SoftwareX 16 (2021) 100834

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

ttcrpy: A Python package for traveltime computation and raytracing

Bernard Giroux

Centre Eau Terre Environnement, Institut national de la recherche scientifique, Québec, Canada

ARTICLE INFO

Article history:
Received 5 July 2021
Received in revised form 29 September 2021
Accepted 30 September 2021

Keywords:
Traveltime
Raytracing
Geophysics
Python
C++

ABSTRACT

`ttcrpy` is a package for computing traveltimes and raytracing of seismic and electromagnetic waves for geophysical applications, e.g. ray-based seismic/GPR tomography, microseismic event location (joint hypocenter-velocity inversion), and migration. The package allows performing computations on 2D and 3D rectilinear grids, as well as 2D triangular and 3D tetrahedral meshes. For improved versatility, 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 for practical use.

Code metadata

Current code version	v1.1.8
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00123
Code Ocean compute capsule	https://codeocean.com/algorithm/76d9ec7f-746e-4c8e-b455-9aa0a2f42eb9/
Legal Code License	GPL-3.0
Code versioning system used	git
Software code languages, tools, and services used	Python, C++
Compilation requirements, operating environments & dependencies	C++11 compliant compiler, Cython
If available Link to developer documentation/manual	https://ttcrpy.readthedocs.io/en/latest/
Support email for questions	bernard.giroux@ete.inrs.ca

1. Motivation and significance

Geophysical methods are used to characterize the subsurface and build representations of natural and man-made structures found underground [1]. Among geophysical techniques, seismic methods and Ground-Penetrating-Radar (GPR) play an important role due to their capacity to provide high-resolution images. A common aspect of these two methods is that the ground is probed with waves, of mechanical nature for seismics and electromagnetic for GPR. Wave propagation is influenced by contrasts in acoustic or electromagnetic impedance, and the aim of the methods is to infer the location and magnitude of such contrasts from the measurements.

Processing and interpreting seismic and GPR data often require computing the traveltime of a wave propagating from a source to some receivers, as well as computing the trajectory from that source to the receivers (raytracing), for various spatial distribution of wave velocity. This is the case for instance in traveltime tomography [2], migration [3], and microseismic event location [4]. Depending on the application, data can be collected with sources and receivers covering a volume, or aligned along a plane or a line. Survey design is dictated by the spatial distribution of velocity, which could be 3D, or which could be approximated by a 2D model with acceptable accuracy. Surveys can be carried out over more or less flat land, on accidented terrains with varying topography, or over sites with known infrastructure of arbitrary shape (e.g. mine galleries, tunnels, or dams). The choice of discretization is therefore important to accurately represent the features present at the study site. On the other hand, this choice may be dictated by other considerations. For example, unstructured meshes are well suited to map objects of arbitrary shapes, but are more complex to handle than regular

E-mail address: bernard.giroux@ete.inrs.ca.

https://doi.org/10.1016/j.softx.2021.100834

grids, and could be overkill when no complex features are found at the location. Unstructured meshes could also not be handled by a critical component of the computing code at hand, leading to an intermediate regriding step causing loss of accuracy. The aim of the `ttcrpy` package is to allow seamlessly performing traveltime computation and raytracing for a variety of scenarios. The package currently accepts velocity models discretized with 2D and 3D rectilinear grids as well as 2D triangular and 3D tetrahedral meshes. For additional flexibility, it is also possible to assign slowness (inverse of velocity) values at the nodes or to the cells of the mesh. For the latter case, isotropic velocity can be defined for all supported meshes, and transverse isotropy is possible for rectilinear grids.

The code was integrated in a framework developed to map seismic velocity changes precursor to mining rockbursts [5]. Using data collected in a Canadian mine, their methodology successfully mapped velocity changes associated to two rockburst instances. A critical step in their framework is the comparison of measured arrival times with arrival times computed using updated velocity models, which is performed with ttcrpy. This code is also a key component of `hypopy`, a Python module for seismic hypocenter location working with rectilinear grids [6]. Owing to the capabilities of `ttcrpy`, `hypopy` is currently under development to support tetrahedral meshes. The `ttcrpy` package is versatile, easy to install and simple to use, and it is expected that it will be adopted for a wide array of applications.

The `ttcrpy` package can be used in most situations where traveltime or raypaths are required. The only requirements are that a description of the spatial distribution of velocity and the location of the sources and receivers be defined on a chosen mesh. The general approach is to first create an instance of the meshes supporting the description of the velocity model. This instance can then be used to compute the traveltimes and raypaths for various combinations of sources and receivers. Values of velocity within the mesh can be updated at any point prior to the computations, so the mesh instance can be reused in iterative algorithms.

The general approach used for computing traveltimes with `ttcrpy` combines two sequential steps. The first uses a grid-based method (i.e. the Fast-Sweeping method (FSM) [7], the Shortest-Path method (SPM) [8,9], or the Dynamic Shortest-Path method (DSPM) [10]) to compute traveltime at all nodes of the mesh. The second uses the steepest traveltime gradient method to trace back raypaths for each source–receiver pair. Traveltimes at each receiver are recomputed using slowness values along the raypaths. The second step incurs some overhead, but generally increases accuracy, unless complex velocity models with strong gradients are fed in input [10]. If such a condition is warranted, the second step can be optional. Traveltimes at receiver locations are then interpolated instead.

2. Software description

`ttcrpy` is a free Python package published under the GNU General Public license (GPLv3) [11]. The codebase is open to contributions and further development on GitHub [12]. Installation and distribution have been established since early 2020 through the Python Package Index [13], and the Python API is documented on ReadTheDocs [14].

2.1. Software architecture

At its core, `ttcrpy` is a collection of C++ classes that are wrapped with Cython. Each C++ class holds the implementation of a particular algorithm (Fast-Sweeping in 2D rectilinear grid, or Dynamic Shortest-Path in 3D tetrahedral meshes). C++ classes inherit from two major superclasses: `Grid2D` and `Grid3D`, which share a common interface. On the Cython side, the number of classes is reduced to four: rectilinear grids and unstructured meshes in 2D and 3D. The classes, named `Grid2d`, `Grid3d`, `Mesh2d`, and `Mesh3d`, all share a common interface. The choice of a particular implementation is made during object instantiation, depending on the choice of the class itself as well as on arguments passed to the constructor. Apart from a few Cython declaration files, the Cython code is contained in two files (`rgrid.pyx` and `tmesh.pyx`), which constitute the two modules of the `ttcrpy` package. Fig. 1 depicts the relationship between the C++ classes and their Cython counterpart.

Each Cython class holds a pointer to either a `Grid2D` or `Grid3D` C++ object. Cython class methods take Python objects as arguments and create new C++ object to pass information to the `Grid2D` or `Grid3D` C++ object when needed. At no time do the C++ objects use Python built-in type objects, which means that they can be used without the Python global interpreter lock (GIL). Once objects are instantiated, calculations are done by calling a method called `raytrace`. If calculations must be performed for a number of sources, all source coordinates can be passed at once in argument and, provided that the option was selected at creation time, calculations are then performed in parallel following a task pattern [15]. Parallel calculations are handled on the C++ side using a dedicated thread pool with a predefined number of threads. This means that parallel work is transparent to Python and that it runs in the background, irrespective of the GIL.

2.2. Software functionalities

The main functionalities of `ttcrpy` are to compute traveltime and raypaths for various scenarios. Parameters passed to the constructor determine the choice of the algorithm. Other functionalities that are selected at instantiation are:

1. if slowness values are assigned to grid cells or grid nodes,
2. if anisotropy should be considered (currently available for 2D rectilinear grids only),
3. if traveltimes should be computed in a two-step approach using gradient along raypaths or simply interpolated, and
4. if calculations should be performed in parallel.

Choosing whether slowness values should be assigned to grid cells or grid nodes has implication in tomography problems as it determines the number of unknown parameters to estimate. With rectilinear grids, the number of cells is always smaller than the number of nodes, and it is thus advantageous to assign slowness to cells with this type of grid, as the system to solve will be smaller. The opposite applies to triangular or tetrahedral meshes.

Inverse problems such as tomography require the Jacobian matrix, or Fréchet derivatives, in order to update the sought parameters [16]. Construction of the Jacobian matrix depends on the discretization, and `ttcrpy` contains implementations for a number of scenarios. For instance, consider the case where slowness values are assigned to cells. Fig. 2 shows an example of a raypath that runs across the grid cells, from the transmitter to the receiver. In continuous media, the traveltime t is the integral of slowness along the raypath l from the transmitter Tx to the receiver Rx:

$$t = \int_{\mathrm{Tx}}^{\mathrm{Rx}} s(l)\, dl, \tag{1}$$

where s is the slowness, which is a function of the raypath. In discrete form, the integral becomes a summation over the ray segments,

$$t = \sum_{i}^{\text{segments}} s_j l_i, \tag{2}$$

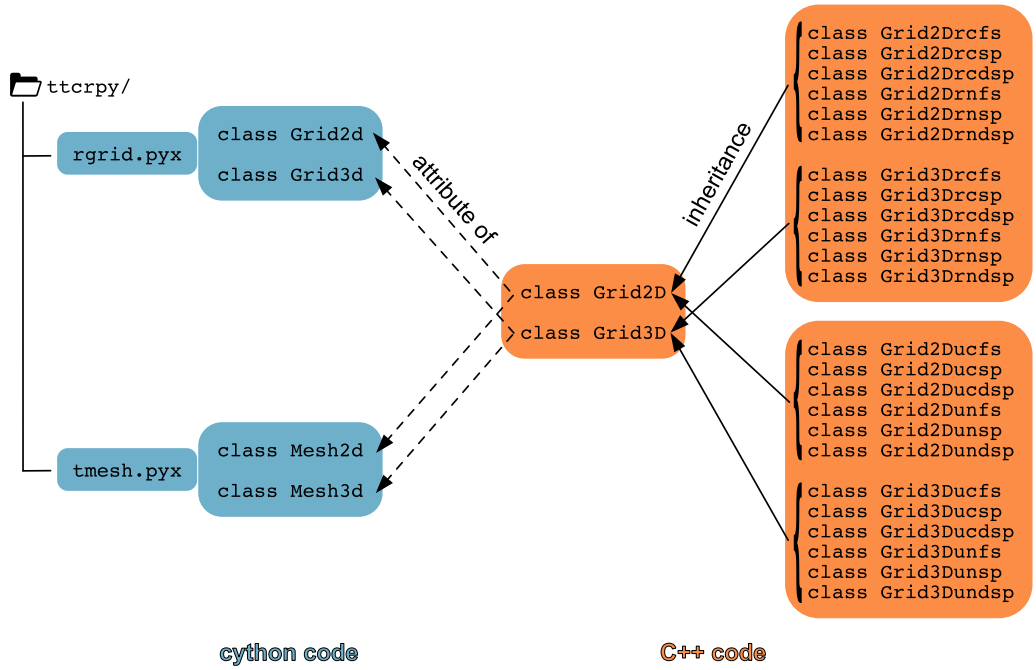

Fig. 1. Structure of the ttcrpy codebase. Naming convention for the C++ classes defines first dimensionality, then type of mesh (letter 'r' for rectilinear and letter 'u' for unstructured), attribution of slowness to cells (letter 'c') or to nodes (letter 'n'), and finally algorithm ('fs' for fast-sweeping, 'sp' for shortest-path, and 'dsp' for dynamic shortest path).

where s_j is the slowness in the cell traversed by the ith segment, and l_i is the length of the segment in the cell. The Jacobian matrix contains the derivatives of t with respect to model parameters, $\partial t/\partial s$ for this specific case, and the elements of the matrix are then either 0 for cells not traversed by ray segments, or l_i for the other cells. Upon user request, when calling the `raytrace` method, the process of filling the matrix for all pairs of source and receivers given the computed raypaths is handled in the Cython methods. This returns a SciPy sparse matrix. Note that an alternative formulation is used when slowness values are assigned at grid nodes.

The choice of using the two-step approach to compute traveltimes may have implications when it is used in tomography problems, especially when the inverse problem is highly ill-conditioned and regularization does not allow for a realistic velocity model. In our experience, this can arise when non physically admissible velocity contrasts are introduced in the model, especially close to the source. In such a case, the gradient of the traveltimes is susceptible to errors and may not point toward the source point with sufficient accuracy. This can preclude convergence of the raytracing step. A runtime exception is raised when convergence cannot be achieved, and calculation of the traveltime is interrupted. This convergence problem can be avoided by selecting the Shortest-Path Method, in which raytracing is performed using a different algorithm. With the SPM, a "discrete" raypath is recorded while traveltimes are computed at all grid nodes [15], and convergence is guaranteed. Performance aspects of these different approaches are discussed in details for tetrahedral meshes in [10].

Most users will want the possibility to save grid values and raypaths for further visualization in external programs such as Paraview. This capability has been implemented in `ttcrpoy` using the VTK format [17]. Grid values refer to scalars assigned either to grid cells or grid nodes, typically whole grid traveltimes or velocity models obtained after tomography. This functionality is achieved through the method `to_vtk`, which is implemented in all four Cython classes.

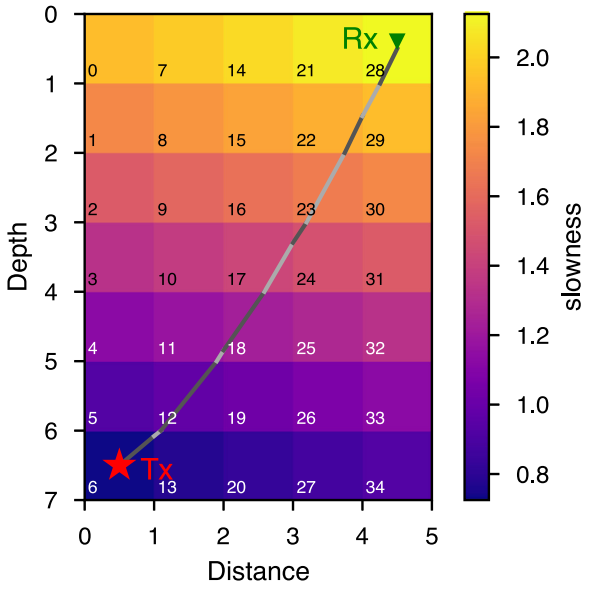

Fig. 2. Example of segmented raypath for a model with slowness values assigned to grid cells. For illustrative purposes, ray segments are alternating between dark and light gray. Grid cell numbers appear in their lower left corner. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3. Illustrative examples

The following simple examples illustrate the main functionalities of the package. The first case is a 3D layered model discretized with a rectilinear grid. The following lines of code show how the model is defined and how an instance of a `Grid3d` object is created, along with how it can be saved for visualization. Space and time units are not explicitly defined in the codebase, and traveltime units will be consistent with the units used for

slowness and spatial dimensions. In the following, units are not defined simply to lighten notation. Fig. 3 shows the resulting model.

```
N = 40          # number of voxels in each direction
dx = 0.5        # cell size

# node coordinates
xn = np.arange(0, (N+1)*dx, dx)
yn = np.arange(0, (N+1)*dx, dx)
zn = np.arange(0, (N+1)*dx, dx)

# create grid with default values (by default,
# slowness is defined for cells)
grid = Grid3d(xn, yn, zn)

# values of slowness along vertical axis,
# define first vertical gradient
V0 = 1.0
V20 = 3.0
b = (V20-V0)/20.0
# fill array
slowness = np.empty((N,))
for n in range(N):
    z = 2*int(zn[n]/2) + 1
    # z at the center of the layer
    slowness[n] = 1.0 / (V0 + b*z)
# repeat for all x & y locations
slowness = np.tile(slowness, N*N)

# Assign slowness to grid
grid.set_slowness(slowness)

# Save to VTK format to visualize
grid.to_vtk({'Velocity': 1./slowness}, 'example1')
```

The next logical step is to compute traveltimes and raypaths for some sources. This is accomplished by the next lines of code.

```
# Define the source location
src = np.array([[0.5, 0.5, 0.5]])

# Define some receivers, first a horizontal spread
rcv = np.c_[np.arange(1.5, 20.0),
            np.arange(1.5, 20.0),
            0.5+np.zeros((19,))]
# Add receivers in a "borehole" - vertical receiver
# line
rcv = np.r_[rcv, np.c_[19.5+np.zeros((19,)),
                       19.5+np.zeros((19,)),
                       np.arange(1.5, 20.0)]]

# Compute traveltimes and raypaths
ref = time.time()
tt, rays = grid.raytrace(src, rcv, return_rays=True)
compute_time1 = time.time() - ref
# Save raypaths
grid.to_vtk({'raypaths for shot no 1': rays},
'example1_rays')
```

The results are shown in Fig. 4. By default, the FSM is used and the raypaths are computed by using the direction of the gradient of traveltime, from the receivers toward the source. For this case, rays should (1) be straight inside the layers, (2) exhibit a sharp change of direction at the interfaces, and (3) the path of head waves should follow exactly the interfaces. Fig. 4 shows that it is not the case for the FSM since rays bend smoothly as they get closer to the interfaces. This means that this solution deviates from theoretical paths they should adopt at the interfaces.

The same example is repeated using the SPM to illustrate the differences that can result from this method. The important thing to remember is that SPM records the raypaths during traveltime computation over the grid nodes. In this case, a second grid must be created to accommodate this functionality. This is done as shown in the following code example.

```
# create SPM grid with default number of secondary
# node (5)
```

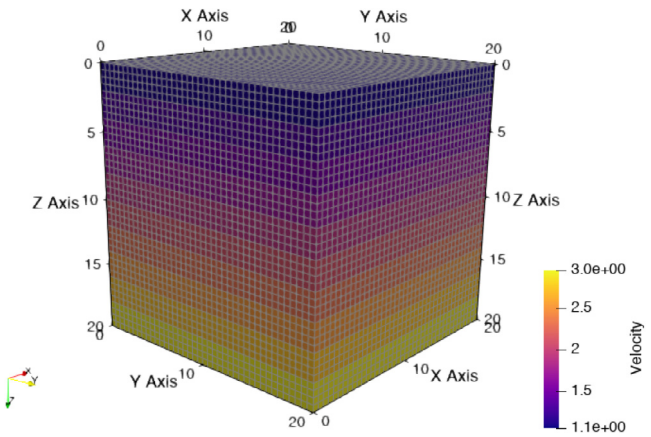

Fig. 3. Velocity model of example 1.

```
grid_spm = Grid3d(xn, yn, zn, method='SPM')

# Compute traveltimes and raypaths
ref = time.time()
tt, rays = grid_spm.raytrace(src, rcv,
                slowness=slowness, return_rays=True)
compute_time2 = time.time() - ref
```

The results are presented in Fig. 5. We see from this example that the raypaths follow trajectories that are closer to the theoretical solution. The cost for this improvement in accuracy is, however, an increase in computation time. On a 2013 Apple Mac Pro, it takes 2.8 s to solve with the FSM and 54.1 s to solve with the SPM. This is almost 1:20 increase in computation time. This large increase in computing resources, combined with many situations where smooth velocity models can be the norm, make the FSM the default option for this type of grid.

We conclude this first example by showing how the Jacobian matrix is computed. This matrix is used in inversion to compute model perturbations. Provided that model perturbations are small between iterations, this matrix can be assumed to remain constant for a few iterations. In such a case, traveltimes are obtained from a fast matrix–vector product. The next lines of code show a comparison example, where the matrix–vector product takes 3.6×10^{-4} s to compute, i.e. about 4 orders of magnitude faster than calling `raytrace`.

```
tt, L = grid_spm.raytrace(src, rcv, slowness=slowness,
                          compute_L=True)
tt2 = L@slowness
np.linalg.norm(tt - tt2)     # equal to 1.45e-13
```

The complete Python code for this example can be found in a Jupyter notebook in the GitHub repository.

The second example illustrates how topography can be taken into account and how secondary arrivals can be computed. A 2D model that contains two irregular layers has been discretized with a triangular mesh. The resulting model is shown in Fig. 6. To get reflected arrival times, one must first record the direct arrival times at points along the interface between the layers. These are then used as a second source for a subsequent modeling run. This approach can be generalized in order to compute multiply transmitted, converted and reflected arrivals, as proposed by [18].

Fig. 7 shows the modeled raypaths obtained after the two runs. We can see in this example the influence of irregular interfaces on wave propagation. The corresponding traveltimes are shown in Fig. 8. The effect of the irregular surface and interface can be seen in the reflected and refracted arrivals.

The complete Python code example, including mesh generation with gmsh, can be found in a second Jupyter notebook in the GitHub repository.

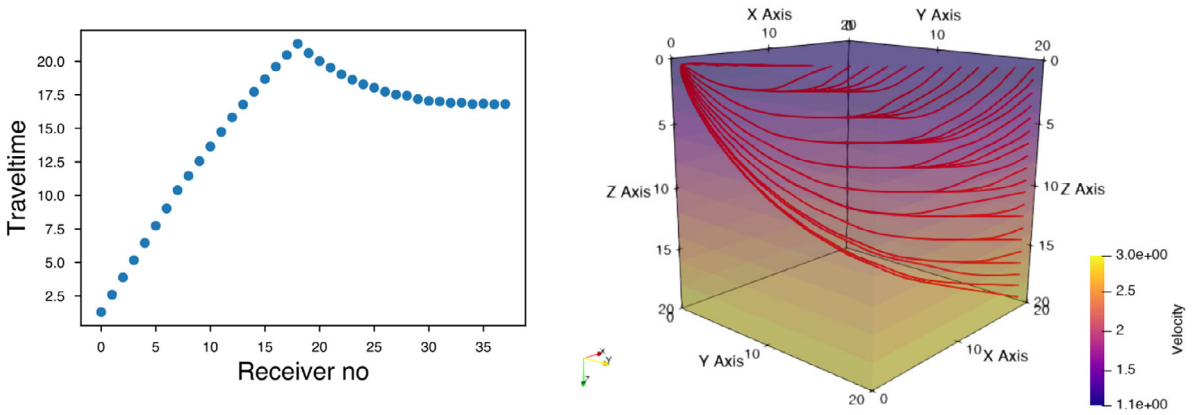

Fig. 4. Traveltimes and raypaths obtained with default parameters.

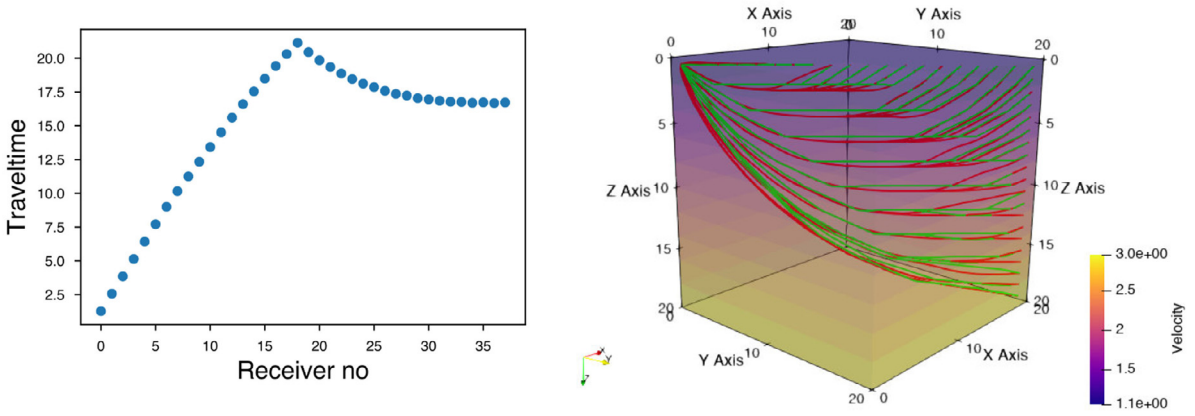

Fig. 5. Traveltimes and raypaths (in green) obtained with the SPM. Raypaths obtained with the FSM are shown in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

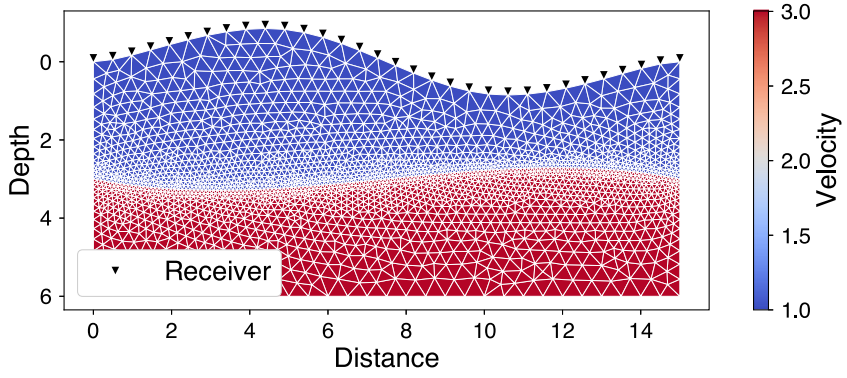

Fig. 6. Velocity model of example 2.

4. Impact

As saw shown in the previous examples, `ttcrpy` is versatile and flexible, and will be useful to tackle any problem where traveltimes and raypaths are needed. In particular, the use of unstructured grids is growing in importance in geophysics [19], and `ttcrpy` provides researchers with a framework that can be used readily with such grids.

Bringing the C++ codebase into Python has been instrumental for its adoption by young researchers (e.g. graduate students) with limited or no experience with compiled languages. This, combined with a parallel implementation, speeds up testing research hypotheses and allows more rapid debugging, as was the case for instance in the development proposed by [5]. The Cython codebase also includes utility routines to compute matrices useful for inverse problems, such as smoothing matrices (1st and 2nd order spatial derivative), matrix of interpolation weights for velocity data points constraint, and matrix of partial derivative of travel time with respect to slowness or velocity.

`ttcrpy` is a relatively young package, but statistics from Google BigQuery indicates that monthly downloads range between 3500 and 8000 from December 2020 to May 2021. Google BigQuery also indicates an increasing trend of downloads from pip, the package installer for Python, starting with 9 in February 2021 and reaching 214 in May 2021. As of 2021, the package is

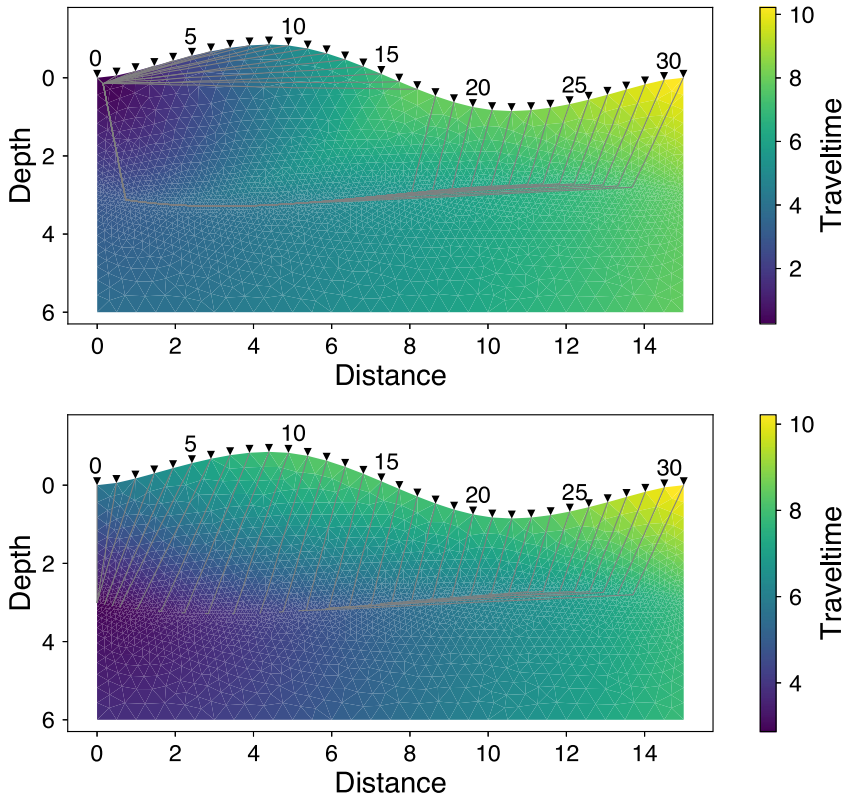

Fig. 7. Raypaths for the direct, reflected and refracted arrivals. Top: first run with primary source at the surface, bottom: second run with secondary source at the interface between the layers.

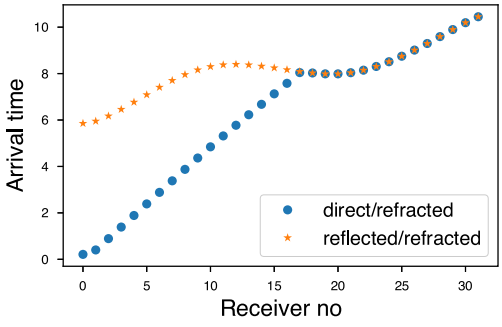

Fig. 8. Traveltime curves modeled in example 2.

also used by the author in his modeling and inversion class, which should increase its visibility outside his group.

5. Conclusions

In this article, the `ttcrpy` package is introduced. It provides a convenient way to compute traveltimes and raypaths for a variety of 2D and 3D grids. The package is designed with flexibility and versatility in mind, without sacrificing performance. The core of the compute-intensive code is programmed in a collection of C++ classes, and Cython is used to interface it with Python.

The package is released under the GNU Public Licence version 3 and source code is hosted on GitHub. Python wheels are provided on the Python Package Index (PyPi) website for easy installation with the package installer for Python (pip). The target user base includes Earth scientists, technical personnel from the private sector and academic researchers.

Current work is dedicated to bug fixing, improving documentation of the C++ codebase, and augmenting the test case suite of automated testing. Efforts are also planned to implement compute-critical portions of the code to GPU.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This project was supported by a NSERC Discovery Grant to the author (RGPIN-2017-06215). The author thanks Christian C. Dupuis, who made constructive comments on the original manuscript.

References

[1] Reynolds JM. An introduction to applied and environmental geophysics. 2nd ed. Wiley; 2011.

[2] Giroux B, Gloaguen E, Chouteau M. `bh_tomo` – a Matlab borehole georadar 2D tomography package. Comput Geosci 2007;33(1):126–37. http://dx.doi.org/10.1016/j.cageo.2006.05.014.

[3] Gray SH, Etgen J, Dellinger J, Whitmore D. Seismic migration problems and solutions. Geophysics 2001;66(5):1622–40. http://dx.doi.org/10.1190/1.1487107, URL http://link.aip.org/link/?GPY/66/1622/1.

[4] Li J, Li C, Morton SA, Dohmen T, Katahara K, Toksöz MN. Microseismic joint location and anisotropic velocity inversion for hydraulic fracturing in a tight Bakken reservoir. Geophysics 2014;79(5):C111–22. http://dx.doi.org/10.1190/geo2013-0345.1.

[5] Dip AC, Giroux B, Gloaguen E. Microseismic monitoring of rockbursts with ensemble Kalman filter. Near Surf Geophys 2021;19(4):429–45. http://dx.doi.org/10.1002/nsg.12158.

[6] Giroux B. hypopy: HYPOcenter location from arrival time data in PYthon. 2021, http://dx.doi.org/10.5281/zenodo.4610634, URL https://github.com/groupeLIAMG/hypopy.

[7] Zhao H. A fast sweeping method for eikonal equations. Math Comp 2005;74(250):603–27, URL http://www.jstor.org/stable/4100081.

[8] Nakanishi I, Yamaguchi K. A numerical experiment on nonlinear image reconstruction from first-arrival times for two-dimensional island arc structure. J Phys Earth 1952;34(2). http://dx.doi.org/10.4294/jpe1952.34.195.
[9] Moser TJ. Shortest path calculation of seismic rays. Geophysics 1991;56(1):59–67. http://dx.doi.org/10.1190/1.1442958.
[10] Nasr M, Giroux B, Dupuis JC. A hybrid approach to compute seismic travel times in three-dimensional tetrahedral meshes. Geophys Prospect 2020;68:1291–313. http://dx.doi.org/10.1111/1365-2478.12930.
[11] Gnu general public license, version 3. 2007, https://www.gnu.org/licenses/gpl-3.0.html. Last retrieved 2021-03-20.
[12] The official ttcrpy repository. 2014, https://github.com/groupeLIAMG/ttcr. [Last accessed 25 March 2021].
[13] Python package index. 2003, https://pypi.org/.
[14] ttcrpy's documentation. 2020, https://ttcrpy.readthedocs.io/en/latest/.
[15] Giroux B, Larouche B. Task-parallel implementation of 3D shortest path raytracing for geophysical applications. Comput Geosci 2013;54:130–41. http://dx.doi.org/10.1016/j.cageo.2012.12.005.
[16] Menke W. Geophysical data analysis: Discrete inverse theory. 3rd ed. Academic Press; 2012.
[17] Schroeder W, Martin K, Lorensen B. The visualization toolkit. 4th ed. Kitware; 2006.
[18] Bai C-Y, Tang X-P, Zhao R. 2-D/3-D multiply transmitted, converted and reflected arrivals in complex layered media with the modified shortest path methods. Geophys J Int 2009;179(1):201–14. http://dx.doi.org/10.1111/j.1365-246X.2009.04213.x.
[19] Lelièvre P, Carter-McAuslan A, Farquharson C, Hurich C. Unified geophysical and geological 3D earth models. Lead Edge 2012;31(3):322–8. http://dx.doi.org/10.1190/1.3694900, arXiv:http://library.seg.org/doi/pdf/10.1190/1.3694900, URL http://library.seg.org/doi/abs/10.1190/1.3694900.