest point to the deepest, regardless of which endpoint is the source.

Parameters

vel_model
[pandas.DataFrame or ModelArrays] Velocity model. A DataFrame must have columns `Depth`, `Vp`, `Vs` (optional: `Rho`, `Qp`, `Qs`). A *`ModelArrays`* instance avoids repeated column extraction when tracing many rays.

z_src
[float] Source depth (positive downward, m).

z_rcv
[float] Receiver depth (positive downward, m).

Returns

LayerStack

4.3 Solver

`laytracer.`**`solve`**(*h*, *v*, *segments*, *interactions*, *epicentral_dist*, *z_src*, *z_rcv*, *requested=None*, *return_ray_path=True*, *need_ray_parameter=True*, *need_tstar=False*, *need_spreading=False*, *need_trans_product=False*, *transcoef_method='standard'*, *tol=0.0001*, *max_iter=10*)

Solve the two-point ray tracing problem for an arbitrary path.

Parameters

h
[numpy.ndarray] Concatenated layer thicknesses for the entire path (m).

v
[numpy.ndarray] Concatenated phase velocities (Vp or Vs) for the entire path (m/s).

segments
[list of dict] Metadata for each logical monotonic segment (used for reconstruction). Dict keys: 'h', 'v', 'phase', 'start_z', 'end_z'.

interactions
[list of dict] Metadata for interactions (reflections/refractions) impacting amplitude. Dict keys: 'type', 'depth', 'in_phase', 'out_phase', 'seg_idx'.

interactions are assumed to occur at the END of the defined segment.

class `laytracer.`**`RayResult`**(*travel_time*, *ray_path*, *ray_parameter*, *tstar=None*, *spreading=None*, *trans_product=None*)

Result of a single two-point ray trace.

Attributes

travel_time
[float] Total travel time (s).

ray_path
[numpy.ndarray or None] Ray coordinates in the 2-D ray plane, shape `(M, 2)` with columns `[x, z]`. *None* if not requested.

ray_parameter
[float or None] Horizontal slowness p (s/m).

tstar
[float or None] Attenuation operator t^* (s), if requested.

spreading
[float or None] Relative geometrical spreading factor $\mathcal{L}$, if requested.

trans_product
[float or None] Product of transmission-coefficient magnitudes along the ray, if requested.

laytracer.**offset**(*q*, *h*, *lmd*)

Total horizontal range $X(q)$.

$$X(q) = \sum_{k=1}^{N} \frac{q\,\lambda_k\,h_k}{\sqrt{1 + (1-\lambda_k^2)\,q^2}}$$

Parameters

q
[float] Dimensionless ray parameter.

h
[numpy.ndarray] Layer thicknesses (m), shape `(N,)`.

lmd
[numpy.ndarray] Velocity ratios $\lambda_k = v_k / v_{\max}$, shape `(N,)`.

Returns

float

laytracer.**offset_dq**(*q*, *h*, *lmd*)

First derivative $\mathrm{d}X/\mathrm{d}q$.

$$\frac{\mathrm{d}X}{\mathrm{d}q} = \sum_{k=1}^{N} \frac{\lambda_k\,h_k}{\left[1 + (1-\lambda_k^2)\,q^2\right]^{3/2}}$$

laytracer.**offset_dq2**(*q*, *h*, *lmd*)

Second derivative $\mathrm{d}^2X/\mathrm{d}q^2$.

$$\frac{\mathrm{d}^2X}{\mathrm{d}q^2} = -3\,q \sum_{k=1}^{N} \frac{(1-\lambda_k^2)\,\lambda_k\,h_k}{\left[1 + (1-\lambda_k^2)\,q^2\right]^{5/2}}$$

laytracer.**q_from_p**(*p*, *vmax*)

Convert horizontal slowness *p* to dimensionless parameter *q*.

$$q = \frac{p\,v_{\max}}{\sqrt{1 - p^2\,v_{\max}^2}}$$

laytracer.**p_from_q**(*q*, *vmax*)

Convert dimensionless parameter *q* to horizontal slowness *p*.

$$p = \frac{q}{v_{\max}\,\sqrt{1+q^2}}$$

laytracer.**initial_q**(*X_target*, *h*, *lmd*)

Asymptotic initial estimate q_0 for the Newton iteration.

Two linear asymptotes of $X(q)$:

- **Near-field** ($q \to 0$): $X \approx m_0\, q$, where $m_0 = \sum \lambda_k h_k$.
- **Far-field** ($q \to \infty$): $X \approx m_\infty q + b_\infty$, where $m_\infty = \sum_{k:\,\lambda_k=1} h_k$ and $b_\infty = \sum_{k:\,\lambda_k<1} \frac{\lambda_k h_k}{\sqrt{1-\lambda_k^2}}$.

The initial estimate is chosen as:

- $q_0 = X/m_0$ when $X \le X^*$ (near-field),
- $q_0 = (X - b_\infty)/m_\infty$ otherwise.

where $X^* = m_0\, q^*$ and $q^* = b_\infty/(m_0 - m_\infty)$.

`laytracer.`**`newton_step`**(*q_i*, *X_target*, *h*, *lmd*)

One second-order (quadratic) Newton step.

Solves the local quadratic approximation

$$\tfrac{1}{2}\, X''(q_i)\, \Delta q^2 + X'(q_i)\, \Delta q + \big[X(q_i) - X_R\big] = 0$$

and selects the root that minimises $|X(q_i + \Delta q) - X_R|$.

Parameters

q_i
[float] Current iterate.

X_target
[float] Desired horizontal range.

h, lmd
[numpy.ndarray] Layer thicknesses and velocity ratios.

Returns

q_new
[float] Updated iterate.

X_new
[float] Offset at `q_new`.

4.4 Multi-ray interface

`laytracer.`**`trace_rays`**(*sources*, *receivers*, *velocity_df*, *source_phase='P'*, *reflection=None*, *refraction=None*, *requested=('travel_times', 'rays', 'ray_parameters')*, *transcoef_method='standard'*, *n_jobs=-1*, *backend='loky'*, *sequential_limit=10000*, *rays_per_chunk=None*, *tol=0.0001*, *max_iter=10*, *verbose=True*)

Trace rays for all source-receiver pairs.

Every source is paired with every receiver, producing `n_src x n_rcv` rays (each source traced to all receivers).

Parameters

sources
[numpy.ndarray] Source coordinates, shape `(n_src, 3)` or `(3,)`.

receivers
[numpy.ndarray] Receiver coordinates, shape `(n_rcv, 3)` or `(3,)`.

velocity_df
[pandas.DataFrame] Velocity model with columns `Depth`, `Vp`, `Vs` and optionally `Rho`, `Qp`, `Qs`.

source_phase
[str] Initial wave phase at source: `'P'` or `'S'`.

reflection
[list of (depth, phase), optional] Reflection points as `(depth, out_phase)` tuples.

refraction
[list of (depth, phase), optional] Refraction / mode-conversion points as `(depth, out_phase)` tuples.

requested
[sequence of str, optional] Explicit set of requested outputs. Valid names are `travel_times`, `rays`, `ray_parameters`, `tstar`, `spreading`, and `trans_product`. The set must include `travel_times`.

transcoef_method
[str] `'standard'` (Zoeppritz) or `'normalized'`.

n_jobs
[int] Number of parallel jobs (`-1` = all physical cores).

backend
[str] Joblib parallel backend (default `'loky'`).

sequential_limit
[int] If the total number of rays is below this threshold, run sequentially to avoid parallel overhead.

rays_per_chunk
[int or None] Maximum number of rays to process per memory-bounded chunk.

tol
[float] Newton convergence tolerance (m).

max_iter
[int] Maximum Newton iterations.

verbose
[bool] If *True*, print progress information for chunked processing.

Returns

TraceResult

class laytracer.**TraceResult**(*travel_times*, *rays=None*, *ray_parameters=None*, *tstar=None*, *spreading=None*, *trans_product=None*)

Container for multi-ray tracing results.

Attributes

travel_times
[numpy.ndarray] Travel times (s), shape `(n_rays,)`.

rays
[list of numpy.ndarray or None] Ray paths; each element is shape `(M_i, 3)` in the original 3-D coordinate system. *None* if not requested.

ray_parameters
[numpy.ndarray or None] Horizontal slowness p for each ray, shape `(n_rays,)`.

tstar
[numpy.ndarray or None] Attenuation operator t^* for each ray, shape `(n_rays,)`.

spreading
[numpy.ndarray or None] Relative geometrical spreading factor for each ray, shape `(n_rays, )`.

trans_product
[numpy.ndarray or None] Product of transmission coefficients along each ray.

4.5 Amplitude

`laytracer.`**`psv_rt_coefficients`**(*p*, *vp1*, *vs1*, *rho1*, *vp2*, *vs2*, *rho2*)

Compute all eight P-SV reflection/transmission coefficients.

Direct port of the Ammon MATLAB `PSVRTmatrix` function (Lay & Wallace / Aki & Richards formulation).

For an incident P-wave the system unknowns are $[R_{PP},\ R_{PS},\ T_{PP},\ T_{PS}]$. For an incident SV-wave the unknowns are $[R_{SP},\ R_{SS},\ T_{SP},\ T_{SS}]$.

Parameters

p
[float or array_like] Ray parameter (horizontal slowness, s/m). Scalar or 1-D array.

vp1, vs1, rho1
[float] P-velocity, S-velocity, density of the *incident* medium.

vp2, vs2, rho2
[float] Same for the *transmitted* medium.

Returns

dict
Keys `'Rpp'`, `'Rps'`, `'Rss'`, `'Rsp'`, `'Tpp'`, `'Tps'`, `'Tss'`, `'Tsp'`. Each value has the same shape as *p* (complex).

`laytracer.`**`normalize_rt_coefficient`**(*R_bar*, *p*, *v_in*, *rho_in*, *v_out*, *rho_out*)

Apply Červený (2001) normalization to a displacement R/T coefficient.

Eq. 5.3.10:

$$R_{mn} = \bar{R}_{mn} \left(\frac{V(\tilde{Q})\,\rho(\tilde{Q})\,P(\tilde{Q})}{V(Q)\,\rho(Q)\,P(Q)} \right)^{1/2}$$

where $P(Q) = (1 - V^2 p^2)^{1/2}$.

Parameters

R_bar
[complex or float] Standard (unnormalized) displacement coefficient.

p
[float] Ray parameter (horizontal slowness, s/m).

v_in
[float] Phase velocity of the incident wave (m/s).

rho_in
[float] Density of the incident medium (kg/m^3).

v_out
[float] Phase velocity of the outgoing (reflected/transmitted) wave (m/s).

rho_out
[float] Density of the outgoing medium (kg/m^3).

Returns

complex or float
Normalized displacement coefficient.

laytracer.**find_brewster_angles**(*rt_coefficients*, *angles*, *keys=None*, *threshold=0.05*, *order=20*)

Find Brewster-like angles (deep minima) in R/T coefficient curves.

A **Brewster angle** (by analogy with optics) is an incidence angle at which a reflection or transmission coefficient passes through zero or a deep minimum. Unlike critical angles, which depend only on velocity ratios, Brewster angles depend on *all six* elastic parameters (Vp, Vs, ρ in both media) and arise from destructive interference between displacement potentials at the interface.

Parameters

rt_coefficients
[dict] Output of *psv_rt_coefficients* — each value is a 1-D array of complex coefficients.

angles
[array_like] Incidence angles (degrees) corresponding to the ray-parameter samples used in *rt_coefficients*. Must have the same length as the coefficient arrays.

keys
[list of str, optional] Which coefficient keys to search (e.g. `['Rps', 'Rss']`). By default all eight keys are searched.

threshold
[float, optional] Only report minima whose absolute value is below this value. Default 0.05.

order
[int, optional] Half-window size passed to `scipy.signal.argrelmin` for local-minimum detection. Default 20.

Returns

dict[str, list[float]]
Mapping from coefficient key to a list of Brewster angles (degrees). Keys with no detected minima are omitted.

laytracer.**transmission_normal**(*v1*, *rho1*, *v2*, *rho2*)

Normal-incidence displacement transmission coefficient.

$$T = \frac{2\,Z_1}{Z_1 + Z_2}, \qquad Z_i = \rho_i\, v_i$$

Parameters

v1, rho1
[float] Velocity (m/s) and density (kg/m^3) on the incident side.

v2, rho2
[float] Velocity and density on the transmitted side.

Returns

float
Displacement amplitude transmission coefficient.

4.6 Visualisation

laytracer.plot.**velocity_profile**(*vel_df*, *param='Vp'*, *ax=None*, *color=None*, *label=None*, *xlim=None*, *ylim=None*, *unit='m'*, ***kwargs*)

Plot a 1-D model parameter–depth step profile.

Parameters

vel_df
[pandas.DataFrame] Velocity model.

param
[str, optional] `'Vp'` (default), `'Vs'`, `'Rho'`, `'Qp'`, or `'Qs'`.

ax
[matplotlib.axes.Axes, optional] Axes to plot on. Created if *None*.

color
[str, optional] Line colour.

label
[str, optional] Legend label.

xlim, ylim
[tuple, optional] Axis limits `(min, max)`.

unit
[str, optional] `'m'` (default) or `'km'`. Scales the vertical depth axis.

Returns

matplotlib.axes.Axes

laytracer.plot.**rays_2d**(*vel_df*, *rays*, *vel_type='Vp'*, *sources=None*, *receivers=None*, *ax=None*, *ray_color='k'*, *ray_alpha=0.6*, *xlim=None*, *ylim=None*, *unit='m'*, *plot_model=True*, *add_colorbar=False*, *discrete_colorbar=False*, *model_alpha=1.0*, *equal_scale=True*, *colorbar_orientation='vertical'*, ***kwargs*)

Plot ray paths over a 2-D layered velocity cross-section.

Parameters

vel_df
[pandas.DataFrame] Velocity model.

rays
[list of numpy.ndarray] Each element is shape `(M, 2)` or `(M, 3)`. If 3-D, the first two columns are treated as horizontal/depth.

vel_type
[str] `'Vp'` or `'Vs'` — used for layer colouring.

sources, receivers
[numpy.ndarray, optional] Coordinate arrays for plotting markers.

ax
[matplotlib.axes.Axes, optional]

ray_color
[str]

ray_alpha
[float]

xlim, ylim
[tuple, optional]

plot_model
[bool] If *True* (default), plot the velocity model background and set axis labels/titles. If *False*, only plot the rays and markers.

unit
[str] `'m'` (default) or `'km'`. Scales coordinates and labels.

add_colorbar
[bool] If *True* (default *False*), add a colorbar for the velocity model. Only applies if *plot_model* is True.

discrete_colorbar
[bool] If *True* (default *False*), quantize the colormap to the unique velocity values in the model.

model_alpha
[float] Opacity of the velocity model layers (0.0 to 1.0). Default 1.0.

equal_scale
[bool] If *True* (default *True*), force equal scaling for x and y axes using `ax.set_aspect('equal')`.

colorbar_orientation
[str] `'vertical'` (default) or `'horizontal'`.

Returns

matplotlib.axes.Axes

`laytracer.plot.`**`rays_3d`**(*vel_df*, *rays*, *vel_type='Vp'*, *sources=None*, *receivers=None*, *ray_color='red'*, *opacity=0.3*, ***kwargs*)

Interactive 3-D ray visualisation using Plotly.

Parameters

vel_df
[pandas.DataFrame] Velocity model.

rays
[list of numpy.ndarray] Each element is shape `(M, 3)`.

vel_type
[str] `'Vp'` or `'Vs'`.

sources, receivers
[numpy.ndarray, optional] Coordinate arrays for plotting markers.

ray_color
[str] Ray trace colour.

opacity
[float] Layer surface opacity.

Returns

plotly.graph_objects.Figure

CHAPTER

FIVE

CHANGELOG

5.1 [v0.3.0] - 2026-03-14

5.1.1 Added

- add explicit `requested={...}` output selection to `trace_rays()` and `solve()`, replacing the coarse amplitude switch (#3)

5.1.2 Changed

- only build and return ray paths, ray parameters, and path-dependent scalar outputs when explicitly requested (#3)

5.1.3 Fixed

- fix degenerate direct-ray amplitude outputs for zero-offset vertical rays and exact same-point rays (#2)
- make amplitude result packing robust to mixed `None` and finite values (#2)
- add regression tests for degenerate direct-ray amplitude cases (#2)

5.2 [v0.2.1] - 2026-03-07

5.2.1 Added

- `normalize_rt_coefficient()` function in `amplitude.py` implementing Červený (2001) Eq. 5.3.10 energy-flux normalization of R/T coefficients (#1)
- `transcoef_method="normalized"` option in `trace_rays` and `solve` for energy-flux-normalized transmission coefficient products (#1)
- new tests: `test_transmission_normalized`, `test_normalized_vertical_ray` (#1)
- methodology docs: section on energy-flux-normalized coefficients (#1)

5.2.2 Changed

- `transcoef_method` values renamed: `"angle"` → `"standard"`, `"angle_normalized"` → `"normalized"` (#1)
- default `transcoef_method` is now `"standard"` (was `"angle"`) (#1)
- updated documentation to reflect new method names and default (#1)
- updated example 5 to use new method names and default (#1)
- updated example 4 to compare standard vs normalized transmission coefficients (#1)

- updated example 3 to show comparison of standard vs normalized transmission and reflection coefficients (#1)

5.2.3 Removed

- `"normal"` (impedance-only) transmission coefficient method — only `"standard"` and `"normalized"` are supported (#1)
- normal-incidence section removed from methodology docs (#1)

5.3 [v0.2.0] - 2026-03-04

5.3.1 Added

- `ModelArrays` dataclass for pre-extracted NumPy arrays, avoiding repeated DataFrame column extraction during parallel tracing
- fast path in `_trace_one` for direct waves (no reflections/refractions)
- memory-aware `rays_per_chunk` auto-sizing and progress reporting with ETA in `trace_rays`

5.3.2 Changed

- `build_layer_stack` now accepts both `pd.DataFrame` and `ModelArrays` (unified from the former `build_layer_stack` / `build_layer_stack_fast` pair)
- batched parallel dispatch: rays are grouped into ~n_workers batches with lightweight NumPy-only serialisation instead of per-ray DataFrame pickling
- updated documentation index page

5.3.3 Removed

- dead first-pass loop in `_trace_one` that built unused variables

5.3.4 Fixed

- handle degenerate case in `_trace_one` function to return minimal result
- fix NaN results for same-depth source–receiver rays (e.g. stations and grid points both at z = 0): zero-thickness layer stack is now handled as a horizontal straight-line ray with correct travel time, geometrical spreading, attenuation t*, and transmission product instead of returning NaN

5.4 [v0.1.0] - 2026-03-03

5.4.1 Added

- Files for initial release
- This changelog
- GitHub Actions CI for pytest, docs build, and release automation

BIBLIOGRAPHY

[1] X. Fang and X. Chen. A fast and robust two-point ray tracing method in layered media with constant or linearly varying layer velocity. *Geophysical Prospecting*, 67(7):1648–1661, 2019. doi:10.1111/1365-2478.12799.

[2] K. Aki and P. G. Richards. *Quantitative Seismology*. University Science Books, 2nd edition, 2002.

[3] V. Červený. *Seismic Ray Theory*. Cambridge University Press, 2001. doi:10.1017/CBO9780511529399.

[4] T. Lay and T. C. Wallace. *Modern Global Seismology*. Academic Press, 1995.

PYTHON MODULE INDEX

INDEX

Online documentation

danikiev.github.io/LayTracer