
ttcrpy

Release 1.3.4

Bernard Giroux

Feb 07, 2024

CONTENTS:

- 1 Getting started 3**
 - 1.1 Installing tterpy 3
 - 1.2 Simple examples 3
- 2 Model discretization 5**
 - 2.1 2D models 5
 - 2.2 3D models 5
 - 2.3 Assigning velocity/slowness 5
- 3 Algorithms 7**
 - 3.1 Shortest-Path 7
 - 3.2 Dynamic Shortest-Path 8
 - 3.3 Fast-Sweeping 9
 - 3.4 Computing traveltimes from raypaths 10
- 4 Performance 11**
 - 4.1 3D Rectilinear Grids 11
- 5 Documentation for the python code 19**
 - 5.1 Module rgrid 19
 - 5.2 Module tmesh 35
- 6 References 47**
- 7 Indices and tables 49**
- Python Module Index 51**
- Index 53**

tcrpy is a package for computing traveltimes and raytracing that was developed with geophysical applications in mind, e.g. ray-based seismic/GPR tomography and microseismic event location (joint hypocenter-velocity inversion). The package contains code to perform computation on 2D and 3D rectilinear grids, as well as 2D triangular and 3D tetrahedral meshes. Three different algorithms have been implemented: the Fast-Sweeping Method, the Shortest-Path Method, and the Dynamic Shortest-Path Method. Calculations can be run in parallel on a multi-core machine. The core computing code is written in C++, and has been wrapped with cython.

The source code of this project is hosted on [GitHub](#).

If you use *tcrpy*, please cite

Giroux B. 2021. *tcrpy*: A Python package for traveltime computation and raytracing. *SoftwareX*, vol. 16, 100834. doi: 10.1016/j.softx.2021.100834 <https://www.sciencedirect.com/science/article/pii/S2352711021001217>

GETTING STARTED

1.1 Installing ttcipy

You can use pip to install the package by doing:

```
pip install ttcipy
```

1.1.1 Requirements

ttcipy needs the following packages:

- numpy (<https://numpy.org>)
- scipy (<https://www.scipy.org>)
- vtk (<https://www.vtk.org>)

1.2 Simple examples

An example showing how easy it is to use the code can be found at https://github.com/groupeLIAMG/ttcipy/blob/master/examples/example_Grid3d.ipynb

A second example illustrating how to run jobs in parallel is given at https://github.com/groupeLIAMG/ttcipy/blob/master/examples/example_tmesh_parallel.ipynb

An example illustrating how to use gmsh to build models with specific geometries is <https://github.com/groupeLIAMG/ttcipy/blob/master/examples/example4.ipynb>

Raytracing in anisotropic media is shown in <https://github.com/groupeLIAMG/ttcipy/blob/master/examples/example5.ipynb>

MODEL DISCRETIZATION

`ttcrpy` supports a number of discretization schemes. 2D and 3D models are possible.

2.1 2D models

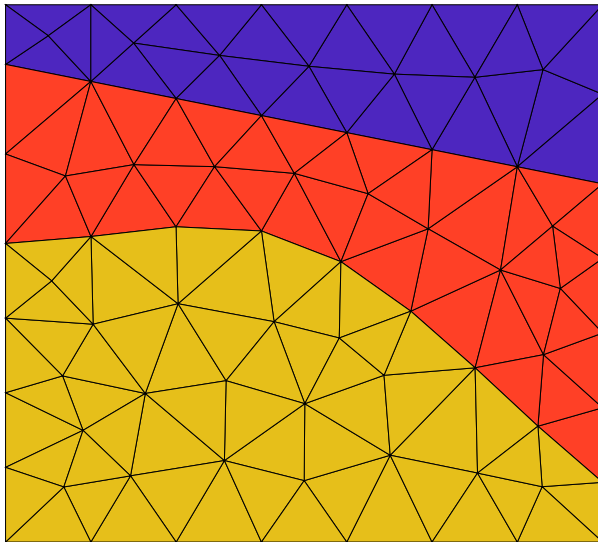
Rectilinear grids and triangular meshes can be built to perform the calculations. By convention, the coordinate axis system is (x, z) , e.g. when saving the models to VTK format.

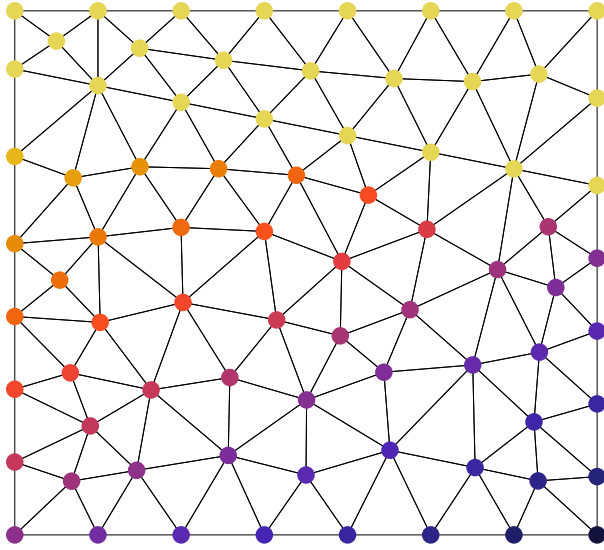
2.2 3D models

Rectilinear grids and tetrahedral meshes can be used for 3D calculations.

2.3 Assigning velocity/slowness

Prior to performing traveltimes computations, it is necessary to define the slowness distribution in space. Two options are possible.





In the leftmost case, slowness values are assigned to the cells of the mesh. In the rightmost case, slowness values are assigned to grid nodes. In the latter case, traveltime computation between two nodes is done by taking the average of the slowness values at the two nodes.

The choice mostly depends on the application. For example, in traveltime tomography the problem is to use traveltime data to estimate the slowness model. Rectilinear grids contain less cells than nodes, hence the number of unknown parameters is less if slowness values are assigned to cells. With tetrahedral meshes, the number of nodes is less than the number of cells, and the system to solve will be smaller if slowness values are assigned to the nodes.

ALGORITHMS

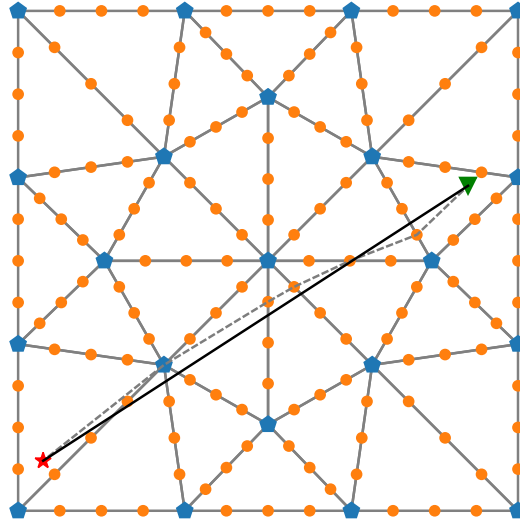
`ttcrpy` contains implementations of three raytracing algorithms.

3.1 Shortest-Path

In the shortest path method (SPM), a grid of nodes is used to build a graph by connecting each node to its neighbours. The connections within the graph are assigned a length equal to the traveltime along it. Hence, by virtue of Fermat's principle which states that a seismic ray follows the minimum traveltime curve, the shortest path between two points within the graph can be seen as an approximation of the raypath.

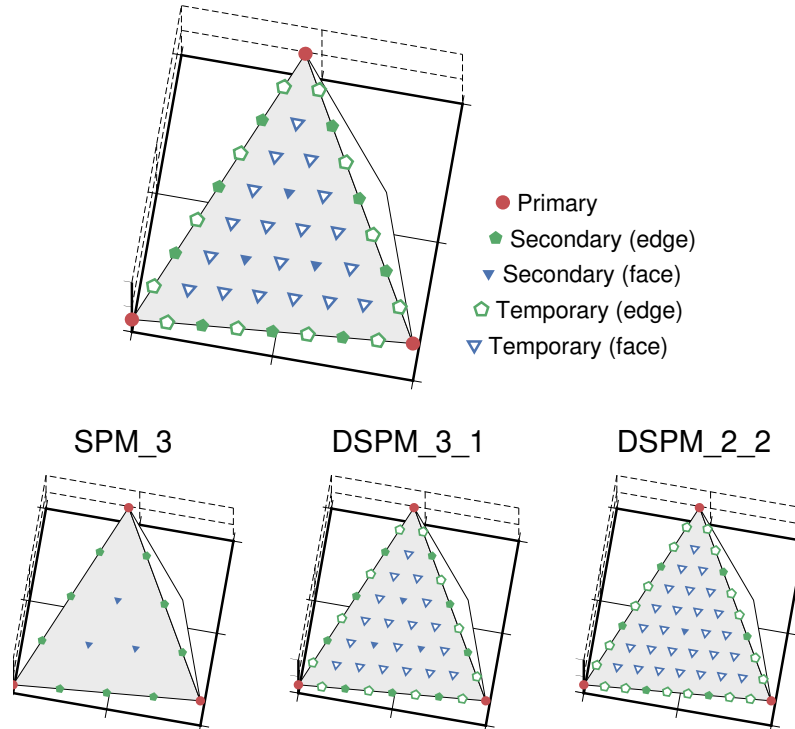
The SPM algorithm proceeds as follows. After construction of the graph, all nodes are initialized to infinite time except the source nodes which are assigned their "time zero" values. A priority queue is then created and all source nodes are pushed into it. Priority queues are a type of container specifically designed such that its first element is always the one with highest priority, according to some strict weak ordering condition. In our case, the highest priority is attributed to the node having the smallest traveltime value. The traveltime is computed for all nodes connected to the earliest source node, the traveltime value at those nodes is updated with their new value, the parent node is set to the source node, and these nodes then are pushed into the queue. Then, the node with highest priority is popped from the queue, and the traveltime is computed at all nodes connected to it except the node parent. The traveltime and parent values are updated if the traveltime is lower than the one previously assigned, and the nodes not already in the queue are pushed in. This process is repeated until the queue is empty.

One particular aspect of the `ttcrpy` implementation is the concept of primary and secondary nodes. Primary nodes are located at the vertexes of the cells, and secondary nodes are surrounding the cells on the edges and faces. In 2D, only secondary edge nodes are introduced. Using secondary nodes allows improving the accuracy and angular coverage of the discrete raypaths. The raypath, however, is an approximation which may deviate from the true raypath, as shown in the figure below which illustrates the case for a homogeneous model.



3.2 Dynamic Shortest-Path

Using secondary nodes can be memory and computationally demanding in 3D. With the dynamic variant of the Shortest-Path, the density of secondary nodes is intentionally set to a low value, and tertiary nodes are added to increase the density in the vicinity of the source.



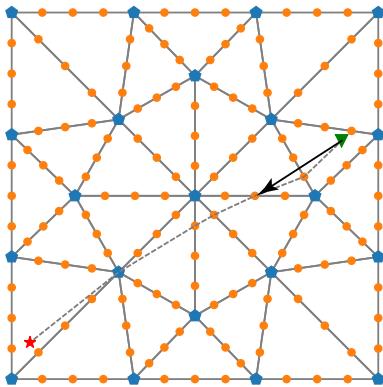
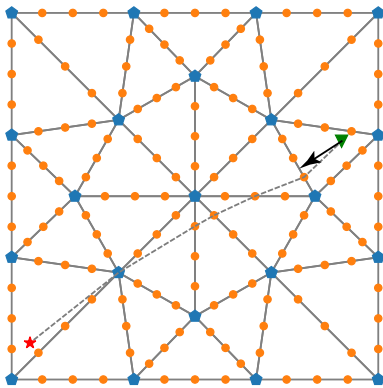
In `ttcrpy`, tertiary nodes are placed within a sphere centered on the source. Tests have shown that a radius of about three times the mean cell edge length provides a good compromise between accuracy and computation time.

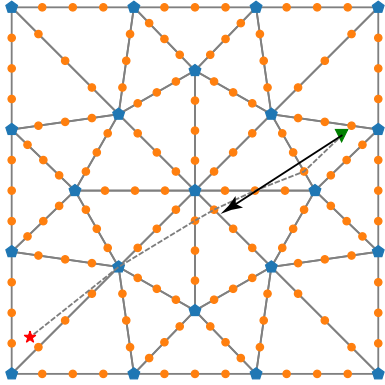
3.3 Fast-Sweeping

The Fast-Sweeping Method avoids the requirement to maintain a sorted list of nodes which can be time consuming and resource intensive. The method relies on Gauss-Seidel iterations to propagate the wave front. At each iteration, all the domain nodes are visited and convergence is reached for nodes along characteristic curves parallel to sweeping directions. The causality is ensured by using several Gauss-Seidel iterations with different directions so that all characteristic curves are scanned.

3.3.1 Raypath computation

Contrary to the SPM and DSPM, the FSM algorithm does not store raypath segments in memory. When raypaths are needed, they must be computed in a second step. The approach implemented in `ttcrpy` is to follow the steepest travel time gradient, from the receiver to the source, as illustrated in the figure below.





3.4 Computing traveltimes from raypaths

With all three algorithms presented above, traveltimes are computed at all grid nodes. In older version of `ttcrpy`, we used to interpolate traveltimes at the receivers coordinates and return the interpolated values. We have observed however that results are more accurate if traveltimes are computed in a subsequent step, in which raypaths are computed from the gradient of the traveltimes (as is done with the FSM when raypaths are needed), and traveltimes integrated along the raypaths.

Computing traveltime from raypaths is available as an option for the fast-sweeping and dynamic shortest-path methods. Because the computational cost of the second step is small in comparison to computing traveltime at the grid nodes, the option is activated by default in 3D. We have observed however that for some model with very complex velocity distributions, convergence issues might arise with this option activated, and we suggest to use the SPM method in the latter case. By design, SPM implementations do not include that option, and traveltimes and raypaths are always computed with values at grid nodes.

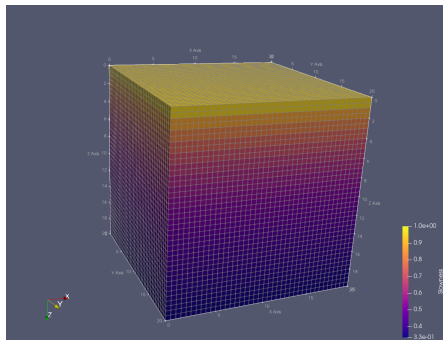
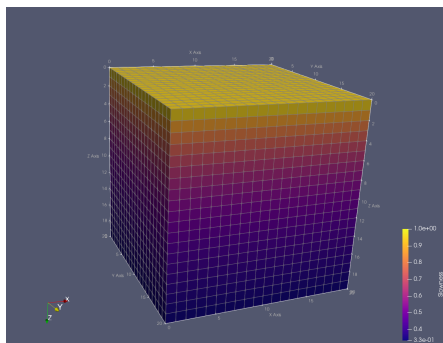
We have also observed that convergence issues arise when sources or receivers are in the cells at the edges of the modeling domain. For that reason, special care should be put when defining input models and parameters.

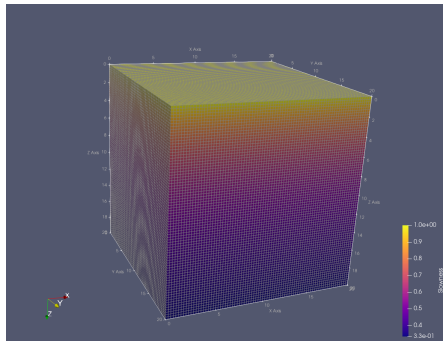
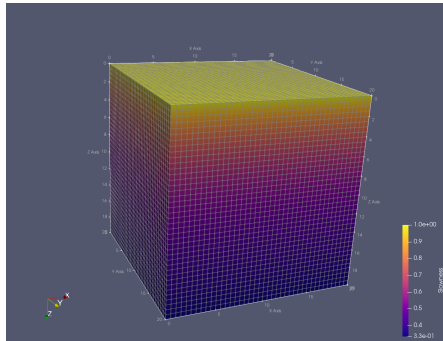
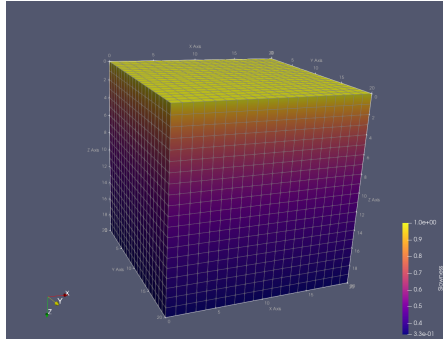
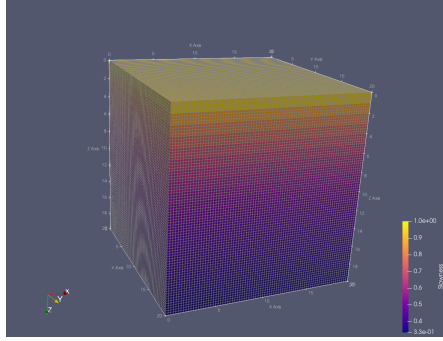
PERFORMANCE

4.1 3D Rectilinear Grids

4.1.1 Models

Performance tests were conducted using two different slowness models: a layer-cake model and a vertical gradient model. Analytic solutions exist for both models, which allows accuracy evaluation. Besides, tests were done for three level of discretization: coarse, medium, and fine. The following figures show the models.





The following figures show the results of the tests. In these figures, models are labelled by two letters: “L” or “G” for layers or gradient, and “C”, “M” or “F” for coarse, medium or fine.

It is important to note that for the layers model, slowness values are assigned to cells, whereas for the gradient model, slowness values are assigned to the nodes of the grid.

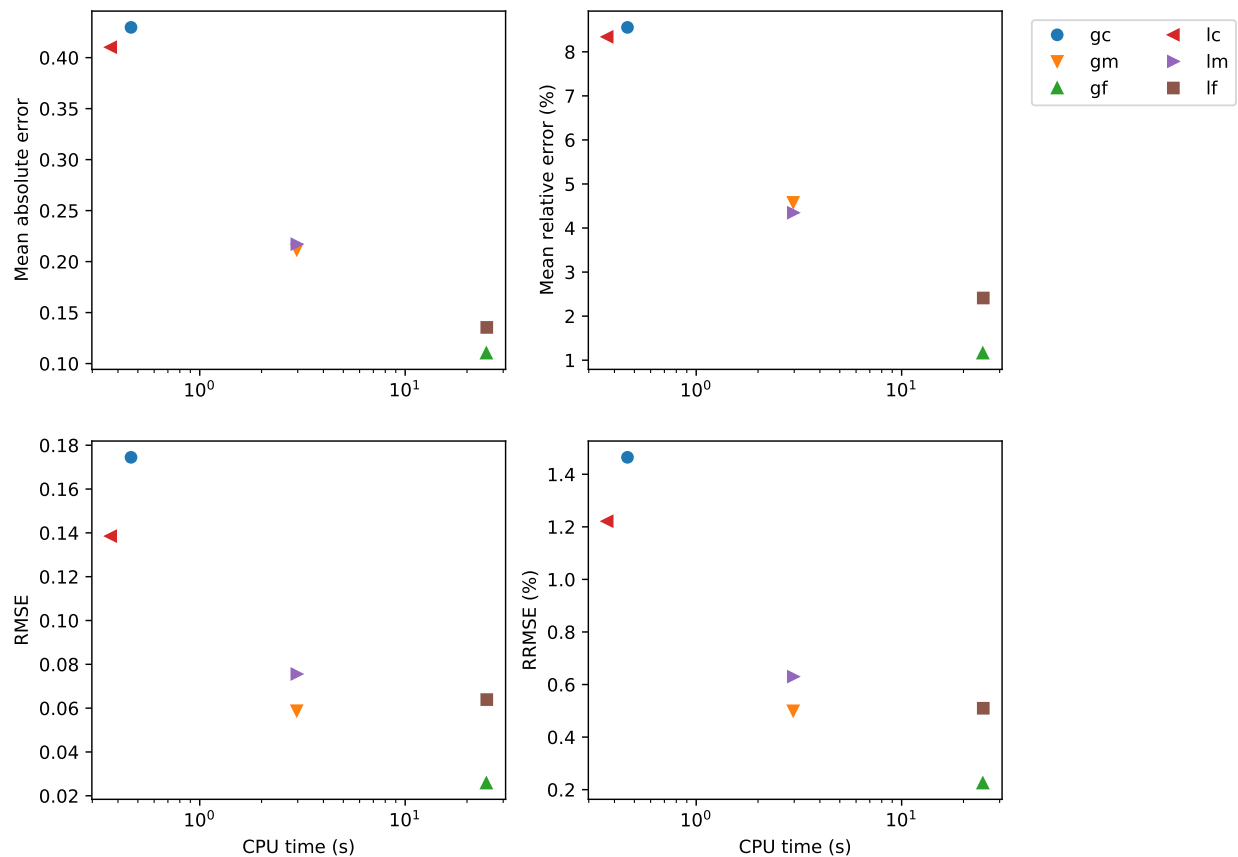
4.1.2 Whole-grid accuracy

In this section, the accuracy of the traveltimes computed over the grid nodes (without using the option to update the traveltimes using the raypaths) is evaluated. Error is computed for nodes for which the coordinates are round numbers.

Fast-Sweeping Method

The results are shown first for the FSM. Accuracy is better for the gradient model, except for the coarse models. In the latter case, cells are too large (as thick as the layers) for the solver to yield satisfying accuracy.

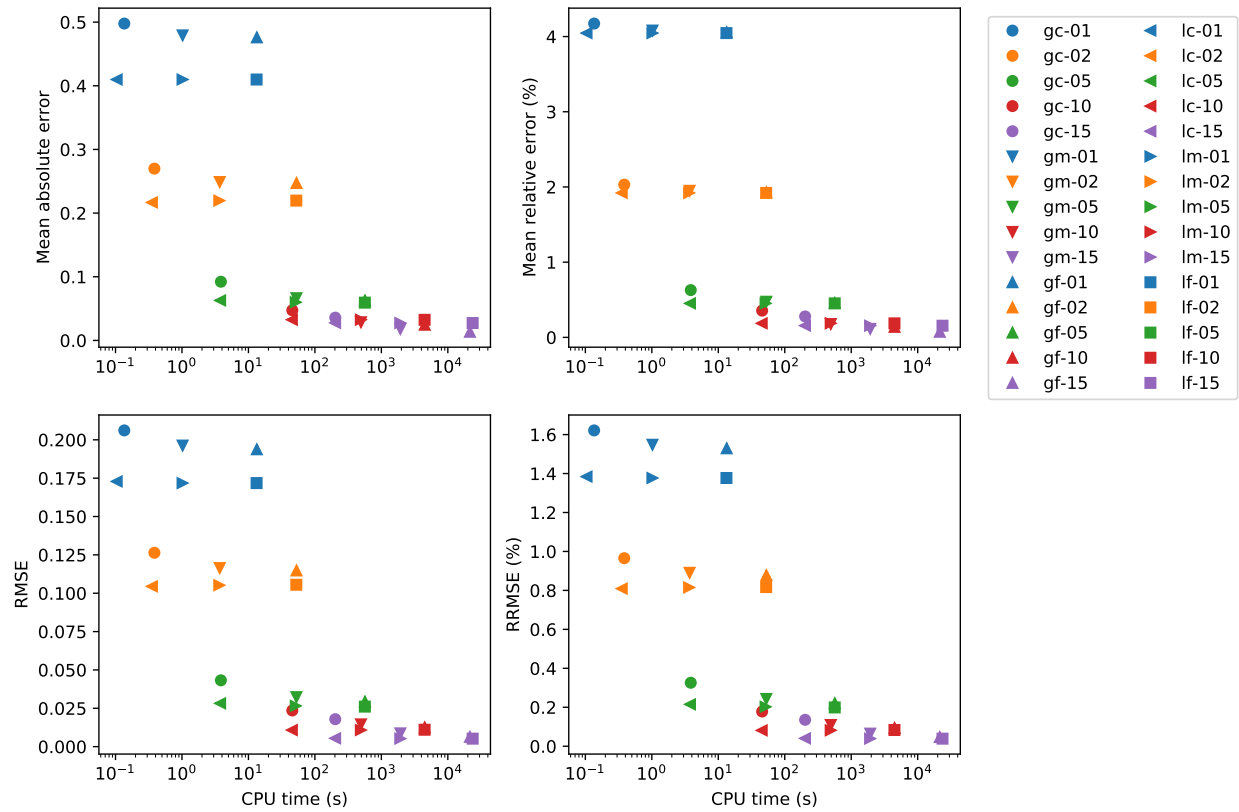
FSM



Shortest-Path Method

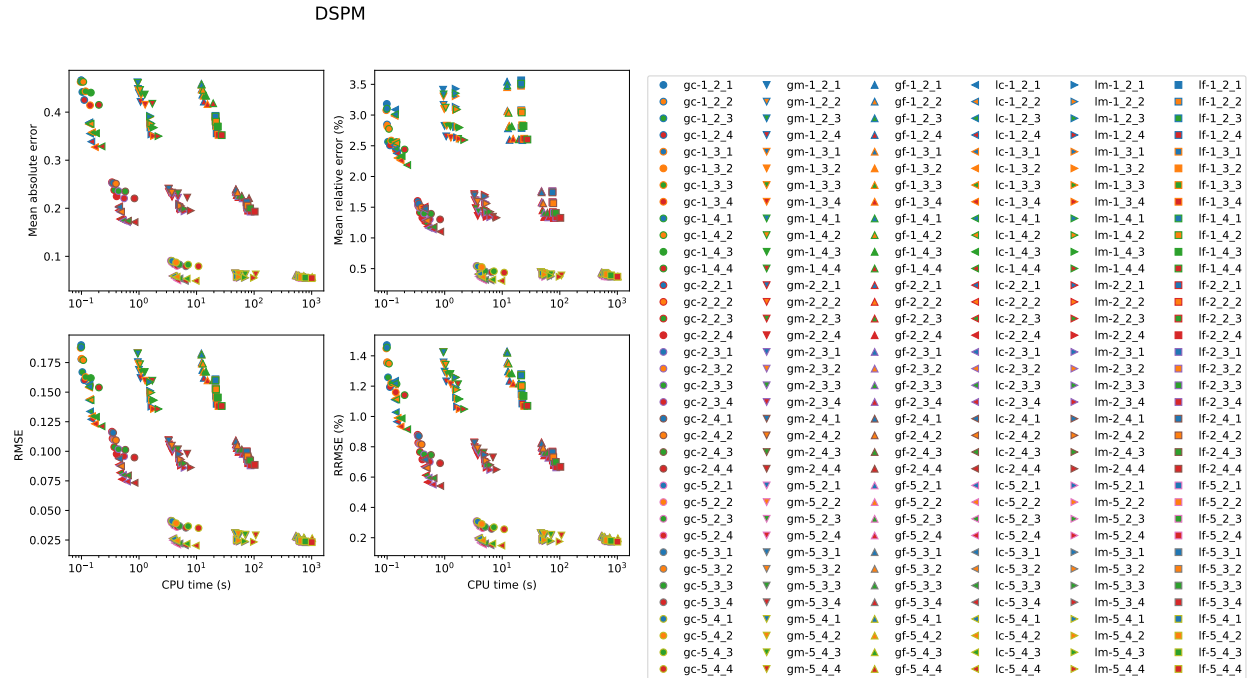
Results for the SPM are shown next. In the legend, the number next to the model label is the number of secondary nodes employed. Increasing this number obviously has an impact on both accuracy and computation time. Using 5 secondary nodes appears to be a good compromise.

SPM



Dynamic Shortest-Path Method

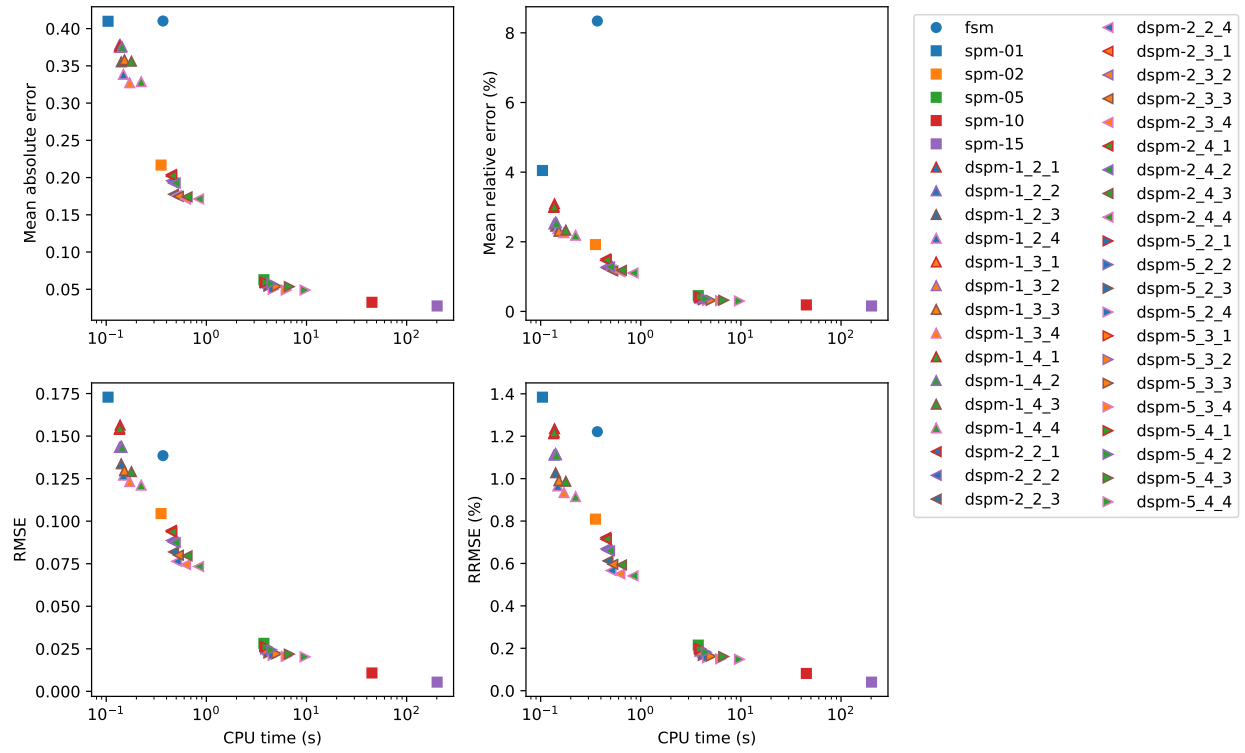
Results for the DSPM are shown next, in a rather busy figure. In the legend, the first number next to the model label is the number of secondary nodes, the second number is the number of tertiary nodes, and the last number is the radius of the sphere containing the tertiary nodes around the source.



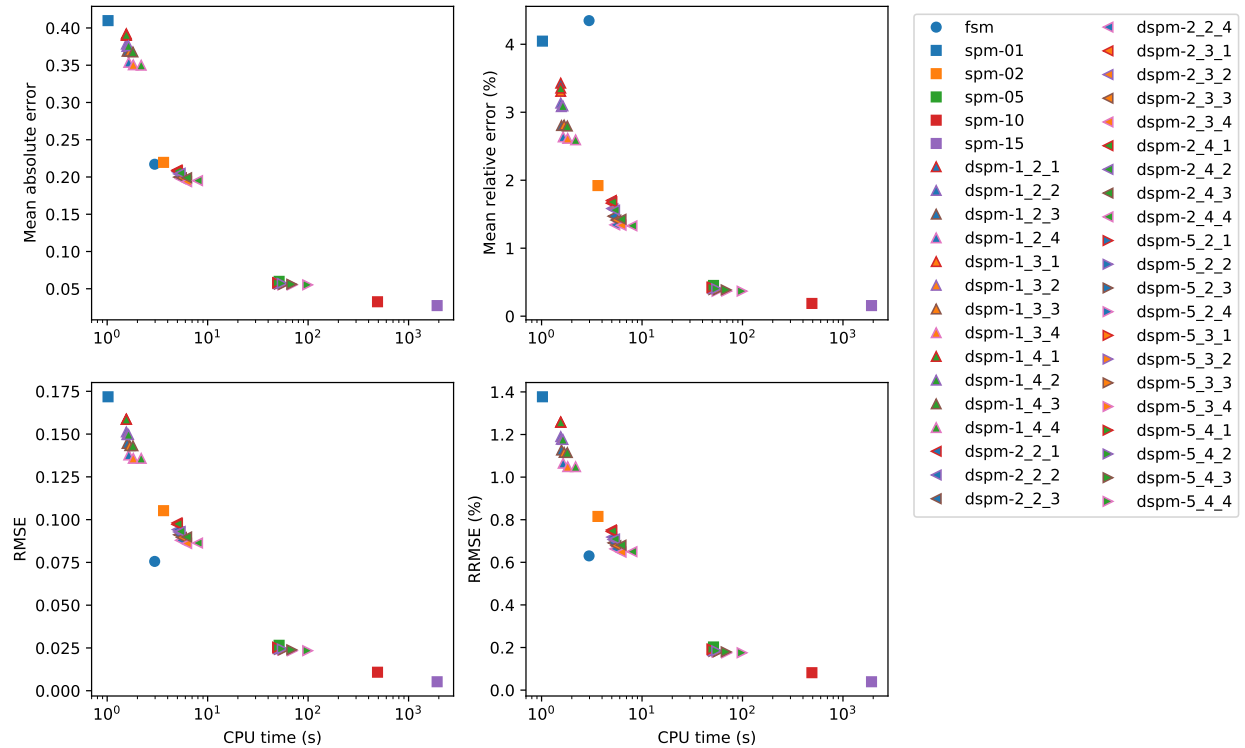
Results by model

The next set of figures contains the accuracy achieved with the three methods for each model. In all cases, the lowest errors are obtained with the SPM with 15 secondary nodes (at the cost of very high computation time). For the gradient model, the FSM is very competitive for the medium and fine models. Otherwise, the DSPM often appears to offer a good compromise.

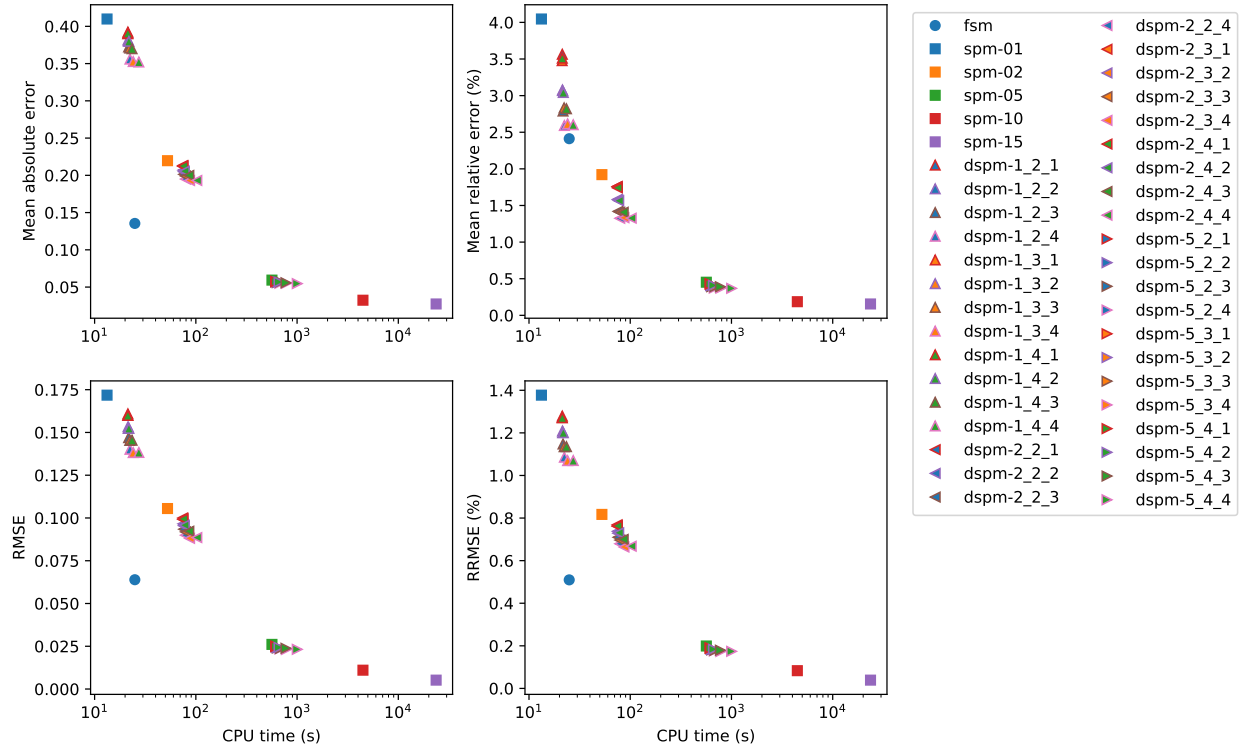
LC



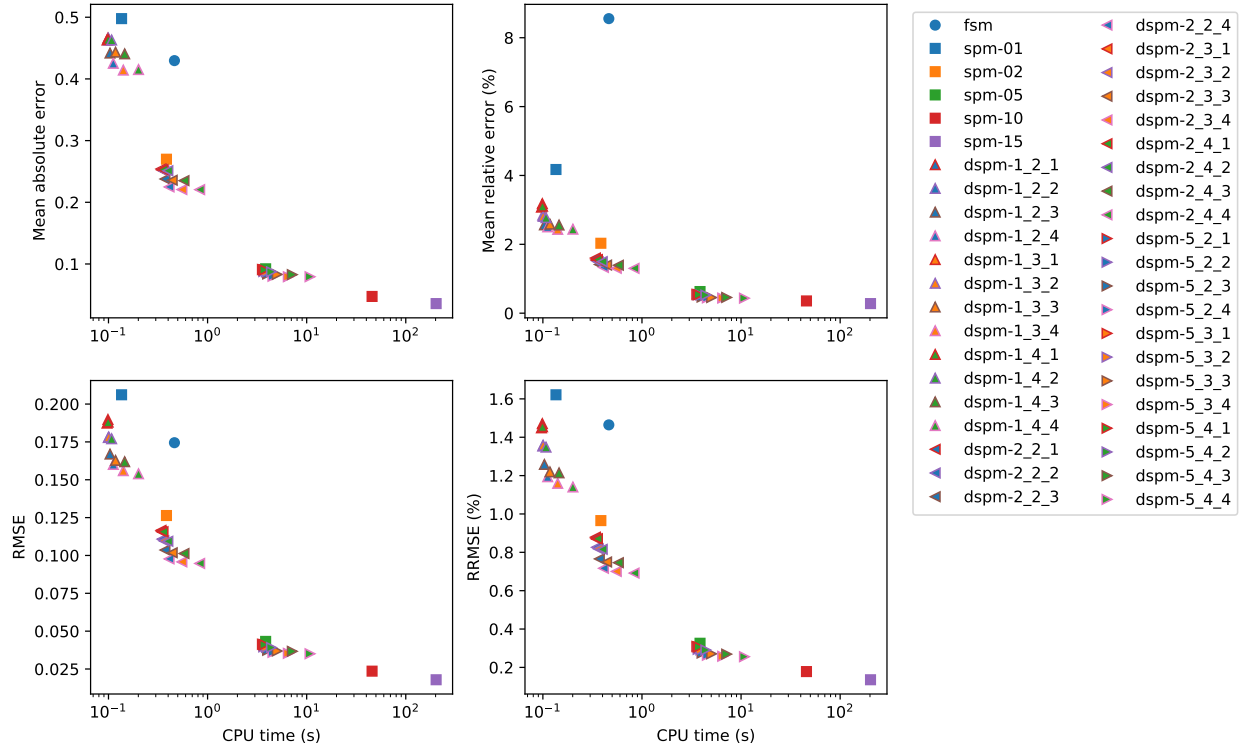
LM



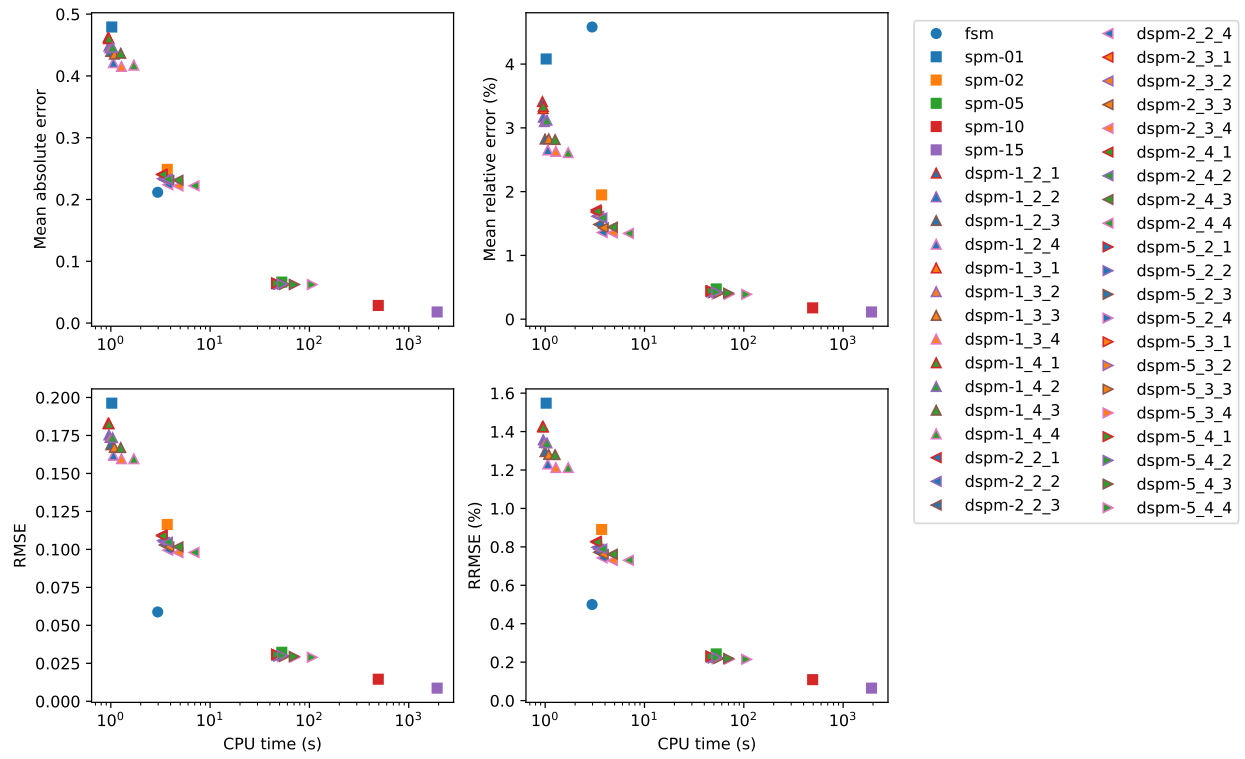
LF



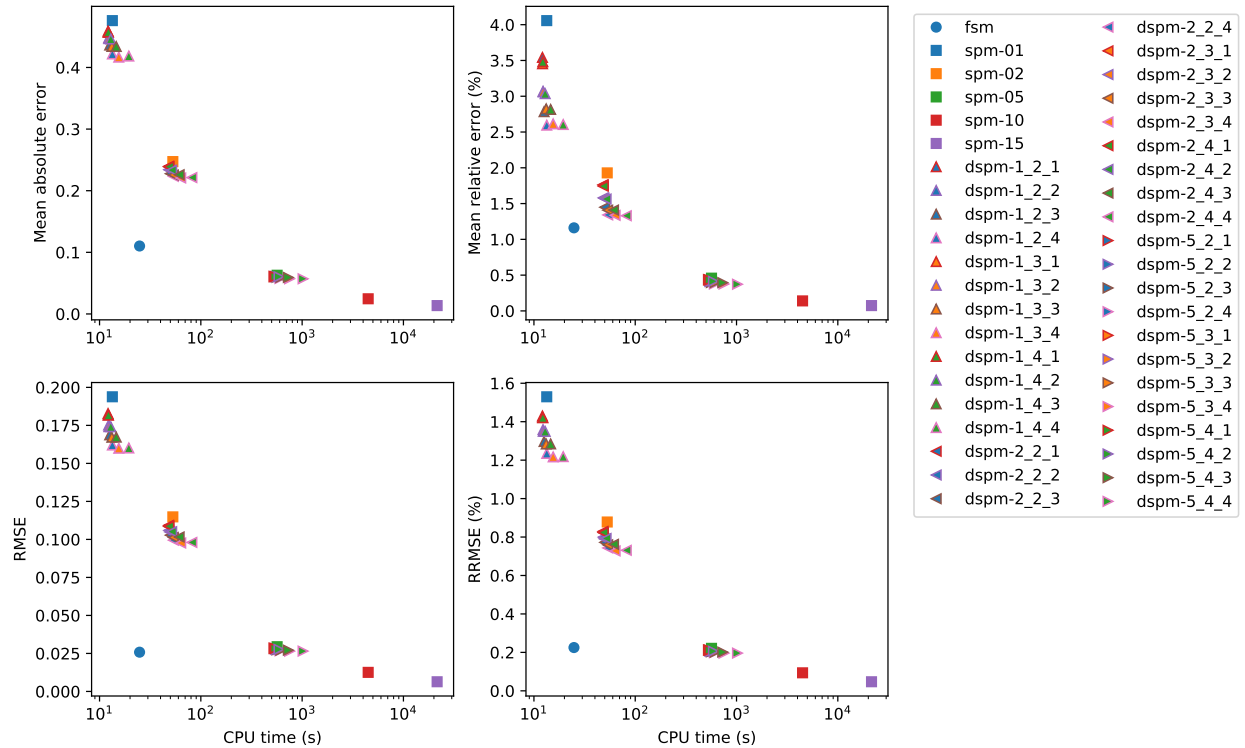
GC



GM



GF



DOCUMENTATION FOR THE PYTHON CODE

The original modules of ttcipy are cgrid2d, cgrid3d, cmesh2d and cmesh3d. These modules are deprecated in favor of rgrid and an upcoming tmesh modules.

5.1 Module rgrid

Raytracing on rectilinear grids

This module contains two classes to perform traveltime computation and raytracing on rectilinear grids:

- *Grid2d* for 2D media
- *Grid3d* for 3D media

Three algorithms are implemented

- the Shortest-Path Method
- the Fast-Sweeping Method
- the Dynamic Shortest-Path Method

Slowness model can be defined in two ways:

- 1) slowness constant within the voxels of the grid (the default)
- 2) slowness defined at nodes of the grid

This code is part of ttcipy (<https://github.com/groupeLIAMG/ttcipy>)

class ttcipy.rgrid.**Grid2d**

class to perform raytracing with 2D rectilinear grids

x

node coordinates along x

Type

np.ndarray

z

node coordinates along z

Type

np.ndarray

dx

node separation along x

Type

float

dz

node separation along z

Type

float

shape

number of parameters along each dimension

Type

(int, int)

nparams

total number of parameters for grid

Type

int

n_threads

number of threads for raytracing

Type

int

Constructor

Grid2d(*x, z, n_threads=1, cell_slowness=1, method='SPM', aniso='iso', eps=1.e-15, maxit=20, weno=1, rotated_template=0, nsnx=10, nsnz=10, n_secondary=3, n_tertiary=3, radius_factor_tertiary=3.0, tt_from_rp=0*) → *Grid2d*

Parameters

- **x** (*np.ndarray*) – node coordinates along x
- **z** (*np.ndarray*) – node coordinates along z
- **n_threads** (*int*) – number of threads for raytracing (default is 1)
- **cell_slowness** (*bool*) – slowness defined for cells (True) or nodes (False) (default is 1)
- **method** (*string*) –
raytracing method (default is SPM)
 - 'FSM' : fast marching method
 - 'SPM' : shortest path method
 - 'DSPM' : dynamic shortest path method
- **aniso** (*string*) –
type of anisotropy (implemented only for the SPM method)
 - 'iso' : isotropic medium
 - 'elliptical' : elliptical anisotropy
 - 'tilted_elliptical' : tilted elliptical anisotropy

- ‘vti_psv’ : vertical transverse isotropy, P and SV waves
- ‘vti_sh’ : vertical transverse isotropy, SH waves
- ‘weakly_anelliptical’ : Weakly-Anelliptical formulation of B. Rommel
- **eps** (*double*) – convergence criterion (FSM) (default is 1e-15)
- **maxit** (*int*) – max number of sweeping iterations (FSM) (default is 20)
- **weno** (*bool*) – use 3rd order weighted essentially non-oscillatory operator (FSM) (default is True)
- **rotated_template** (*bool*) – use rotated templates (FSM)
- **nsnx** (*int*) – number of secondary nodes in x (SPM) (default is 10)
- **nsnz** (*int*) – number of secondary nodes in z (SPM) (default is 10)
- **n_secondary** (*int*) – number of secondary nodes (DSPM) (default is 3)
- **n_tertiary** (*int*) – number of tertiary nodes (DSPM) (default is 3)
- **radius_factor_tertiary** (*double*) – multiplication factor used to compute radius of sphere around source that includes tertiary nodes (DSPM). The radius is the average edge length multiplied by this factor (default is 3)
- **tt_from_rp** (*bool*) – compute traveltimes using raypaths (available for FSM and DSPM only) (default is False)

Notes

For raytracing in anisotropic media, the convention for inputting slowness depends on the model. For elliptical anisotropy, the method `set_slowness` is used to input horizontal slowness, while for weakly anelliptical anisotropy, the method `set_vti_slowness` is used to input vertical slowness.

`compute_D(coord)`

Return matrix of interpolation weights for velocity data points constraint

Parameters

coord (*np.ndarray*, *shape* (*npts*, 2)) – coordinates of data points

Returns

D – Matrix of interpolation weights

Return type

scipy csr_matrix, shape (npts, nparams)

Note: In the current implementation, no check is made to see if the coordinates are on a node, edge, or corner.

`compute_K(order=1)`

Compute smoothing matrices

Parameters

order (*int*) – order of smoothing operator, accept 1 or 2 (1 by default)

Returns

Kx, Kz – matrices for derivatives along x & z

Return type

tuple of csr_matrix

static data_kernel_straight_rays(Tx, Rx, grx, grz, aniso=False) → L

Raytracing with straight rays in 2D

Parameters

- **Tx** (*np.ndarray*) –
source coordinates, nTx by 2
 - 1st column contains X coordinates,
 - 2nd contains Z coordinates
- **Rx** (*np.ndarray*) –
receiver coordinates, nTx by 2
 - 1st column contains X coordinates,
 - 2nd contains Z coordinates
- **grx** (*np.ndarray*) – grid node coordinates along x
- **grz** (*np.ndarray*) – grid node coordinates along z
- **aniso** (*bool*) – compute L for elliptically anisotropic medium (True) or isotropic medium (False)

Returns

L – data kernel matrix (tt = L*slowness)

Return type

scipy csr_matrix

Note: Tx and Rx should contain the same number of rows, each row corresponding to a source-receiver pair

dx

node separation along x

Type

float

dz

node separation along x

Type

float

get_grid_traveltimes(thread_no=0)

Obtain traveltimes computed at primary grid nodes

Parameters**thread_no** (*int*) – thread used to computed traveltimes (default is 0)**Returns**

tt

Return type

np ndarray, shape (nx, nz)

get_number_of_cells()**Returns**

number of cells in grid

Return type

int

get_number_of_nodes()**Returns**

number of nodes in grid

Return type

int

get_s0(hypo, slowness=None)

Return slowness at source points

Parameters

- **hypo** (*np.ndarray with 5 columns*) –
hypo holds source information, i.e.
 - 1st column is event ID number
 - 2nd column is origin time
 - 3rd column is source easting (X)
 - 4th column is source elevation (Z)
- **slowness** (*np ndarray, shape (nx, nz) (optional)*) – slowness at grid nodes or cells (depending on cell_slowness) slowness may also have been flattened (with default 'C' order)

Returns

s0 – slowness at source points

Return type

np.ndarray

get_slowness()

Returns slowness of grid

Returns

slowness

Return type

np ndarray, shape (nx, nz)

is_outside(pts)

Check if points are outside grid

Parameters

pts (*np ndarray, shape(npts, 3)*) – coordinates of points to check

Returns

True if at least one point outside grid

Return type

bool

n_threads

number of threads for raytracing

Type

int

nparams

total number of parameters for grid

Type

int

raytrace(*source*, *rcv*, *slowness=None*, *xi=None*, *theta=None*, *Vp0=None*, *Vs0=None*, *delta=None*, *epsilon=None*, *gamma=None*, *thread_no=None*, *aggregate_src=False*, *compute_L=False*, *return_rays=False*) → tt, rays, L

Perform raytracing

Parameters

- **source** (*2D np.ndarray with 2 or 3 columns*) – see notes below
- **rcv** (*2D np.ndarray with 2 columns*) – Columns correspond to x, y and z coordinates
- **slowness** (*np ndarray, shape (nx, nz) (None by default)*) – slowness at grid nodes or cells (depending on cell_slowness) slowness may also have been flattened (with default ‘C’ order) if None, slowness must have been assigned previously
- **xi** (*np ndarray, shape (nx, nz) (None by default)*) – xi at grid cells (only for SPM & cell_slowness=True) xi may also have been flattened (with default ‘C’ order) if None, xi must have been assigned previously
- **theta** (*np ndarray, shape (nx, nz) (None by default)*) – theta at grid cells (only for SPM & cell_slowness=True) theta may also have been flattened (with default ‘C’ order) if None, theta must have been assigned previously
- **Vp0** (*np ndarray, shape (nx, nz) (None by default)*) – Vp0 at grid cells (only for SPM & cell_slowness=True) Vp0 may also have been flattened (with default ‘C’ order) if None, Vp0 must have been assigned previously
- **Vs0** (*np ndarray, shape (nx, nz) (None by default)*) – Vs0 at grid cells (only for SPM & cell_slowness=True) Vs0 may also have been flattened (with default ‘C’ order) if None, Vs0 must have been assigned previously
- **delta** (*np ndarray, shape (nx, nz) (None by default)*) – delta at grid cells (only for SPM & cell_slowness=True) delta may also have been flattened (with default ‘C’ order) if None, delta must have been assigned previously
- **epsilon** (*np ndarray, shape (nx, nz) (None by default)*) – epsilon at grid cells (only for SPM & cell_slowness=True) epsilon may also have been flattened (with default ‘C’ order) if None, epsilon must have been assigned previously
- **gamma** (*np ndarray, shape (nx, nz) (None by default)*) – gamma at grid cells (only for SPM & cell_slowness=True) gamma may also have been flattened (with default ‘C’ order) if None, gamma must have been assigned previously
- **thread_no** (*int (None by default)*) – Perform calculations in thread number “thread_no” if None, attempt to run in parallel if warranted by number of sources and value of n_threads in constructor
- **aggregate_src** (*bool (False by default)*) – if True, all source coordinates belong to a single event

- **compute_L** (*bool (False by default)*) – Compute matrices of partial derivative of travel time w/r to slowness
- **return_rays** (*bool (False by default)*) – Return raypaths

Returns

- **tt** (*np.ndarray*) – travel times for the appropriate source-rcv (see Notes below)
- **rays** (*list of np.ndarray*) – Coordinates of segments forming raypaths (if `return_rays` is True)
- **L** (*scipy csr_matrix*) – Matrix of partial derivative of travel time w/r to slowness

Notes

If source has 2 columns:

- Columns correspond to x and z coordinates
- Origin time (t0) is 0 for all points

If source has 3 columns:

- 1st column corresponds to origin times
- 2nd & 3rd columns correspond to x and z coordinates

source and rcv can contain the same number of rows, each row corresponding to a source-receiver pair, or the number of rows may differ if `aggregate_src` is True or if all rows in source are identical.

set_Vp0(*v*)

Assign vertical Vp to grid (VTI medium)

Parameters

v (*np ndarray, shape (nx, nz)*) – v may also have been flattened (with default 'C' order)

set_Vs0(*v*)

Assign vertical Vs to grid (VTI medium)

Parameters

v (*np ndarray, shape (nx, nz)*) – v may also have been flattened (with default 'C' order)

set_delta(*d*)

Assign Thomsen delta parameter to grid (VTI medium, P-SV waves)

Parameters

d (*np ndarray, shape (nx, nz)*) – d may also have been flattened (with default 'C' order)

set_epsilon(*e*)

Assign Thomsen epsilon parameter to grid (VTI medium, P-SV waves)

Parameters

e (*np ndarray, shape (nx, nz)*) – e may also have been flattened (with default 'C' order)

set_gamma(*g*)

Assign Thomsen gamma parameter to grid (VTI medium, SH waves)

Parameters

g (*np ndarray*, *shape (nx, nz)*) – g may also have been flattened (with default ‘C’ order)

set_s2(*g*)

Assign weakly anelliptical parameter s2

Parameters

g (*np ndarray*, *shape (nx, nz)*) – g may also have been flattened (with default ‘C’ order)

set_s4(*g*)

Assign weakly anelliptical parameter s4

Parameters

g (*np ndarray*, *shape (nx, nz)*) – g may also have been flattened (with default ‘C’ order)

set_slowness(*slowness*)

Assign slowness to grid

Parameters

slowness (*np ndarray*, *shape (nx, nz)*) – slowness may also have been flattened (with default ‘C’ order)

set_tilt_angle(*theta*)

Assign tilted elliptical anisotropy angle to grid

Parameters

theta (*np ndarray*, *shape (nx, nz)*) – theta may also have been flattened (with default ‘C’ order)

set_traveltime_from_raypath(*ttrp*)

Set option to compute traveltime using raypath

Parameters

ttrp (*bool*) – option value

set_use_thread_pool(*use_thread_pool*)

Set option to use thread pool instead of parallel loop

Parameters

use_thread_pool (*bool*) – option value

set_velocity(*velocity*)

Assign velocity to grid

Parameters

velocity (*np ndarray*, *shape (nx, nz)*) – velocity may also have been flattened (with default ‘C’ order)

set_xi(*xi*)

Assign elliptical anisotropy ratio to grid

Parameters

xi (*np ndarray*, *shape (nx, nz)*) – xi may also have been flattened (with default ‘C’ order)

shape

number of parameters along each dimension

Type

list of int

to_vtk(*fields*, *filename*)

Save grid variables and/or raypaths to VTK format

Parameters

- **fields** (*dict*) – dict of variables to save to file. Variables should be np.ndarray of size equal to either the number of nodes of the number of cells of the grid, or a list of raypath coordinates.
- **filename** (*str*) – Name of file without extension for saving (extension vtr will be added). Raypaths are saved in separate files, and filename will be appended by the dict key and have a vtp extension.

Notes

VTK files can be visualized with Paraview (<https://www.paraview.org>)

x

node coordinates along x

Type

np.ndarray

z

node coordinates along z

Type

np.ndarray

class ttrcpy.rgrid.**Grid3d**

class to perform raytracing with 3D rectilinear grids

x

node coordinates along x

Type

np.ndarray

y

node coordinates along y

Type

np.ndarray

z

node coordinates along z

Type

np.ndarray

dx

node separation along x

Type
float

dy
node separation along y

Type
float

dz
node separation along z

Type
float

shape
number of parameters along each dimension

Type
(int, int, int)

nparams
total number of parameters for grid

Type
int

n_threads
number of threads for raytracing

Type
int

Constructor

Grid3d(*x, y, z, n_threads=1, cell_slowness=1, method='FSM', tt_from_rp=1, interp_vel=0, eps=1.e-15, maxit=20, weno=1, nsnx=5, nsny=5, nsnz=5, n_secondary=2, n_tertiary=2, radius_factor_tertiary=3.0, translate_grid=False*) → *Grid3d*

Parameters

- **x** (*np.ndarray*) – node coordinates along x
- **y** (*np.ndarray*) – node coordinates along y
- **z** (*np.ndarray*) – node coordinates along z
- **n_threads** (*int*) – number of threads for raytracing (default is 1)
- **cell_slowness** (*bool*) – slowness defined for cells (True) or nodes (False) (default is 1)
- **method** (*string*) –
raytracing method (default is FSM)
 - 'FSM' : fast marching method
 - 'SPM' : shortest path method
 - 'DSPM' : dynamic shortest path
- **tt_from_rp** (*bool*) – compute traveltimes from raypaths (FSM or DSPM only) (default is 1)

- **interp_vel** (*bool*) – interpolate velocity instead of slowness at nodes (for cell_slowness == False or FSM) (default is False)
- **eps** (*double*) – convergence criterion (FSM) (default is 1e-15)
- **maxit** (*int*) – max number of sweeping iterations (FSM) (default is 20)
- **weno** (*bool*) – use 3rd order weighted essentially non-oscillatory operator (FSM) (default is True)
- **nsnx** (*int*) – number of secondary nodes in x (SPM) (default is 5)
- **nsny** (*int*) – number of secondary nodes in y (SPM) (default is 5)
- **nsnz** (*int*) – number of secondary nodes in z (SPM) (default is 5)
- **n_secondary** (*int*) – number of secondary nodes (DSPM) (default is 2)
- **n_tertiary** (*int*) – number of tertiary nodes (DSPM) (default is 2)
- **radius_factor_tertiary** (*double*) – multiplication factor used to compute radius of sphere around source that includes tertiary nodes (DSPM). The radius is the average edge length multiplied by this factor (default is 3)
- **translate_grid** (*bool*) – Translate the grid such that origin is (0, 0, 0) to perform computations, which may increase accuracy when large values, e.g. UTM coordinates, are used. When raytracing, src and rcv should be given in the original system, and output raypath coordinates are also given in the original system (default if False)

static builder(*filename, n_threads=1, method='FSM', tt_from_rp=1, interp_vel=0, eps=1.e-15, maxit=20, weno=1, nsnx=5, nsny=5, nsnz=5, n_secondary=2, n_tertiary=2, radius_factor_tertiary=3.0, translate_grid=0*)

Build instance of Grid3d from VTK file

Parameters

- **filename** (*str*) – Name of file holding a vtkRectilinearGrid. The grid must have point or cell attribute named either 'Slowness', 'slowness', 'Velocity', 'velocity', or 'P-wave velocity'
- **Constructor** (*Other parameters are defined in*) –

Returns

grid – grid instance

Return type

Grid3d

compute_D(*coord*)

Return matrix of interpolation weights for velocity data points constraint

Parameters

coord (*np.ndarray, shape (npts, 3)*) – coordinates of data points

Returns

D – Matrix of interpolation weights

Return type

scipy csr_matrix, shape (npts, nparams)

Note: In the current implementation, no check is made to see if the coordinates are on a node, edge, face, or corner.

compute_K()

Compute smoothing matrices (2nd order derivative)

Returns

Kx, Ky, Kz – matrices for derivatives along x, y, & z

Return type

tuple of *csr_matrix*

static data_kernel_straight_rays(*Tx, Rx, grx, gry, grz, centers*) -> *L, (xc, yc, zc)*

Raytracing with straight rays in 3D

Parameters

- **Tx** (*np.ndarray*) –
source coordinates, nTx by 3
 - 1st column contains X coordinates,
 - 2nd contains Y coordinates
 - 3rd contains Z coordinates
- **Rx** (*np.ndarray*) –
receiver coordinates, nTx by 3
 - 1st column contains X coordinates,
 - 2nd contains Y coordinates
 - 3rd contains Z coordinates
- **grx** (*np.ndarray*) – grid node coordinates along x
- **gry** (*np.ndarray*) – grid node coordinates along y
- **grz** (*np.ndarray*) – grid node coordinates along z
- **centers** (*bool*) – return coordinates of center of cells (False by default)

Returns

- **L** (*scipy csr_matrix*) – data kernel matrix ($tt = L * \text{slowness}$)
- **(xc, yc, zc)** (tuple of *np.ndarray*) – vectors of coordinates of center of cells

Note: Tx and Rx should contain the same number of rows, each row corresponding to a source-receiver pair

dx

node separation along x

Type

float

dy

node separation along y

Type

float

dz

node separation along z

Type

float

get_grid_traveltimes(*thread_no=0*)

Obtain traveltimes computed at primary grid nodes

Parameters

thread_no (*int*) – thread used to computed traveltimes (default is 0)

Returns

tt – traveltimes

Return type

np ndarray, shape (nx, ny, nz)

get_number_of_cells()

Returns

number of cells in grid

Return type

int

get_number_of_nodes()

Returns

number of nodes in grid

Return type

int

get_s0(*hypo, slowness=None*)

Return slowness at source points

Parameters

- **hypo** (*np.ndarray with 5 columns*) –

hypo holds source information, i.e.

- 1st column is event ID number
- 2nd column is origin time
- 3rd column is source easting
- 4th column is source northing
- 5th column is source elevation

- **slowness** (*np ndarray, shape (nx, ny, nz) (optional)*) – slowness at grid nodes or cells (depending on cell_slowness) slowness may also have been flattened (with default 'C' order)

Returns

s0 – slowness at source points

Return type

np.ndarray

get_slowness()

Returns slowness of grid

Returns

slowness

Return type

np ndarray, shape (nx, ny, nz)

ind(*i, j, k*)

Return node index

Parameters

- **i** (*int*) – index of node along x
- **j** (*int*) – index of node along y
- **k** (*int*) – index of node along z

Returns

node index for a “flattened” grid

Return type

int

indc(*i, j, k*)

return cell index

Parameters

- **i** (*int*) – index of cell along x
- **j** (*int*) – index of cell along y
- **k** (*int*) – index of cell along z

Returns

cell index for a “flattened” grid

Return type

int

is_outside(*pts*)

Check if points are outside grid

Parameters

pts (*np ndarray, shape(npts, 3)*) – coordinates of points to check

Returns

True if at least one point outside grid

Return type

bool

n_threads

number of threads for raytracing

Type

int

nparams

total number of parameters for grid

Type
int

raytrace(*source*, *rcv*, *slowness=None*, *thread_no=None*, *aggregate_src=False*, *compute_L=False*, *compute_M=False*, *return_rays=False*)

raytrace(*source*, *rcv*, *slowness=None*, *thread_no=None*, *aggregate_src=False*, *compute_L=False*, *compute_M=False*, *return_rays=False*) -> *tt*, *rays*, *M*, *L*

Perform raytracing

Parameters

- **source** (*2D np.ndarray with 3, 4 or 5 columns*) – see notes below
- **rcv** (*2D np.ndarray with 3 columns*) – Columns correspond to x, y and z coordinates
- **slowness** (*np ndarray, shape (nx, ny, nz) (None by default)*) – slowness at grid nodes or cells (depending on cell_slowness) slowness may also have been flattened (with default ‘C’ order) if None, slowness must have been assigned previously
- **thread_no** (*int (None by default)*) – Perform calculations in thread number “thread_no” if None, attempt to run in parallel if warranted by number of sources and value of n_threads in constructor
- **aggregate_src** (*bool (False by default)*) – if True, all source coordinates belong to a single event
- **compute_L** (*bool (False by default)*) –
Compute matrices of partial derivative of travel time w/r to slowness (implemeted for the SPM & DSPM with slowness defined at cells).
- **compute_M** (*bool (False by default)*) – Compute matrices of partial derivative of travel time w/r to velocity Note : compute_M and compute_L are mutually exclusive
- **return_rays** (*bool (False by default)*) – Return raypaths

Returns

- **tt** (*np.ndarray*) – travel times for the appropriate source-rcv (see Notes below)
- **rays** (*list of np.ndarray*) – Coordinates of segments forming raypaths (if return_rays is True)
- **M** (*list of csr_matrix*) – matrices of partial derivative of travel time w/r to velocity. the number of matrices is equal to the number of sources
- **L** (*scipy csr_matrix*) – Matrix of partial derivative of travel time w/r to slowness. if input argument source has 5 columns, L is a list of matrices and the number of matrices is equal to the number of sources otherwise, L is a single csr_matrix

Notes

If source has 3 columns:

- Columns correspond to x, y and z coordinates
- Origin time (t0) is 0 for all points

If source has 4 columns:

- 1st column corresponds to origin times
- 2nd, 3rd & 4th columns correspond to x, y and z coordinates

If source has 5 columns:

- 1st column corresponds to event ID
- 2nd column corresponds to origin times
- 3rd, 4th & 5th columns correspond to x, y and z coordinates

For the latter case (5 columns), source and rcv should contain the same number of rows, each row corresponding to a source-receiver pair. For the 2 other cases, source and rcv can contain the same number of rows, each row corresponding to a source-receiver pair, or the number of rows may differ if `aggregate_src` is True or if all rows in source are identical.

set_slowness(*slowness*)

Assign slowness to grid

Parameters

slowness (*np ndarray*, *shape (nx, ny, nz)*) – slowness may also have been flattened (with default ‘C’ order)

set_traveltime_from_raypath(*ttrp*)

Set option to compute traveltimes using raypath

Parameters

ttrp (*bool*) – option value

set_use_thread_pool(*use_thread_pool*)

Set option to use thread pool instead of parallel loop

Parameters

use_thread_pool (*bool*) – option value

set_velocity(*velocity*)

Assign velocity to grid

Parameters

velocity (*np ndarray*, *shape (nx, ny, nz)*) – velocity may also have been flattened (with default ‘C’ order)

shape

number of parameters along each dimension

Type

list of int

to_vtk(*fields, filename*)

Save grid variables and/or raypaths to VTK format

Parameters

- **fields** (*dict*) – dict of variables to save to file. Variables should be np.ndarray of size equal to either the number of nodes or the number of cells of the grid, or a list of raypath coordinates.
- **filename** (*str*) – Name of file without extension for saving (extension vtr will be added). Raypaths are saved in separate files, and filename will be appended by the dict key and have a vtp extension.

Notes

VTK files can be visualized with Paraview (<https://www.paraview.org>)

x

node coordinates along x

Type

np.ndarray

y

node coordinates along y

Type

np.ndarray

z

node coordinates along z

Type

np.ndarray

`ttrcpy.rgrid.set_verbose(v)`

Set verbosity level for C++ code

Parameters

v (*int*) – verbosity level

5.2 Module tmesh

Raytracing on unstructured triangular and tetrahedral meshes

This module contains two classes to perform traveltime computation and raytracing on unstructured meshes:

- *Mesh2d* for 2D media
- *Mesh3d* for 3D media

Three algorithms are implemented

- the Shortest-Path Method
- the Fast-Sweeping Method
- the Dynamic Shortest-Path Method

Slowness model can be defined in two ways:

- 1) slowness constant within the voxels of the mesh (the default)
- 2) slowness defined at nodes of the mesh

This code is part of ttr (<https://github.com/groupeLIAMG/ttr>)

class `ttrcpy.tmesh.Mesh2d`

class to perform raytracing with triangular meshes

nparams

total number of parameters for grid

Type

int

n_threads

number of threads for raytracing

Type

int

Constructor

Mesh2d(*nodes, triangles, n_threads=1, cell_slowness=1, method='FSM', aniso='iso', eps=1e-15, maxit=20, process_obtuse=1, n_secondary=5, n_tertiary=2, radius_factor_tertiary=2, tt_from_rp=0*) → *Mesh2d*

Parameters

- **nodes** (*np.ndarray, shape (nnodes, 2)*) – node coordinates
- **triangles** (*np.ndarray of int, shape (ntriangles, 3)*) – indices of nodes forming the triangles
- **n_threads** (*int*) – number of threads for raytracing (default is 1)
- **cell_slowness** (*bool*) – slowness defined for cells (True) or nodes (False) (default is 1)
- **method** (*string*) –
raytracing method (default is FSM)
 - 'FSM' : fast marching method
 - 'SPM' : shortest path method
 - 'DSPM' : dynamic shortest path
- **aniso** (*string*) –
type of anisotropy (implemented only for the SPM method)
 - 'iso' : isotropic medium
 - 'elliptical' : elliptical anisotropy
 - 'tilted_elliptical' : tilted elliptical anisotropy
 - 'weakly_anelliptical' : Weakly-Anelliptical formulation of B. Rommel
- **eps** (*double*) – convergence criterion (FSM) (default is 1e-15)
- **maxit** (*int*) – max number of sweeping iterations (FSM) (default is 20)
- **process_obtuse** (*bool*) – use method of Qian et al (2007) to improve accuracy for triangles with obtuse angle (default is True)
- **n_secondary** (*int*) – number of secondary nodes (SPM) (default is 5)
- **n_tertiary** (*int*) – number of tertiary nodes (DSPM) (default is 2)
- **radius_factor_tertiary** (*double*) – multiplication factor used to compute radius of sphere around source that includes tertiary nodes (DSPM). The radius is the average edge length multiplied by this factor (default is 2)

- **tt_from_rp** (*bool*) – compute traveltimes using raypaths (default is False)

Notes

For raytracing in anisotropic media, the convention for inputting slowness depends on the model. For elliptical anisotropy, the method *set_slowness* is used to input horizontal slowness, while for weakly anelliptical anisotropy, the method *is* used to input vertical slowness.

static builder(*filename, n_threads, cell_slowness, method, eps, maxit, process_obtuse, n_secondary, n_tertiary, radius_factor_tertiary, tt_from_rp*)

Build instance of Mesh2d from VTK file

Parameters

- **filename** (*str*) – Name of file holding a vtkUnstructuredGrid. The grid must have point or cell attribute named either ‘Slowness’, ‘slowness’, ‘Velocity’, ‘velocity’, or ‘P-wave velocity’. All cells must be of type vtkTriangle
- **Constructor** (*Other parameters are defined in*) –

Returns

mesh – mesh instance

Return type

Mesh2d

get_grid_traveltimes(*thread_no=0*)

Obtain traveltimes computed at primary grid nodes

Parameters

thread_no (*int*) – thread used to computed traveltimes (default is 0)

Returns

tt – traveltimes

Return type

np ndarray, shape (nnodes,)

get_number_of_cells()

Returns

number of cells in grid

Return type

int

get_number_of_nodes()

Returns

number of nodes in grid

Return type

int

n_threads

number of threads for raytracing

Type

int

nparams

total number of parameters for mesh

Type

int

raytrace(*source*, *rcv*, *slowness=None*, *thread_no=None*, *aggregate_src=False*, *compute_L=False*, *return_rays=False*) → *tt*, *rays*

Perform raytracing

Parameters

- **source** (*2D np.ndarray with 2 or 3 columns*) – see notes below
- **rcv** (*2D np.ndarray with 2 columns*) – Columns correspond to x and z coordinates
- **slowness** (*np ndarray, (None by default)*) – slowness at grid nodes or cells (depending on cell_slowness) if None, slowness must have been assigned previously
- **thread_no** (*int (None by default)*) – Perform calculations in thread number “thread_no” if None, attempt to run in parallel if warranted by number of sources and value of n_threads in constructor
- **aggregate_src** (*bool (False by default)*) – if True, all source coordinates belong to a single event
- **compute_L** (*bool (False by default)*) – Compute matrices of partial derivative of travel time w/r to slowness
- **return_rays** (*bool (False by default)*) – Return raypaths

Returns

- **tt** (*np.ndarray*) – travel times for the appropriate source-rcv (see Notes below)
- **rays** (*list of np.ndarray*) – Coordinates of segments forming raypaths (if return_rays is True)

Notes**If source has 2 columns:**

- Columns correspond to x and z coordinates
- Origin time (t0) is 0 for all points

If source has 3 columns:

- 1st column corresponds to origin times
- 2nd & 3rd columns correspond to x and z coordinates

source and rcv can contain the same number of rows, each row corresponding to a source-receiver pair, or the number of rows may differ if aggregate_src is True or if all rows in source are identical.

set_s2(s2)

Assign energy-velocity parameter s2 to grid

Parameters

s2 (*np ndarray, shape (nparams,)*) –

set_s4(*s4*)

Assign energy-velocity parameter s4 to grid

Parameters

s4 (*np ndarray, shape (nparams,)*) –

set_slowness(*slowness*)

Assign slowness to grid

Parameters

slowness (*np ndarray, shape (nparams,)*) –

set_tilt_angle(*theta*)

Assign tilted elliptical anisotropy angle to grid

Parameters

theta (*np ndarray, shape (nparams,)*) –

set_traveltime_from_raypath(*ttrp*)

Set option to compute traveltime using raypath

Parameters

ttrp (*bool*) – option value

set_use_thread_pool(*use_thread_pool*)

Set option to use thread pool instead of parallel loop

Parameters

use_thread_pool (*bool*) – option value

set_velocity(*velocity*)

Assign velocity to grid

Parameters

velocity (*np ndarray, shape (nparams,)*) –

set_xi(*xi*)

Assign elliptical anisotropy ratio to grid

Parameters

xi (*np ndarray, shape (nparams,)*) –

to_vtk(*fields, filename*)

Save mesh variables and/or raypaths to VTK format

Parameters

- **fields** (*dict*) – dict of variables to save to file. Variables should be np.ndarray of size equal to either the number of nodes of the number of cells of the mesh, or a list of raypath coordinates.
- **filename** (*str*) – Name of file without extension for saving (extension vtu will be added). Raypaths are saved in separate files, and filename will be appended by the dict key and have a vtp extension.

Notes

VTK files can be visualized with Paraview (<https://www.paraview.org>)

class ttcrpy.tmesh.Mesh3d

class to perform raytracing with tetrahedral meshes

nparams

total number of parameters for grid

Type

int

n_threads

number of threads for raytracing

Type

int

Constructor

Mesh3d(*nodes, tetra, n_threads, cell_slowness, method, gradient_method, tt_from_rp, process_vel, eps, maxit, min_dist, n_secondary, n_tertiary, radius_factor_tertiary, translate_grid=False*) → *Mesh3d*

Parameters

- **nodes** (*np.ndarray, shape (nnodes, 3)*) – node coordinates
- **tetra** (*np.ndarray of int, shape (ntetra, 4)*) – indices of nodes forming the tetrahedra
- **n_threads** (*int*) – number of threads for raytracing (default is 1)
- **cell_slowness** (*bool*) – slowness defined for cells (True) or nodes (False) (default is 1)
- **method** (*string*) –
raytracing method (default is FSM)
 - 'FSM' : fast marching method
 - 'SPM' : shortest path method
 - 'DSPM' : dynamic shortest path
- **gradient_method** (*int*) –
method to compute traveltime gradient (default is 1)
 - 0 : least-squares first-order
 - 1 : least-squares second-order
 - 2 : Averaging-Based method
- **tt_from_rp** (*bool*) – compute traveltimes from raypaths (FSM or DSPM only) (default is 1)
- **process_vel** (*bool*) – process velocity instead of slowness at nodes when interpolating and computing matrix of partial derivative of traveltime w/r to model parameters (interpolation: for cell_slowness == False or FSM) (defaults is False)
- **eps** (*double*) – convergence criterion (FSM) (default is 1e-15)
- **maxit** (*int*) – max number of sweeping iterations (FSM) (default is 20)
- **min_dist** (*double*) – tolerance for backward raytracing (default is 1e-5)
- **n_secondary** (*int*) – number of secondary nodes (SPM & DSPM) (default is 2)

- **n_tertiary** (*int*) – number of tertiary nodes (DSPM) (default is 2)
- **radius_factor_tertiary** (*double*) – multiplication factor used to compute radius of sphere around source that includes tertiary nodes (DSPM). The radius is the average edge length multiplied by this factor (default is 3)
- **translate_grid** (*bool*) – Translate the grid such that origin is (0, 0, 0) to perform computations, which may increase accuracy when large values, e.g. UTM coordinates, are used. When raytracing, src and rcv should be given in the original system, and output raypath coordinates are also given in the original system (default if False)

static builder(*filename, n_threads, cell_slowness, method, gradient_method, tt_from_rp, process_vel, eps, maxit, min_dist, n_secondary, n_tertiary, radius_factor_tertiary, translate_grid=0*)

Build instance of Mesh3d from VTK file

Parameters

- **filename** (*str*) – Name of file holding a vtkUnstructuredGrid. The grid must have point or cell attribute named either ‘Slowness’, ‘slowness’, ‘Velocity’, ‘velocity’, or ‘P-wave velocity’. All cells must be of type vtkTetra
- **Constructor** (*Other parameters are defined in*) –

Returns

mesh – mesh instance

Return type

Mesh3d

compute_D(*coord*)

Return matrix of interpolation weights for velocity data points constraint

Parameters

coord (*np.ndarray, shape (npts, 3)*) – coordinates of data points

Returns

D – Matrix of interpolation weights

Return type

scipy csr_matrix, shape (npts, nparams)

compute_K(*order=2, taylor_order=2, weighting=True, squared=True, s0inside=False, additional_points=0*)

Compute smoothing matrices (spatial derivative)

Parameters

- **order** (*int*) – order of derivative (1 or 2, 2 by default)
- **taylor_order** (*int*) – order of taylor series expansion (1 or 2, 2 by default)
- **weighting** (*bool*) – apply inverse distance weighting (True by default)
- **squared** (*bool*) – Second derivative evaluated by taking the square of first derivative. Applied only if order == 2 (True by default)
- **s0inside** (*bool*) – (experimental) ignore slowness value at local node (value is a filtered estimate) (False by default)
- **additional_points** (*int*) – use additional points to compute derivatives (minimum sometimes yield noisy results when rays are close to domain limits) (0 by default)

Returns

Kx, Ky, Kz – matrices for derivatives along x, y, & z

Return typetuple of `csr_matrix`**data_kernel_straight_rays**(*Tx*, *Rx*) → *L*

Raytracing with straight rays in 3D

Parameters

- **Tx** (*np.ndarray*) –
source coordinates, nTx by 3
 - 1st column contains X coordinates,
 - 2nd contains Y coordinates
 - 3rd contains Z coordinates
- **Rx** (*np.ndarray*) –
receiver coordinates, nRx by 3
 - 1st column contains X coordinates,
 - 2nd contains Y coordinates
 - 3rd contains Z coordinates

Returns*L* – data kernel matrix ($tt = L * \text{slowness}$)**Return type**scipy `csr_matrix`

Note: Tx and Rx should contain the same number of rows, each row corresponding to a source-receiver pair

get_grid_traveltimes(*thread_no=0*)

Obtain traveltimes computed at primary grid nodes

Parameters**thread_no** (*int*) – thread used to computed traveltimes (default is 0)**Returns***tt* – traveltimes**Return type***np.ndarray*, shape (nnodes,)**get_number_of_cells**()**Returns**

number of cells in grid

Return type*int***get_number_of_nodes**()**Returns**

number of nodes in grid

Return type*int*

get_s0(*hypo*, *slowness=None*)

Return slowness at source points

Parameters

- **hypo** (*np.ndarray with 5 columns*) –
hypo holds source information, i.e.
 - 1st column is event ID number
 - 2nd column is origin time
 - 3rd column is source easting
 - 4th column is source northing
 - 5th column is source elevation
- **slowness** (*np ndarray, shape (nparams,) (optional)*) – slowness at grid nodes or cells (depending on cell_slowness)

Returns

s0 – slowness at source points

Return type

np.ndarray

is_outside(*pts*)

Check if points are outside grid

Parameters

pts (*np ndarray, shape (npts, 3)*) – coordinates of points to check

Returns

True if at least one point outside grid

Return type

bool

n_threads

number of threads for raytracing

Type

int

nparams

total number of parameters for mesh

Type

int

raytrace(*source*, *rcv*, *slowness=None*, *thread_no=None*, *aggregate_src=False*, *compute_L=False*, *return_rays=False*) → *tt*, *rays*, *L*

Perform raytracing

Parameters

- **source** (*2D np.ndarray with 3, 4 or 5 columns*) – see notes below
- **rcv** (*2D np.ndarray with 3 columns*) – Columns correspond to x, y and z coordinates
- **slowness** (*np ndarray, (None by default)*) – slowness at grid nodes or cells (depending on cell_slowness) if None, slowness must have been assigned previously

- **thread_no** (*int (None by default)*) – Perform calculations in thread number “thread_no” if None, attempt to run in parallel if warranted by number of sources and value of n_threads in constructor
- **aggregate_src** (*bool (False by default)*) – if True, all source coordinates belong to a single event
- **compute_L** (*bool (False by default)*) – Compute matrices of partial derivative of travel time w/r to slowness (or velocity if process_vel == True in constructor)
- **return_rays** (*bool (False by default)*) – Return raypaths

Returns

- **tt** (*np.ndarray*) – travel times for the appropriate source-rcv (see Notes below)
- **rays** (*list of np.ndarray*) – Coordinates of segments forming raypaths (if return_rays is True)
- **L** (*list of csr_matrix or scipy csr_matrix*) – Matrix of partial derivative of travel time w/r to slowness. if input argument source has 5 columns or if slowness is defined at nodes, L is a list of matrices and the number of matrices is equal to the number of sources otherwise, L is a single csr_matrix

Notes

If source has 3 columns:

- Columns correspond to x, y and z coordinates
- Origin time (t0) is 0 for all points

If source has 4 columns:

- 1st column corresponds to origin times
- 2nd, 3rd & 4th columns correspond to x, y and z coordinates

If source has 5 columns:

- 1st column corresponds to event ID
- 2nd column corresponds to origin times
- 3rd, 4th & 5th columns correspond to x, y and z coordinates

For the latter case (5 columns), source and rcv should contain the same number of rows, each row corresponding to a source-receiver pair. For the 2 other cases, source and rcv can contain the same number of rows, each row corresponding to a source-receiver pair, or the number of rows may differ if aggregate_src is True or if all rows in source are identical.

set_slowness(*slowness*)

Assign slowness to grid

Parameters

slowness (*np ndarray, shape (nparams,)*) –

set_traveltime_from_raypath(*ttrp*)

Set option to compute traveltimes using raypath

Parameters

ttrp (*bool*) – option value

set_use_thread_pool(*use_thread_pool*)

Set option to use thread pool instead of parallel loop

Parameters

use_thread_pool (*bool*) – option value

set_velocity(*velocity*)

Assign velocity to grid

Parameters

velocity (*np ndarray, shape (nparams,)*) –

to_vtk(*fields, filename*)

Save mesh variables and/or raypaths to VTK format

Parameters

- **fields** (*dict*) – dict of variables to save to file. Variables should be np.ndarray of size equal to either the number of nodes of the number of cells of the mesh, or a list of raypath coordinates.
- **filename** (*str*) – Name of file without extension for saving (extension vtu will be added). Raypaths are saved in separate files, and filename will be appended by the dict key and have a vtp extension.

Notes

VTK files can be visualized with Paraview (<https://www.paraview.org>)

ttcrpy.tmesh.set_verbose(*v*)

Set verbosity level for C++ code

Parameters

v (*int*) – verbosity level

REFERENCES

The main papers describing the algorithms found in *ttcrpy* are:

- Nasr, Maher; Giroux, Bernard et Dupuis, Christian J. (2020). A hybrid approach to compute seismic travel times in 3D tetrahedral meshes. *Geophys. Prospect.* DOI : 10.1111/1365-2478.12930 <https://onlinelibrary.wiley.com/doi/abs/10.1111/1365-2478.12930>
- Giroux B et Larouche B. 2013. Task-parallel implementation of 3D shortest path raytracing for geophysical applications, *Computers & Geosciences*, 54, 130-141. DOI : <https://www.sciencedirect.com/science/article/pii/S0098300412004128>
- Nasr M, Giroux B, Dupuis JC, 2018. An optimized approach to compute traveltimes in 3D unstructured meshes. *SEG Technical Program Expanded Abstracts 2018*, pp. 5073-5077. DOI : 10.1190/segam2018-2997918.1 <https://library.seg.org/doi/10.1190/segam2018-2997918.1>
- Giroux B. 2014. Comparison of grid-based methods for raytracing on unstructured meshes, *SEG Technical Program Expanded Abstracts 2014*, pp. 3388-3392. DOI : <https://library.seg.org/doi/10.1190/segam2014-1197.1>
- Giroux B, 2013. Shortest path raytracing in tetrahedral meshes. 75th EAGE Conference & Exhibition, Londres, 10-13 June. DOI : 10.3997/2214-4609.20130236 <https://www.earthdoc.org/content/papers/10.3997/2214-4609.20130236>

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

t

`ttcrpy.rgrid`, [19](#)

`ttcrpy.tmesh`, [35](#)

B

`builder()` (*tcrpy.rgrid.Grid3d static method*), 29
`builder()` (*tcrpy.tmesh.Mesh2d static method*), 37
`builder()` (*tcrpy.tmesh.Mesh3d static method*), 41

C

`compute_D()` (*tcrpy.rgrid.Grid2d method*), 21
`compute_D()` (*tcrpy.rgrid.Grid3d method*), 29
`compute_D()` (*tcrpy.tmesh.Mesh3d method*), 41
`compute_K()` (*tcrpy.rgrid.Grid2d method*), 21
`compute_K()` (*tcrpy.rgrid.Grid3d method*), 29
`compute_K()` (*tcrpy.tmesh.Mesh3d method*), 41
`Constructor` (*tcrpy.rgrid.Grid2d attribute*), 20
`Constructor` (*tcrpy.rgrid.Grid3d attribute*), 28
`Constructor` (*tcrpy.tmesh.Mesh2d attribute*), 36
`Constructor` (*tcrpy.tmesh.Mesh3d attribute*), 40

D

`data_kernel_straight_rays()` (*tcrpy.rgrid.Grid2d static method*), 22
`data_kernel_straight_rays()` (*tcrpy.rgrid.Grid3d static method*), 30
`data_kernel_straight_rays()`
 (*tcrpy.tmesh.Mesh3d method*), 42
`dx` (*tcrpy.rgrid.Grid2d attribute*), 19, 22
`dx` (*tcrpy.rgrid.Grid3d attribute*), 27, 30
`dy` (*tcrpy.rgrid.Grid3d attribute*), 28, 30
`dz` (*tcrpy.rgrid.Grid2d attribute*), 20, 22
`dz` (*tcrpy.rgrid.Grid3d attribute*), 28, 30

G

`get_grid_traveltimes()` (*tcrpy.rgrid.Grid2d method*), 22
`get_grid_traveltimes()` (*tcrpy.rgrid.Grid3d method*), 31
`get_grid_traveltimes()` (*tcrpy.tmesh.Mesh2d method*), 37
`get_grid_traveltimes()` (*tcrpy.tmesh.Mesh3d method*), 42
`get_number_of_cells()` (*tcrpy.rgrid.Grid2d method*), 22

`get_number_of_cells()` (*tcrpy.rgrid.Grid3d method*), 31
`get_number_of_cells()` (*tcrpy.tmesh.Mesh2d method*), 37
`get_number_of_cells()` (*tcrpy.tmesh.Mesh3d method*), 42
`get_number_of_nodes()` (*tcrpy.rgrid.Grid2d method*), 23
`get_number_of_nodes()` (*tcrpy.rgrid.Grid3d method*), 31
`get_number_of_nodes()` (*tcrpy.tmesh.Mesh2d method*), 37
`get_number_of_nodes()` (*tcrpy.tmesh.Mesh3d method*), 42
`get_s0()` (*tcrpy.rgrid.Grid2d method*), 23
`get_s0()` (*tcrpy.rgrid.Grid3d method*), 31
`get_s0()` (*tcrpy.tmesh.Mesh3d method*), 42
`get_slowness()` (*tcrpy.rgrid.Grid2d method*), 23
`get_slowness()` (*tcrpy.rgrid.Grid3d method*), 31
`Grid2d` (*class in tcrpy.rgrid*), 19
`Grid2d` (*tcrpy.rgrid.Grid2d attribute*), 20
`Grid3d` (*class in tcrpy.rgrid*), 27
`Grid3d` (*tcrpy.rgrid.Grid3d attribute*), 28

I

`ind()` (*tcrpy.rgrid.Grid3d method*), 32
`indc()` (*tcrpy.rgrid.Grid3d method*), 32
`is_outside()` (*tcrpy.rgrid.Grid2d method*), 23
`is_outside()` (*tcrpy.rgrid.Grid3d method*), 32
`is_outside()` (*tcrpy.tmesh.Mesh3d method*), 43

M

`Mesh2d` (*class in tcrpy.tmesh*), 35
`Mesh2d` (*tcrpy.tmesh.Mesh2d attribute*), 36
`Mesh3d` (*class in tcrpy.tmesh*), 39
`Mesh3d` (*tcrpy.tmesh.Mesh3d attribute*), 40
`module`
 tcrpy.rgrid, 19
 tcrpy.tmesh, 35

N

`n_threads` (*tcrpy.rgrid.Grid2d attribute*), 20, 23

`n_threads` (*ttcrpy.rgrid.Grid3d* attribute), 28, 32
`n_threads` (*ttcrpy.tmesh.Mesh2d* attribute), 36, 37
`n_threads` (*ttcrpy.tmesh.Mesh3d* attribute), 40, 43
`nparams` (*ttcrpy.rgrid.Grid2d* attribute), 20, 24
`nparams` (*ttcrpy.rgrid.Grid3d* attribute), 28, 32
`nparams` (*ttcrpy.tmesh.Mesh2d* attribute), 35, 37
`nparams` (*ttcrpy.tmesh.Mesh3d* attribute), 40, 43

R

`raytrace()` (*ttcrpy.rgrid.Grid2d* method), 24
`raytrace()` (*ttcrpy.rgrid.Grid3d* method), 33
`raytrace()` (*ttcrpy.tmesh.Mesh2d* method), 38
`raytrace()` (*ttcrpy.tmesh.Mesh3d* method), 43

S

`set_delta()` (*ttcrpy.rgrid.Grid2d* method), 25
`set_epsilon()` (*ttcrpy.rgrid.Grid2d* method), 25
`set_gamma()` (*ttcrpy.rgrid.Grid2d* method), 25
`set_s2()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_s2()` (*ttcrpy.tmesh.Mesh2d* method), 38
`set_s4()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_s4()` (*ttcrpy.tmesh.Mesh2d* method), 38
`set_slowness()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_slowness()` (*ttcrpy.rgrid.Grid3d* method), 34
`set_slowness()` (*ttcrpy.tmesh.Mesh2d* method), 39
`set_slowness()` (*ttcrpy.tmesh.Mesh3d* method), 44
`set_tilt_angle()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_tilt_angle()` (*ttcrpy.tmesh.Mesh2d* method), 39
`set_travelttime_from_raypath()`
 (*ttcrpy.rgrid.Grid2d* method), 26
`set_travelttime_from_raypath()`
 (*ttcrpy.rgrid.Grid3d* method), 34
`set_travelttime_from_raypath()`
 (*ttcrpy.tmesh.Mesh2d* method), 39
`set_travelttime_from_raypath()`
 (*ttcrpy.tmesh.Mesh3d* method), 44
`set_use_thread_pool()` (*ttcrpy.rgrid.Grid2d* method),
 26
`set_use_thread_pool()` (*ttcrpy.rgrid.Grid3d* method),
 34
`set_use_thread_pool()` (*ttcrpy.tmesh.Mesh2d*
 method), 39
`set_use_thread_pool()` (*ttcrpy.tmesh.Mesh3d*
 method), 44
`set_velocity()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_velocity()` (*ttcrpy.rgrid.Grid3d* method), 34
`set_velocity()` (*ttcrpy.tmesh.Mesh2d* method), 39
`set_velocity()` (*ttcrpy.tmesh.Mesh3d* method), 45
`set_verbose()` (in module *ttcrpy.rgrid*), 35
`set_verbose()` (in module *ttcrpy.tmesh*), 45
`set_Vp0()` (*ttcrpy.rgrid.Grid2d* method), 25
`set_Vs0()` (*ttcrpy.rgrid.Grid2d* method), 25
`set_xi()` (*ttcrpy.rgrid.Grid2d* method), 26
`set_xi()` (*ttcrpy.tmesh.Mesh2d* method), 39

`shape` (*ttcrpy.rgrid.Grid2d* attribute), 20, 26
`shape` (*ttcrpy.rgrid.Grid3d* attribute), 28, 34

T

`to_vtk()` (*ttcrpy.rgrid.Grid2d* method), 27
`to_vtk()` (*ttcrpy.rgrid.Grid3d* method), 34
`to_vtk()` (*ttcrpy.tmesh.Mesh2d* method), 39
`to_vtk()` (*ttcrpy.tmesh.Mesh3d* method), 45
`ttcrpy.rgrid`
 module, 19
`ttcrpy.tmesh`
 module, 35

X

`x` (*ttcrpy.rgrid.Grid2d* attribute), 19, 27
`x` (*ttcrpy.rgrid.Grid3d* attribute), 27, 35

Y

`y` (*ttcrpy.rgrid.Grid3d* attribute), 27, 35

Z

`z` (*ttcrpy.rgrid.Grid2d* attribute), 19, 27
`z` (*ttcrpy.rgrid.Grid3d* attribute), 27, 35