SeismicMesh

Release V1.0

May 07, 2022

Contents

1 Contents **3**
- 1.1 Introduction . . . 3
- 1.2 Overview . . . 3
- 1.3 Installation . . . 7
- 1.4 Basics . . . 8
- 1.5 Modules . . . 15

Bibliography **17**

Author Keith Roberts

Contact krober@usp.br

Web Site https://github.com/krober10nd/SeismicMesh

Date May 07, 2022

Abstract

This document describes the *SeismicMesh* package. This work aims to create end-to-end workflows to build quality models suitable for Finite Element numerical wave propagators.

The package uses Python to wrap computationally intensive operations in C++ and has support for distributed memory parallelism.

CHAPTER 1

Contents

1.1 Introduction

Generating a high-quality graded mesh for a geophysical domain represents a modern challenge for sophisticated geophysical simulation workflows. In these applications, a domain is discretized typically with simplicial elements (e.g., triangles/tetrahedrals) that adapt in size to features of interest. These meshes are commonly used with Finite Element and Finite Volume methods to solve Partial Differential Equations (PDEs) that model physical processes such as the acoustic or elastic wave equation. These kind of simulations are often used in geophysical exploration studies to solve inverse problems such as Full Waveform Inversion.

There are many aspects to consider when building a mesh for a geophysical inverse problem. Mesh elements must be sufficiently well-shaped, sized to maintain numerical stability, minimize numerical dissipation, and maximize physical and numerical accuracy. For applications such as Travel Time Tomography and Reverse Time Migration, material discontinuities need to be well represented to ensure reflections are as accurate as possible. In cases with complex and irregular rock structures, explicit geometries may not exist complicating mesh generation workflows and requiring the use of graphical user interfaces to create these geometries.

The purpose of this software is quickly assemble meshes for geophysical simulation using the Finite Element/Finite Volume methods. This is accomplished by giving users simple controls to design their own meshes without having to learn or write much code or more complex software. This program strives for automation and reproduction of the mesh.

This package contains the technology to build a 2D/3D mesh from a seismic velocity model in a scriptable manner. An approach to build mesh sizing functions that can lead to high-quality, graded meshes with the generator is detailed. For geometry creation, we rely on signed distance functions to define domain features implicitly, which avoids the need to supply explicit locations of segments/surfaces.

1.2 Overview

This software aims to create end-to-end workflows (e.g., from seismic velocity model to simulation ready mesh) to build quality two- and three-dimensional (2D and 3D) unstructured triangular and tetrahedral meshes for seismic domains. The main advantage of triangular meshes over cubes or rectangles is their ability to cost-effectively resolve

variable material properties. These generated meshes are suitable for acoustic and elastic numerical wave propagators and a focus is placed on parallel unstructured mesh generation capabilities. A simple application program interfaces (API) written in Python enables the user to call parallel meshing algorithms that can be scaled up on distributed memory clusters.

SeismicMesh is currently being used to generate simplical meshes in 2D and 3D for acoustic and elastic wave propagators written using a Domain Specific Language called *Firedrake* [firedrake]. These type of numerical simulations are used in Full Waveform Inversion, Reverse Time Migration, and Time Travel Tomography applications.

1.2.1 Mesh definition

Note: Triangular and tetrahedral conformal meshes.

The domain Ω is partitioned into a finite set of cells $\mathcal{T}_h = T$ with disjoint interiors such that

$\cup_{T \in \mathcal{T}_h} T = \Omega$

Together, these cells form a mesh of the domain Ω. In our case, the cells are triangles and thus the mesh is a triangulation composed of nt triangles and np vertices in either two or three dimensional space. Note in 3D, the triangulation is composed of tetrahedral but we still refer to it as a triangulation. In 2D, a triangle has 3 vertices, 3 edges, and 1 face. In 3D, a tetrahedral has 4 vertices, 6 edges, and 4 facets. These cells t are obtained by tessellating a set of vertices that lie in two or three dimensional space using the well-known and efficient Delaunay triangulation algorithms.

High quality cells

Note: A high quality cell has a cell quality of 1.

Any set of points can be triangulated, but the resulting triangulation will likely not be useable for numerical simulation. Thus, we strive to produce high-quality geometric meshes.

There are many definitions of mesh quality. Here I use the following formula to quantify how *well-shaped* the cells where cell quality is defined as d * circumcircle_radius / incircle_radius (where d is 2 for triangles and 3 for tetrahedra). The value is between 0 and 1, where 1 is a perfectly symmetrical simplex.

1.2.2 Software architecture

Note: Python calls peformant C++ libraries like CGAL and Boost.

The software is implemented in a mixed language environment (Python and C++). The Python language is used for the API while computationally expensive operations are performed in C++. The two languages are linked together with *pybind11* and installation is carried out using *cmake*. The Computational Geometry Algorithms Library [cgal] is used to perform geometric operations that use floating point arithmetic to avoid numerical precision issues. Besides this, several common Python packages: *numpy*, *scipy*, *meshio*, *segyio*, and *mpi4py* are used.

1.2.3 Inputs

Seismic velocity model

A seismic velocity model is defined on an axis-aligned regular Cartesian grid in either 2D or 3D containing scalar values (typically the P-wave velocity speed at each grid point).

Currently, *SeismicMesh* can read seismic velocity models from SEG-y files or in binary format. The latter requires some more information; see the docstring.

Signed distance function

Given a point x, the signed distance function returns the d distance to the boundary of the domain Ω.

Let ΩDR^N be the domain in N dimensions. The boundary of the domain to-be-meshed is represented as the 0-level set of a continuous function:

$() : D ß R.$

such that:

$\Omega := xD, (x) < 0$

where : $DR + ßR$ is Lipschitz continuous and called the level set function. If we assume $|()| = 0$ on the set $xD, (x) = 0$, then we have $\Omega = xD, (x) = 0$ i.e., the boundary Ω is the 0-level set of $()$. The property that $|()| = 0$ is satisfied if $()$ is a signed distance function.

Note: We provide several simple signed distance functions: such as a Rectangle, Cube, and Disk. See the geometry module.

Mesh sizing function

Given a point x, the sizing function $f(h)$ returns the isotropic mesh size defined at x. By mesh size, we specifically mean the triangular edge length nearby *x* assuming the triangles will be close to equilateral in the finished mesh.

The purpose of *get_sizing_function_from_segy* to build this function directly from the seismic velocity model provided.

1.2.4 *DistMesh* algorithm

Note: This program uses a modified version of the *DistMesh* algorithm [distmesh] to generate simplical meshes.

For the generation of triangular meshes in 2D and 3D, we use a modified version of the *DistMesh* algorithm [distmesh]. The algorithm is both simple and practically useful as it can produce high-geometric quality meshes in N-dimensional space. Further, by utilizing our approach to produce mesh size functions, the mesh generation algorithm is capable of generating high-quality meshes faithful to user-defined target sizing fields. A benefit of this is that mesh sizes can be built to respect numerically stability requirements a priori.

Briefly, the mesh generation algorithm is iterative and terminates after a pre-set number of iterations (e.g., 50-100). It commences with an initial distribution of vertices in the domain and iteratively relocates the vertices to create higher-geometric quality elements. The edges of the mesh act as *springs* that obey a constitutive law (e.g., Hooke's Law) otherwise referred to as a *force function*. During each meshing iteration, the discrepancy between the length of the edges in the mesh connectivity and their target length from the sizing function produce movement in the triangles' vertices.

The boundary of the domain is enforced by projecting any points that leave the domain back into it each meshing iteration. After a sufficient number of iterations, an equilibrium-like state is almost always approached and the movement of the vertices becomes relatively small. The equilibrium-like state of the mesh connectivity corresponds to a mesh that contains mostly isotropic equilateral triangles, which is critical for numerical simulation. However, as with nearly all mesh generators, a sequence of mesh improvement strategies are applied after mesh generation terminates to ensure the mesh will be robust for simulation.

Mesh adaptation

Warning: Functionality to adapt an existing mesh is a work in progress

3D *Sliver* removal

3D Delaunay mesh generation algorithms form degenerate elements called *slivers*. If any *sliver* exists in a 3D mesh, the numerical solution can become unstable. Fortunately, this problem does not occur in 2D and, in 2D, a high quality mesh free of degenerate elements is easily achieved. To tackle this problem in 3D, a method similar to that of [slivers] was implemented. This algorithm aims at removing *slivers* while preserving the triangulation sizing distribution and domain boundary.

The *sliver* removal technique fits well within the *DistMesh* framework. For example, like the mesh generation approach, the algorithm operates iteratively. Each meshing iteration, it perturbs *only* vertices associated with *slivers* so that the circumspheres' radius of the *sliver* tetrahedral increases rapidly (i.e.., gradient ascent of the circumsphere radius) [slivers]. The method operates on an existing mesh that ideally already has a high-mesh quality and is efficient since it uses CGAL's incremental Delaunay capabilities. The perturbation of a vertex of the *sliver* leads to a local combinational change in the nearby mesh connectivity to maintain Delaunayhood and almost always destroys the *sliver* in lieu of elements with larger dihedral angles.

Note: A *sliver* element is defined by their dihedral angle (i.e., angle between two surfaces) of which a tetrahedral has 6. Generally, if a 3D mesh has a minimum dihedral angle less than 1 degree, it will be numerically unstable. We've had success in simulating with meshes that have minimum dihedral angles of minimally around 5 degrees.

1.2.5 Parallelism and speed

Note: This code uses distributed memory parallelism with the MPI4py package.

When constructing models at scale, the primary computational bottleneck in the *DistMesh* algorithm becomes the time spent in the Delauany triangulation algorithm, which occurs each iteration of the mesh generation step. The other steps involving the formation and calculation of the target sizing field and signed distance function are far less demanding. Using *mpi4py*, I implemented a simplified version of the [hpc_del] to parallelize the Delaunay triangulation algorithm. This approach scales well and reduces the time spent performing each meshing iteration thus making the approach feasible for large-scale 3D mesh generation applications. The domain is decomposed into axis-aligned *slices* than cut one axis of the domain. While this strategy doesn't fare well with load balancing, it simplifies the implementation and runtime communication cost associated with neighboring processor exchanges.

When possible, *SeismicMesh* uses low-level functionality from the CGAL package including the evaluation of geometric predicates, circumball calculations, polygonal intersection tests, and incremental triangulation capabilities.

1.2.6 References

1.3 Installation

1.3.1 Requirements

You need to have the following software properly installed in order to build *SeismicMesh*:

- Python >= 3.0

Note: The file `setup.cfg` in the main directory indicates all the Python dependencies and their respective version numbers (if necessary). These packages should be installed at compile time by setuptools

Note: On some Linux systems, users may have to resort to *apt install python3-segyio* to installing segyio on their systems.

- pybind11 >= 2.6
- C++ compiler (GNU or Intel) with support for std++14 or newer.
- cmake >= 3.0
- CGAL >= 5.0.0 which requires:
  - MPFR
  - GMP
  - Boost > 1.4.8

Note: CGAL requires Boost, MPFR and GMP. These packages may already be installed on your standard Linux box.

Warning: Make sure your package manager is downloading CGAL >= 5.0 otherwise you will not be able to install *SeismicMesh*!

1.3.2 Compilation by source

After installing all dependencies, perform

```
$  pip install -e .
```

Note: If you do not have administrative rights on your system, add the flag `--user` to the command above.

Warning: With this said, the preferred method of installation using pypi: pip install SeismicMesh

1.3.3 Testing

Testing is accomplished with *tox*. The *tox* package can be installed like so:

```
pip install tox
```

To test the installation, serial and parallel capabilities, you can use *tox* from the top directory of the package:

```
$ tox
```

1.3.4 Installation on Clusters

Note: Make sure the CXX environment variable points to your intended compiler!

If installing on a cluster by source with a local installation of `CGAL` and `Boost`, you'll need to edit `setup.cfg` with the CMake arguments so to point the installation to the correct directories. Namely, in `setup.py` you'll have to edit the list called `cmake_args` to include

```
-DCMAKE_CXX_COMPILER=+/PATH/TO/CPPCOMPILER

-DBoost_INCLUDE_DIR=+/PATH/TO/BOOST/

-DMPFR_LIBRARIES=+/PATH/TO/libmpfr.a

-DMPFR_INCLUDE_DIR=+/PATH/TO/MPFR/include
```

Warning: Under construction. Contributions very welcome!

1.4 Basics

SeismicMesh supports the generation of both 2D and 3D meshes in either serial or parallel from seismic velocity models.

Here I show how to build meshes from sizing functions created with the software and explain what the sizng options mean. The API for serial/parallel and 2D/3D is identical.

Assuming you've coded a short Python script to call *SeismicMesh* (similar to what is shown in the examples), you simply call the script with python for serial execution:

```
python your_script.py
```

Distributed memory parallelism can be used by first writing an extra import statement for `mpi4py` (`import mpi4py`) near your other imports. Following this write the following line near the top of your script before you call the *SeismicMesh* API):

```
comm = MPI.COMM_WORLD
```

Note: This line has no effect on serial execution and its fine to leave it in if you intend to only use serial execution.

Parallel execution takes place by typing by:

```
mpiexec -n N python your_script.py
```

where *N* is the number of cores (e.g., 2,3,4 etc.)

Warning: Oversubscribing the mesh generation problem to too many cores will surely lead to problems and slow downs. In general, keeping the minimum number of vertices per rank to between 20-50k/rank results in optimal performance.

1.4.1 Example data

Note: Users should create a directory called *velocity_models* and place their seismic velocity models there.

A 2D model (BP2004):

```
wget http://s3.amazonaws.com/open.source.geoscience/open_data/bpvelanal2004/vel_z6.
↪25m_x12.5m_exact.segy.gz
```

A 3D model (EAGE Salt):

```
https://s3.amazonaws.com/open.source.geoscience/open_data/seg_eage_models_cd/Salt_
↪Model_3D.tar.gz
```

1.4.2 File I/O and visualization of meshes

Meshes can be written to disk in a variety of formats using the Python package *meshio* (https://pypi.org/project/meshio/).

Warning: Note that *SeismicMesh* sizing function makes the assumption that the first dimension is z and the second is x while the third is y. This is done in this way since 2D seismological simulations take place in the z-x plane and 3D in the z-x-y plane. As a result, the meshes when loaded into visualization software will appear rotated 90 degrees. For visualization, we can output in the vtk format using MeshIO (as shown in the examples) and then load the vtk file into ParaView.

1.4.3 Some things to know

This seismic velocity model is passed to the *get_sizing_function_from_segy* along with the domain extents

```
from SeismicMesh import get_sizing_function_from_segy

# Construct a mesh sizing function from a seismic velocity model
ef = get_sizing_function_from_segy(fname, bbox, other-args-go-here,...)
```

- The user specifies the filename *fname* of the seismic velocity model (e.g., either SEG-y or binary)
- The user specifies the domain extents of the *velocity model* as a tuple of coordinates in meters representing the corners of the domain:

$$bbox = (z_{min}, z_{max}, x_{min}, x_{max})$$

- In 3D:

$$bbox = (z_{min}, z_{max}, x_{min}, x_{max}, y_{min}, y_{max})‘$$

1.4.4 Geometry

SeismicMesh can mesh any domain defined by a signed distance function. However, we provide some basic domain shapes: a rectangle, a cube, or a disk.

For example:

```
from SeismicMesh import Rectangle, Cube, Disk

rectangle = Rectangle(bbox)
cube = Cube(bbox)
disk = Disk(x0=[0,0],r=1) # center of (0,0) with a radius of 1.0
```

Note: A good reference for various signed distance functions can be found at: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

1.4.5 Mesh size function

The notion of an adequate mesh size is determined by a combination of the physics of acoustic/elastic wave propagation, the desired numerical accuracy of the solution (e.g., spatial polynomial order, timestepping method, etc.), and allowable computational cost of the model amongst other things. In the following sub-sections, each available mesh sizing strategy is briefly described and pseudo code is provided.

Note: The final mesh size map is taken as the minimum of all supplied sizing functions.

Note: The mesh size map dictates the triangular edge lengths in the final mesh (assuming these triangles will be equilateral).

Wavelength-to-gridscale

The highest frequency of the source wavelet f_{max} and the smallest value of the velocity model v_{min} define the shortest scale length of the problem since the shortest spatial wavelength λ_{min} is equal to the $\frac{v_{min}}{f_{max}}$. For marine domains, v_{min} is approximately 1,484 m/s, which is the speed of sound in seawater, thus the finest mesh resolution is near the water layer.

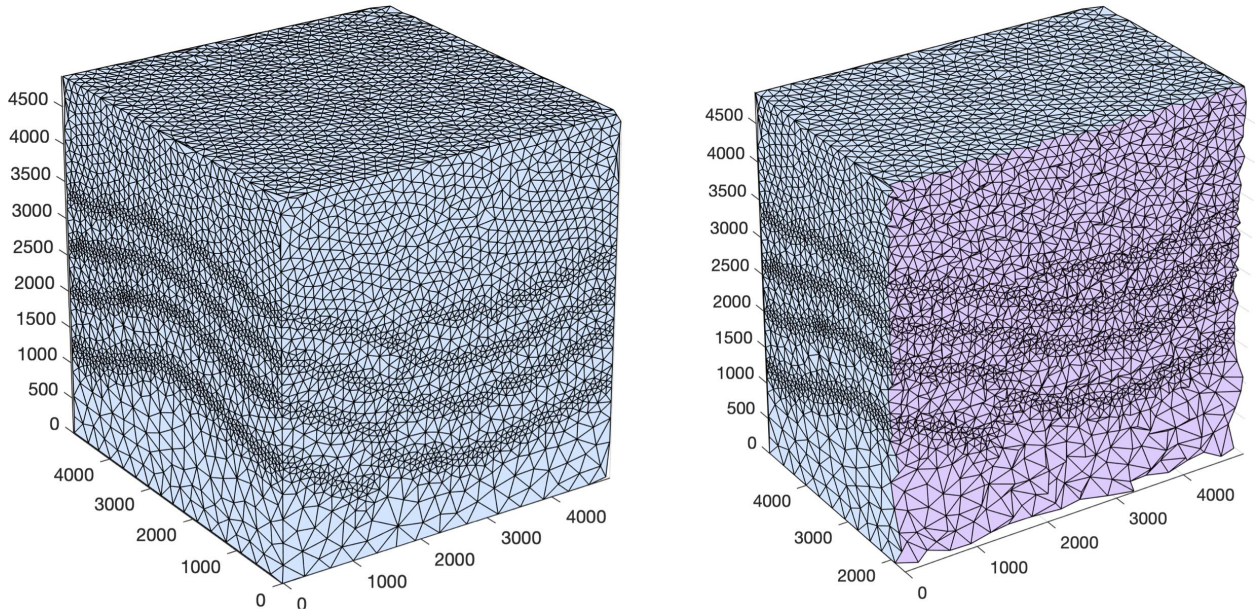
The user is able to specify the number of vertices per wavelength α_{wl} the peak source frequency f_{max}. This sizing heuristic also can be used to take into account varying polynomial orders for finite elements. For instance if using quadratic P=2 elements, α_{wl} can be safely be set to 5 to avoid excessive dispersion and dissipation otherwise that would occur with P=1 elements:

```
# Construct mesh sizing object from velocity model
ef = get_sizing_function_from_segy(fname, bbox,
    wl=3, # :math:`\alpha_{wl}` number of grid points per wavelength
    freq=2, # maximum source frequency for which the wavelength is calculated
)
```

Resolving seismic velocity gradients

Seismic domains are known for sharp gradients in material properties, such as seismic velocity. These sharp gradients lead to reflections and refractions in propagated waves, which are critical for successful imaging. Thus, finer mesh resolution can be deployed inversely proportional to the local standard deviation of P-wave velocity. The local standard deviation of seismic P-wave velocity is calculated in a sliding window around each point on the velocity model. The user chooses the mapping relationship between the local standard deviation of the seismic velocity model and the values of the corresponding mesh size nearby it. This parameter is referred to as the $grad$ and is specified in meters. For instance a $grad$ of 50 would imply that the largest gradient in seismic P-wave velocity is mapped to a minimum resolution of 50-m.:

```
ef = get_sizing_function_from_segy(fname, bbox,
    grad=50, # the desired mesh size in meters near the sharpest gradient in the␣
↪domain
)
```



Courant-Friedrichs-Lewey (CFL) condition

Almost all numerical wave propagators utilize explicit time-stepping methods in the seismic domain. The major advantage for these explicit methods is computational speed. However, it is well-known that all explicit or semi-explicit methods require that the Courant number be bounded above by the Courant-Friedrichs-Lewey (CFL) condition. Ignoring this condition will lead to a numerically unstable simulation. Thus, we must ensure that the Courant number is indeed bounded for the overall mesh size function.

After sizing functions have been activated, a conservative maximum Courant number is enforced.

For the linear acoustic wave equation assuming isotropic mesh resolution, the CFL condition is commonly described by

$$C_r(x) = \frac{(\Delta t * v_p(x))}{dim * h(x)}$$

where h is the diameter of the circumball that inscribes the element either calculated from $f(h)$ or from the actual mesh cells, dim is the spatial dimension of the problem (2 or 3), Δt is the intended simulation time step in seconds and v_p is the local seismic P-wave velocity. The above equation can be rearranged to find the minimum mesh size possible for a given v_p and Δt, based on some user-defined value of $Cr \leq 1$. If there are any violations of the CFL, they can bed edited before building the mesh so to satisfy that the maximum Cr is less than some conservative threshold. We normally apply $Cr = 0.5$, which provides a solid buffer but this can but this can be controlled by the user like the following:

```
ef = get_sizing_function_from_segy(fname, bbox,
    cr=0.5, # maximum bounded Courant number to be bounded in the mesh sizing function
    dt=0.001, # for the given :math:`\Delta t` of 0.001 seconds
    ...
)
```

Further, the space order of the method (p) can also be incorporated into the above formula to consider the higher spatial order that the simulation will use:

```
ef = get_sizing_function_from_segy(fname, bbox,
    cr=0.5, # maximum bounded Courant number :math:`Cr_{max}` in the mesh
    dt=0.001, # for the given :math:`\Delta t` of 0.001 seconds
    space_order = 2, # assume quadratic elements :math:`P=2`
    ...
)
```

The above code implies that the mesh will be used in a simulation with $P = 2$ quadratic elements, and thus will ensure the Cr_{max} is divided by $\frac{1}{space\_order}$

Mesh size gradation

In regions where there are sharp material contrasts, the variation in element size can become substantially large, especially using the aforementioned sizing strategies such as the wavelength-to-gridscale. Attempting to construct a mesh with such large spatial variations in mesh sizes would result in low-geometric quality elements that compromise the numerical stability of a model.

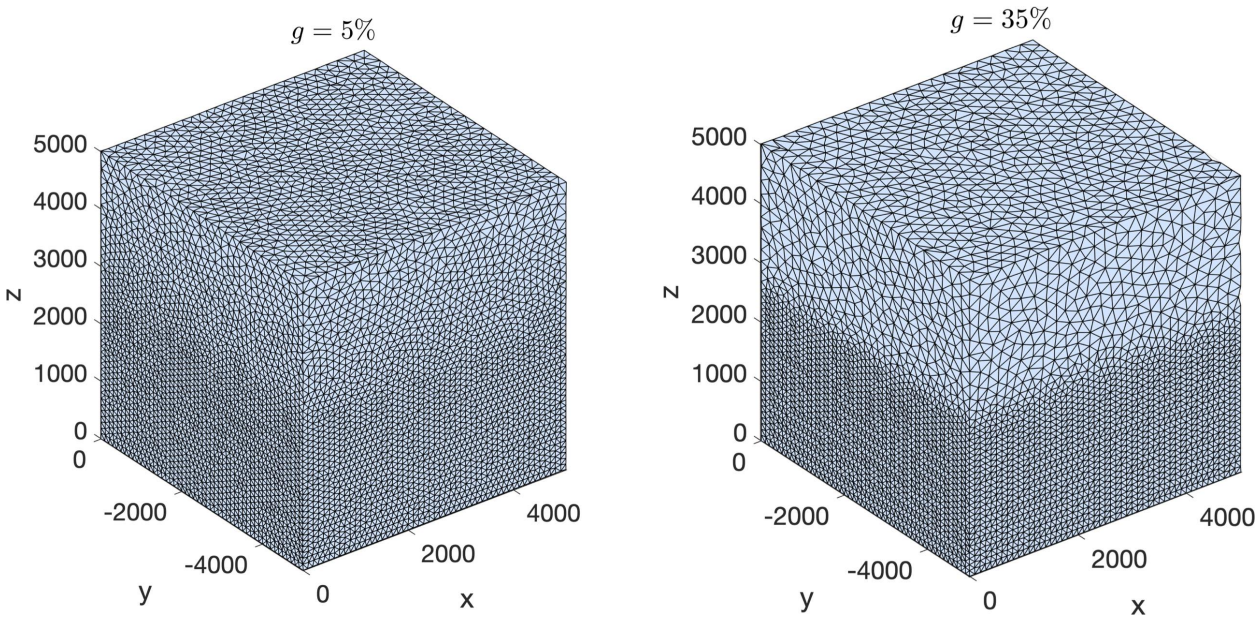
Thus, the final stage of the development of a mesh size function $h(x)$ involves ensuring a size smoothness limit, g such that for any two points x_i, x_j, the local increase in size is bounded such as:

$$h(\boldsymbol{x_j}) \leq h(\boldsymbol{x_i}) + \alpha_g ||\boldsymbol{x_i} - \boldsymbol{x_j}||$$

A smoothness criteria is necessary to produce a mesh that can simulate physical processes with a practical time step as sharp gradients in mesh resolution typically lead to highly skewed angles that result in poor numerical performance.

We adopt the method to smooth the mesh size function originally proposed by [grading]. A smoother sizing function is congruent with a higher overall element quality but with more triangles in the mesh. Generally, setting $0.2 \leq \alpha_g \leq 0.3$ produces good results:

```
ef = get_sizing_function_from_segy(fname, bbox,
    ...
    grade=0.15, # :math:`g` cell-to-cell size rate growth bound
    ...
)
```



Domain padding

Note: It is assumed that the top side of the domain represents the free-surface thus no domain padding applied there.

In seismology applications, the goal is often to model the propagation of an elastic or acoustic wave through an infinite domain. However, this is obviously not possible so the domain is approximated by a finite region of space. This can lead to undesirable artificial reflections off the sides of the domain however. A common approach to avoid these artificial reflections is to pad the domain and enforce absorbing boundary conditions in this extension. In terms of meshing to take this under consideration, the user has the option to specify a domain extension of variable width on all three sides of the domain like so:

```
ef = get_sizing_function_from_segy(fname, bbox,
    domain_pad=250, # domain will be pad by 250-m on all three sides of the domain
    ...
)
```

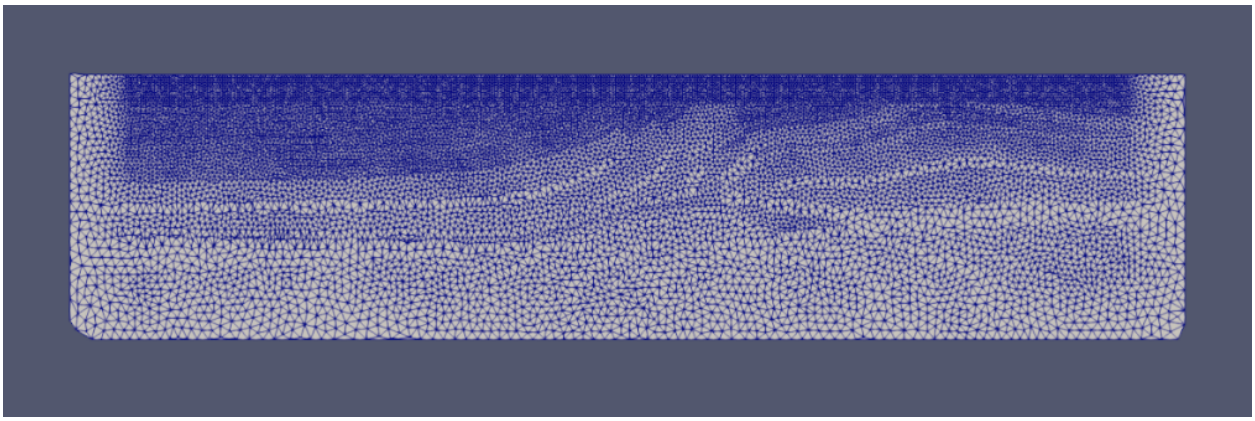
In this domain pad, mesh resolution can be adapted according to following three different styles.

- `Linear` - pads the seismic velocities on the edges of the domain linearly increasing into the domain pad.
- `Constant` - places a constant velocity of the users selection in the domain pad.
- `Edge` - pads the seismic velocity about the domain boundary so that velocity profile is identical to its edge values.

An example of the `edge` style is below:

```
ef = get_sizing_function_from_segy(fname, bbox,
    domain_pad=250, # domain will be extended by 250-m on all three sides
    padstyle="edge", # velocity will be extend from values at the edges of the domain
    ...
)
```

Note: In our experience, the `edge` option works the best at reducing reflections with absorbing boundary conditions.



1.4.6 Mesh generation

After building your signed distance function and sizing function, call the `generate_mesh` function to generate the mesh:

```
points, cells = generate_mesh(domain=rectangle, edge_length=ef, h0=hmin)
```

Note: *ef* is a sizing function created using get_sizing_function_from_segy

You can change how many iterations are performed by altering the kwarg *max_iter*:

```
points, cells = generate_mesh(domain=rectangle, edge_length=ef, h0=hmin, max_iter=100)
```

Note: Generally setting max_iter to between 50 to 100 iterations produces a high quality triangulation. By default it runs 50 iterations.

When executing in parallel, the user can optionally choose which axis (0, 1, or 2 [if 3D]) to decompose the domain:

```
points, cells = generate_mesh(domain=cube, edge_length=ef, h0=hmin, max_iter=100,␣
↪axis=2)
```

Note: Generally axis=1 works the best in 2D or 3D since typically mesh sizes increase in size from the free surface to the depths of the model. In this situation, the computational load tends to be better balanced.

1.4.7 Mesh improvement (*sliver* removal)

3D *Sliver* removal

It is strongly encouraged to run the sliver removal method by passing the point of set of a previously generated mesh:

```
points, cells = sliver_removal(
    points=points, domain=cube, h0=minimum_mesh_size, edge_length=ef
)
```

Note: Please remember to import this method at the top of your script (e.g., *from SeismicMesh import sliver_removal*)

By default, `min_dh_angle_bound` is set to 10. The sliver removal algorithm will attempt 50 iterations but will terminate earlier if no slivers are detected. Generally, if more than 50 meshing iterations were used to build the mesh, this algorithm will converge in 10-20 iterations.

Warning: Do not set the minimum dihedral angle bound greater than 15 unless you've already successfully ran the mesh with a lower threshold. Otherwise, the method will likely not converge.

References

1.5 Modules

Here we document the public API

1.5.1 *SeismsicMesh.geometry*

Routines to perform geometrical/topological operations and calculate things on meshes.

1.5.2 *SeismsicMesh.sizing*

Function to build a $f(h)$ mesh sizing function from a seismic velocity model. Assumes the domain can be represented by a rectangle (2D) or cube (3D) and thus builds a $f(d)$ accordingly.

1.5.3 *SeismsicMesh.generation*

Functions to build and improve a simplical mesh that conforms to the signed distance function $f(d)$ and $f(h)$.

Bibliography

[hpc_del] Peterka, Tom, Dmitriy Morozov, and Carolyn Phillips. "High-performance computation of distributed-memory parallel 3D Voronoi and Delaunay tessellation." SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2014.

[distmesh] P.-O. Persson, G. Strang, A Simple Mesh Generator in MATLAB. SIAM Review, Volume 46 (2), pp. 329-345, June 2004 (PDF)

[firedrake] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. ACM Trans. Math. Softw., 43(3):24:1–24:27, 2016. URL: http://arxiv.org/abs/1501.01809, arXiv:1501.01809, doi:10.1145/2998441.

[cgal] The CGAL Project. CGAL User and Reference Manual. CGAL Editorial Board, 5.0.2 edition, 2020

[slivers] Tournois, Jane, Rahul Srinivasan, and Pierre Alliez. "Perturbing slivers in 3D Delaunay meshes." Proceedings of the 18th international meshing roundtable. Springer, Berlin, Heidelberg, 2009. 157-173.

[grading] Persson, Per-Olof. "Mesh size functions for implicit geometries and PDE-based gradient limiting." Engineering with Computers 22.2 (2006): 95-109.