NOAA Technical Memorandum NOS CS 47

OCSMesh: a data-driven automated unstructured mesh generation software for coastal ocean modeling

Silver Spring, Maryland
December 2021

noaa National Oceanic and Atmospheric Administration

U.S. DEPARTMENT OF COMMERCE
National Ocean Service
Coast Survey Development Laboratory

Office of Coast Survey
National Ocean Service
National Oceanic and Atmospheric Administration
U.S. Department of Commerce

The Office of Coast Survey (OCS) is the Nation's only official chartmaker. As the oldest United States scientific organization, dating from 1807, this office has a long history. Today it promotes safe navigation by managing the National Oceanic and Atmospheric Administration's (NOAA) nautical chart and oceanographic data collection and information programs.

There are four components of OCS:

> **The Coast Survey Development Laboratory develops new and efficient techniques to accomplish Coast Survey missions and to produce new and improved products and services for the maritime community and other coastal users.**
>
> **The Marine Chart Division acquires marine navigational data to construct and maintain nautical charts, Coast Pilots, and related marine products for the United States.**
>
> **The Hydrographic Surveys Division directs programs for ship and shore-based hydrographic survey units and conducts general hydrographic survey operations.**
>
> **The Navigational Services Division is the focal point for Coast Survey customer service activities, concentrating predominately on charting issues, fast-response hydrographic surveys, and Coast Pilot updates.**

NOAA Technical Memorandum NOS CS 47

OCSMesh: a data-driven automated unstructured mesh generation software for coastal ocean modeling

Soroosh Mani[1, 2], Jaime R. Calzada[1, 3], Saeed Moghimi[1, 4], Y. Joseph Zhang[3], Edward Myers[1], Shachak Pe’eri[1]

1. Office of Coast Survey, Coast Survey Development Laboratory, Silver Spring, Maryland
2. Spatial Front Inc., Bethesda, Maryland
3. Virginia Institute of Marine Science, Gloucester Point, Virginia
4. University Corporation for Atmospheric Research, Boulder, Colorado

December 2021

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF CODE SNIPPETS

ABSTRACT

In this document we introduce OCSMesh, an unstructured mesh generation Python tool for coastal modeling application. OCSMesh uses an object oriented programming paradigm for generating the computational domain and mesh size definitions. In its current release, OCSMesh uses Jigsaw as the meshing engine. This release targets SCHISM (Semi-implicit Cross-scale Hydro-science Integrated System Model) coastal ocean model; however the generated triangular mesh can be used for any solver. OCSMesh has the capability to handle common data formats for digital elevation model data (DEM) as well as coverage definitions such as polygons, lines, and points. OCSMesh also supports some of the commonly used mesh formats in coastal modeling, such as Surface-water Modeling System (SMS) 2DM format and ADvanced CIRCulation (ADCIRC)/SCHISM GRD format. The process of generating a mesh using OCSMesh involves creating objects to define geometry, i.e., the domain of simulation, as well as the element size for the domain. The result is a mesh object that can be further manipulated through the API for operations such as extending the domain, adding mesh refinement based on different criteria as well as interpolating DEM data on the mesh results. The definition of geometry objects in OCSMesh can be based on different underlying types such as rasters, mesh, or coverage data (e.g., polygons and multi-polygons). Similarly, size functions can be constructed based on either raster data or mesh data. This flexible framework gives the user the ability to define the meshing process in multiple separate stages to achieve computation efficiency. It also helps with starting the meshing process from an intermediate stage in case something fails. OCSMesh classes are composed of objects from commonly used Python libraries. Because of this one can easily extract the objects from OCSMesh, modify them using other available tools, and then pass them back to OCSMesh for further processing. The result is more flexibility for the users to implement custom workflows and automate the mesh generation process. The document ends by providing two examples of OCSMesh use cases.

Key Words: mesh, unstructured, data-driven, coastal modeling, automated, SCHISM

1. Introduction

Mesh generation has always been a crucial and time consuming step for modelers who would rather focus on getting the physics of their simulation right, than spending time on tweaking nodes and elements. The Coastal Marine Modeling Branch at NOAA Office of Coast Survey uses many coastal ocean models to make predictions about different coastal processes for purposes such as helping with precise navigation or coastal flooding disaster mitigation. Some of these models rely on an unstructured mesh, hence the need for a tool to generate or modify this type of mesh automatically.

OCSMesh addresses this issue by providing the building blocks to automate mesh generation, cleanup and interpolation process. It can automatically process multiple tiles of topobathy data, i.e., digital elevation model (DEM), and generate simulation domain and element-size function. While automation is one of the strengths of OCSMesh, it also provides tools that help Geographic Information Systems (GIS) and modeling experts incorporate their domain expertise into the mesh generation process by providing the ability to add custom regions of refinement, use multistage meshing and incorporate local refinements.

OCSMesh relies on Jigsaw-Python, a Python wrapper for Jigsaw library, as its meshing engine (Engwirda, 2014).

The current OCSMesh target solver is SCHISM (Zhang & Baptista, 2008; Zhang, Ye, Stanev, & Grashorn, 2016). However, it can be used to generate unstructured triangular mesh for any coastal modeling solver. With that in mind, we proceed to showcase using OCSMesh for SCHISM in this document. The core idea in OCSMesh is to represent the DEM as closely as possible, so the main workflow is designed around this idea. However, additional functionality is added to support custom workflows such as starting from existing mesh or Shapefiles.

OCSMesh code can be found at: https://github.com/noaa-ocs-modeling/OCSMesh

2. Terminology

This section provides a generic guidance on the terminology used throughout the rest of the document.

- **Geometry/Modeling/Simulation domain**: From now on, "domain" defines the area that needs to be meshed. Usually it includes ocean, estuaries, rivers and floodplains.
- **Geom (object)**: The object representing the domain in OCSMesh. It is a singleton per mesh generation. It also is an umbrella term for the object that handles different kinds of geometry definitions.
- **Size function/Hfun (object)/Hmat (object)**: The singleton object (per mesh) that defines the element size in different points of the domain. Usually the term "Hmat" is used when the element size is defined on a structured grid and "Hfun" is used when sizes are defined on an unstructured mesh. Hfun is also more broadly used as a shorthand for size-function in this document.
- **Mesh/Mesh object**: It is an object that defines mesh points and vertices in OCSMesh. This object provides the ability to manipulate resulting mesh and interpolate DEM values on mesh nodes.
- **Command Line Interface (CLI)/Entry point**: The OCSMesh interface for command line scripting
- **Parsers**: File reader/writer for different mesh formats; right now `.grd` format and Aquaveo SMS`.2dm` (Aquaveo, 2017) are supported.
- **DEM/Digital Elevation Model**: Computer representation of elevation data obtained in various ways (e.g., LIDAR, RADAR, Imaging) such as NCEI CUDEM (Cooperative Institute for Research in Environmental Sciences (CIRES) at the University of Colorado, Boulder, 2014), GEBCO (GEBCO Bathymetric Compilation Group 2020, 2020).
- **Raster (object)**: Raster is a format for input DEM data. In this document it also refers to the objects created by OCSMesh to handle this type of data.
- **Jigsaw mesh/msh_t object:** This refers to `msh_t` type defined by Jigsaw-Python which is a representation of mesh based on vertices as points in space, and edges, triangles, quadrilateral elements, etc. as defined by vertex index.
- **Coordinates reference system (CRS)**: The set of projections and transformations used to locally or globally display 3D points of earth surface on a 2D map.

3. Features

This section provides an itemized overview of some of OCSMesh's features.

- Contour Extraction from DEM
- Multi-tile DEM handling (including overlaps)
- Coordinates reference system (CRS) handling
- Contour-based refinement
- Topography based refinement
- Custom region refinement
- Channel detection and refinement
- Interoperate with other Python mesh and GIS tools
- Automation building blocks

4. Quick Start

This section serves as a “Getting Started” guide for OCSMesh. It demonstrates a high level idea of how OCSMesh handles meshing definitions and results.

In the example below the following steps are explained:

- Identifying the domain (where to mesh)
- Specifying the minimum and maximum size of elements as well as refinements
- Mesh generation and interpolation of topobathy data on the mesh

The domain can be defined in multiple different ways. In this case, it is identified by a single DEM Raster file and the maximum level contour within the DEM. The level needs to be specified in meters above sea level (positive up).

OCSMesh supports reading rasters through RasterIO (Gillies, Rasterio, 2021). This means formats such as GeoTIFF and NetCDF are readily supported.

```
from ocsmesh import Raster, Geom, Hfun, Mesh, JigsawDriver, utils

raster_for_geom = Raster(“/path/to/raster.tif”)
geom = Geom(raster_for_geom, zmax=20)
```

Code Snippet 1 Creating geometry object from a raster file

After specifying the domain, element sizes need to be specified. Similar to geometry, a raster (the same as in geometry) can be used to create the size function.

```
raster_for_hfun = Raster(“/path/to/raster.tif”)
hfun = Hfun(raster_for_hfun, hmin=100, hmax=8000)
```

Code Snippet 2 Creating size function from a raster file

Please note:

1. The minimum and maximum sizes are both specified in meters, regardless of raster CRS
2. Two separate raster objects are used to create the geometry and size function. Raster objects are Python mutable objects. That means if the geometry or the size function result in a programmatic “side-effect”, raster data could be corrupted for use by other objects.

```
# Don’t do this!!
raster = Raster(“/path/to/raster.tif”)
geom = Geom(raster, zmax=20)
hfun = Hfun(raster, hmin=100, hmax=8000)
```

Code Snippet 3 What to avoid when initializing geometry and size function from a raster object

Note that specifying a minimum global size for the size function only relaxes (not enforces) the meshing criteria. In other words, it does not result in mesh refinement. In order to get smaller elements than the global maximum mesh size, refinements need to be specified. The details of each type of refinement is discussed later in this document. Code Snippet 4 shows how refinement can be specified for the size function:

```
# Topography gradient based refinement
hfun.add_subtidal_flow_limiter()

# Constant value of 100 meters for elevations above 0 meter msl
hfun.add_constant_value(100, 0)

# Value of 100 meters for 0 meter msl contour and 200m for -10m msl with
# expansion rate of 1/1000
hfun.add_contour(0, 0.001, 100)
hfun.add_contour(-10, 0.001, 200)
```

Code Snippet 4 How to add mesh size refinement to a size function object

Before passing the geometry and size function to the mesh driver, their results can be extracted and inspected. Code Snippet 5 shows how this can be achieved.

```
# Use geopandas to visualize shapely geometry
import geopandas as gpd

multipolygon = geom.get_multipolygon()
gpd.GeoSeries(multipolygon, crs=geom.crs).plot()


hfun_msh_t = hfun.msh_t()
utils.tricontourf(hfun_msh_t)
```

Code Snippet 5 How to visualize generated geometry and size function objects

The geometry multipolygon is calculated every time `get_multipolygon()` is called. Thus, if the geometry is plotted first and then passed to the driver (which in turn calls `get_multipolygon()`) it will result in calculating the polygon twice. This computationally expensive double calculation (especially in the case of multi-DEM collector objects) can be avoided. However this topic is beyond the scope of this document.

Code Snippet 6 shows how to generate the mesh.

```
driver = JigsawDriver(geom, hfun)
mesh = driver.run()
```

Code Snippet 6 Command to generate the mesh from geometry and size function

The result is the mesh without any elevation data. Code Snippet 7 demonstrates how to interpolate the elevations from rasters.

```
raster_for_interp = Raster("/path/to/raster.tif")
list_of_rasters = [raster_for_interp]
mesh.interpolate(list_of_rasters)
```

Code Snippet 7 Command to interpolate DEMs on the generated mesh

As shown in the Code Snippet 7, `interpolate()` accepts a list to interpolate multiple rasters. The last raster in the list has the highest priority. The data from rasters with higher priority takes precedence over data from lower priority rasters in case of overlap.

After interpolation the mesh object can be written to a file. Note that currently OCSMesh only supports writing mesh to disk in EPSG 4326.

```
mesh.write("/path/to/output", format="2dm", overwrite=True)
```

Code Snippet 8 Write mesh to the disk

This example laid out the generic workflow for using OCSMesh. Some more advanced use cases such as using multiple DEMs or using an existing mesh to specify the new domain are discussed in the Examples section.

Note that usually CRS is handled automatically. However, there are cases where CRS considerations are important.

5. Meshing Process

Overview

This section discusses the process of generating a mesh in more detail. This involves 5 main steps:

1. Generating geometry; the domain that needs to be meshed
2. Generating size function specifying how large the elements should be at different points in the domain
3. Mesh generation using Jigsaw library
4. Mesh cleanup: Jigsaw doesn't always generate a simulation ready mesh. Because of this, the output mesh needs to be cleaned up by inspection and remeshing iterations. This is done automatically in OCSMesh.
5. Interpolating topo/bathy data on the cleaned up mesh
   a. Boundary segment extraction could either be considered a part of meshing, or preprocessing of the simulation. To generate boundaries in OCSMesh, the user specifies an elevation threshold for detecting open ocean boundary.

Units and CRS Consideration

During Euclidean meshing in Jigsaw, there's no concept of units, CRS or projection. In other words the geometry is meshed using the size values given by the size function. That means OCSMesh needs to take care of passing the values with consistent units and CRS to Jigsaw. OCSMesh uses meters for specifying mesh size, whether it's the global size or local refinement target size. The output for geometry and mesh-size coordinates need to be in meters, too.

OCSMesh Jigsaw driver uses Jigsaw-Python's `msh_t` representation for geometry, size function, and mesh objects. The `msh_t` objects are retrieved from geometries and size-functions by calling their `msh_t()` method. This method checks if the geometry or size-function is "geographic" (i.e., non-projected). If it is, then it finds the estimate Universal Transverse Mercator (UTM) CRS of the geometry or size function based on the center of their bounding boxes and transforms all `msh_t` objects into that UTM CRS. On the other hand, if the CRS is not geographic it is assumed that the values are in consistent length units and no transformation is needed.

This transformation logic is implemented for all types of geometry and size function (raster, mesh, polygon, and collector) in the `msh_t()` method. On the other hand, the polygon retrieval methods (e.g., `get_multipolygon()` for geometry objects) return polygons in the original CRS of the objects, not in UTM.

In cases where the UTM projection is not well suited (e.g., large regions or cross-day-line) there is currently no built-in solution other than projecting input geometry and size function prior to passing them to the meshing driver.

Meshing Setup

Geometry

The process of meshing in OCSMesh starts with defining the domain. The geometry can be defined in a couple of different ways:

- Based on raster
- Based on mesh
- Based on polygon (shapely objects)
- Collection of all other types (collector type)

Raster-based

Raster based geometry defines the domain polygon by finding polygon(s) bounded from one or two sides by specified elevation contour(s) extracted from the input raster. This type of geometry is created from an OCSMesh raster object (see Raster section) through which different operations are provided, such as reprojecting and clipping. Note that when a geometry is created from a raster, it doesn't copy the raster object. Instead it just holds a reference to the original object. As a result, any modification to the underlying raster within the geometry object has an impact on other objects that use the same raster object and vice versa.

Example code below demonstrates how to create a raster based geometry:

```
from ocsmesh import Raster, Geom

raster = Raster("/path/to/raster.tif")
geom = Geom(raster, zmax=20)
```

Code Snippet 9 Create raster based geometry object

The `zmax` in the example above is used to extract the polygons bounded by elevation of 20m from the input DEM. In theory one could also specify `zmin`, but specifying minimum elevation (i.e., max depth) is rarely used.

The polygon of this geometry object can be retrieved as show in Code Snippet 10.

```
multi_polygon = geom.get_multipolygon()
```

Code Snippet 10 Get Shapely multipolygon of the geometry object

Note that the polygon calculation is executed when this method gets called (lazy evaluation).

Jigsaw doesn't accept Shapely objects as input. It requires a custom object that represents the geometry in vertex-edge format. To get the geometry in the proper format, one can use msh_t() method (Code Snippet 11).

```
geom_msh_t = geom.msh_t()
```

Code Snippet 11 Get Jigsaw mesh object from the geometry object

OCSMesh Jigsaw driver automatically calls this method when passing geometry information to Jigsaw. As shown later, the same Application Programming Interface (API) is defined for size-function and mesh objects.

Mesh-based

Mesh based geometry defines the domain by finding the polygon tiled by all the elements in the input mesh. This type of geometry is mostly a tool for convenience. It can be used when the user already has an existing mesh. The resulting geometry can then be combined with a few regional raster based geometries to generate the final domain. This approach is explained in Local Refinement section.

```
from ocsmesh import Mesh, Geom

mesh = Mesh.open("/path/to/mesh.14")
geom = Geom(mesh)
```

Code Snippet 12 Create mesh based geometry object

Mesh based geometry supports contour extraction as well, but the feature works differently compared to the raster based geometry. Multipolygon and mesh representation of the mesh-based geometry can be extracted using the same methods as raster (get_multipolygon() and msh_t() respectively).

Polygon-based

Polygon based geometry defines the domain based on the underlying Shapely polygon object. This type of geometry serves a similar purpose as mesh-based geometry. The user can execute a set operations on some existing polygons to reach a desired domain shape, then create a polygon based geometry from it (Code Snippet 13).

```
from ocsmesh import Geom
from shapely.geometry import Polygon

poly = Polygon([[0, 0], [1, 0], [1, 1], [0, 1]])
geom = Geom(poly)
```

Code Snippet 13 Create a Shapely geometry based geometry object

Collector type

Usually combining different types of geometries requires a decent understanding of OCSMesh and its underlying packages. The collector geometry automates this process by accepting an input list of different geometry types and optionally a base mesh or polygon, and processing and combining them as described later in this section.

Sometimes there’s an existing polygon or mesh from older simulations. This polygon or mesh can be used to limit what part of the input DEMs to consider when extracting the domain. This feature is demonstrated in Examples section.

```
from ocsmesh import Geom, Raster
from shapely.geometry import Polygon

poly = Polygon([[0, 0], [1, 0], [1, 1], [0, 1]])
raster1 = Raster("/path/to/raster1.tif")
raster2 = Raster("/path/to/raster2.tif")
raster3 = Raster("/path/to/raster3.tif")

geom = Geom([poly, raster1, raster2, raster3], zmax=20)
```

Code Snippet 14 Create a collector type geometry object

Collector geometry defines an API similar to the raster-based geometry to extract the polygon and the mesh representation.

Combine process

Collector geometry processes its inputs by executing the following steps:

1. A list of polygons (both interior and exterior) is extracted from the base mesh or base polygon, if provided.
2. The specified contour is extracted from each input geometry for the domain boundary
    - In case of non-raster types their multipolygon is used without any contour extraction.
3. If defined, patches are extracted from the specified sources.
    - For geometry collectors, patches are masked rasters from which polygons are extracted locally.

    - For example from 10 input tiles the requirement can be to extract 0m contour from the first 5 and 10m contour from the other 5.
4. The line segments from the base mesh or polygon that lie under the raster are removed, so that the new contour lines calculated from the geometry input data override them.
5. The resulting shapes from all geometry inputs and the patches along with the optional input base mesh or base polygon are unioned to get the final geometry.

Geometry-combine functionality is also available through the Command Line Interface (CLI). However, some of the advanced features are only accessible through the Python API.

A note on union priority

Note that currently no explicit priority checking is used during geometry creation due to performance concerns. That means if a low resolution raster A and a high resolution raster B are passed to the geometry creation algorithm, both have the same importance in producing the final polygon, and their polygons are combined (union) in case of an overlap. With priority processing enabled, the first item in the list will have the lowest priority and the last one will have the highest priority. Figure 1 and Figure 2 show how priority processing should work. Figure 1 shows two overlaying polygons which should be unioned to form the final geometry. The yellow transparent polygon shows the higher priority DEM geometry and the solid blue polygon under it shows the lower priority DEM geometry. The black line shows the bounding box of the higher priority DEM. During priority processing, all the lower priority polygons within the high priority DEM bounding box are ignored. Figure 2 depicts the difference between union operations when using priority checking versus when no priority is used. The green contour shows the polygons from Figure 1 when unioned with priority checking. The red small polygons show the difference when priority is not taken into account (current implementation).

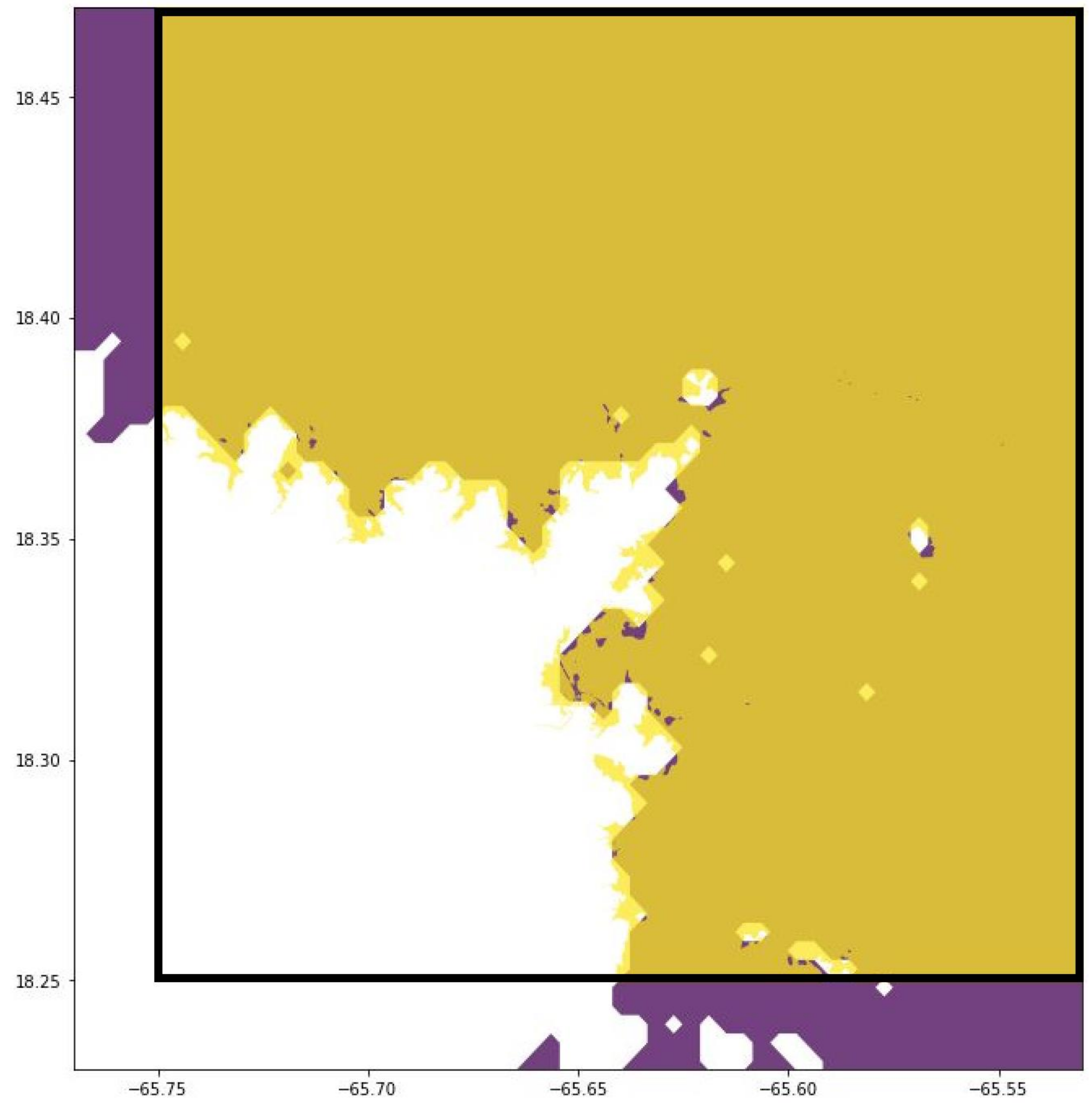

Figure 1 Two overlaying DEMs to be combined (yellow: higher priority, blue: lower priority, black line: high priority DEM bounding box)

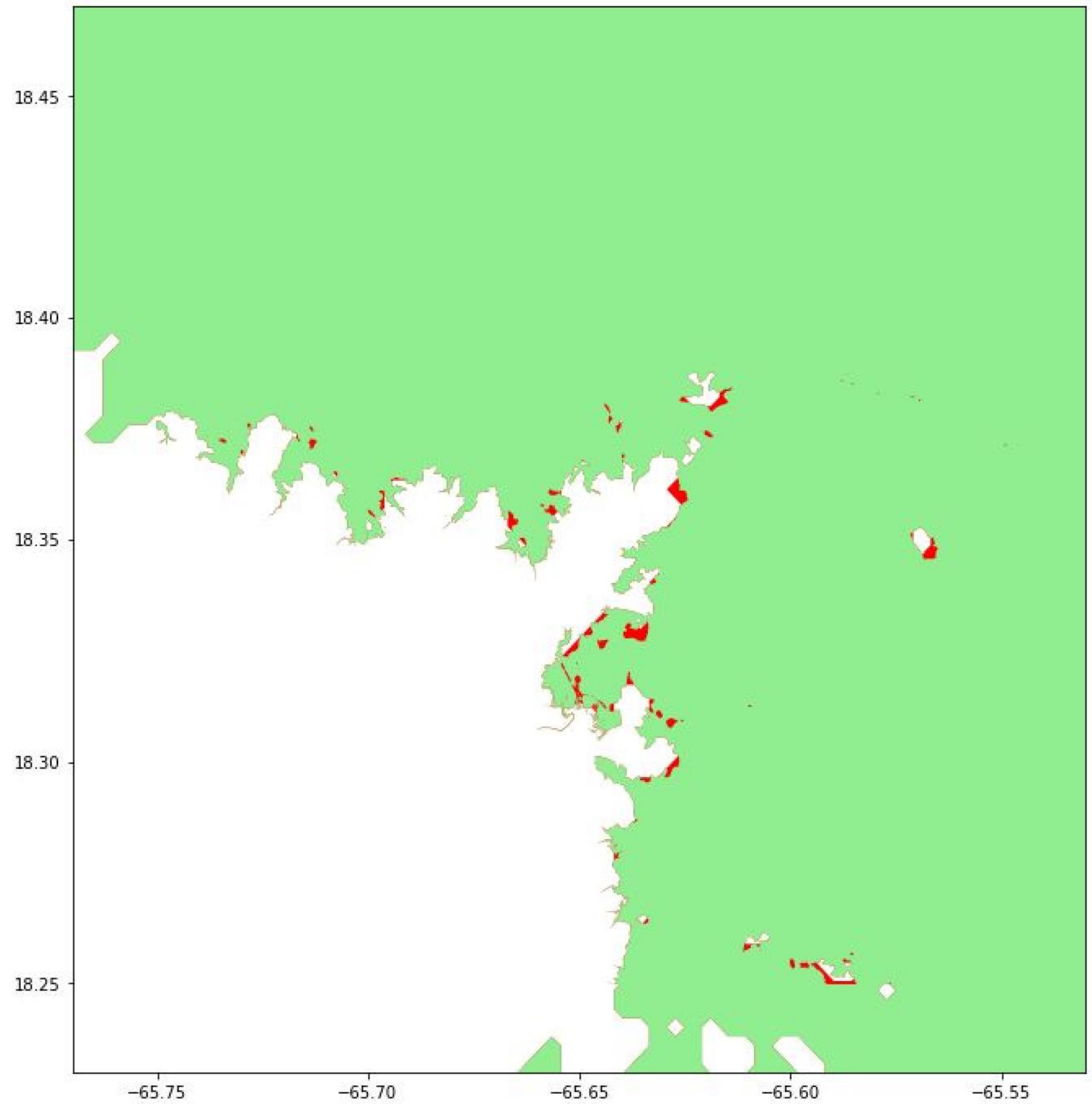

Figure 2 Comparison of union operation with (green) and without (difference in red) priority checking

It's also important to keep in mind that geometry creation priority is different from the interpolation priority.

There are also more advanced ways of specifying different contour specifications on different DEM (e.g., patches). The discussion of these methods is beyond the scope of this document.

Stateless operations

The get_multipolygon() method as well as msh_t() method (which in turn relies on get_multipolygon()) does not modify anything in the object or store any cache. In other words, geometry object is stateless and always recalculates required polygon or Jigsaw mesh object upon calling to the respective methods.

Factory constructor

All the different types of geometry objects are created using a factory pattern. The same factory class "Geom" returns geometries of different types based on its input arguments. The return objects in different cases are not instances of Geom, but instead of RasterGeom, MeshGeom, PolygonGeom, or MultiPolygonGeom, or GeomCollector.

Size Function

Size function object is used by OCSMesh to specify the element sizes in the domain. Specifying the sizes consists of basics such as global minimum and maximum and advanced specifications such as automated size calculation based on distance from feature lines or based on the bathymetry. The following are the different types of size specifications available:

- Global min/max
- Linearly increasing with the distance from auto extracted contours at specified level. Size starts from a given value (equal or greater than global min) to the global maximum. Expansion rate can be specified to control mesh transition.
- Automatic size calculation based on bathymetry gradient.
- Constant size in a region bounded by auto extracted contours of two different user specified levels (or bounded from one side by a single contour)
- Constant size in a user specified region (polygon input). This approach optionally accepts an expansion rate to linearly expand size values outside the specified region.
- Based on input mesh's element sizes (in case of mesh based hfun)
- Channel detection and refinement based on specified size and width

The sizing specification can combine different methods mentioned above, in which case the minimum of all different specifications is used as the final size. Just like geometry, size function can be defined based on different inputs (i.e., raster or mesh). The raster-type size function has the complete set of refinement features itemized above. Mesh-based size function has a subset of these capabilities. Collector type size function provides all the methods available to raster-based size function.

Mesh Size Specification

Element sizes need to be specified for each point in the domain. A naive approach to achieve this is to create a uniform rectangular grid over the whole domain and specify element size on each point of the grid. In OCSMesh this is called a "Hmat" (h as in size of the element at coordinate x: h(x) and mat for matrix).

This works fine for small domains. However, if the domain is large the machine might run out of memory. One solution is to reduce the resolution of the hmat grid (bigger squares). In certain cases this might be a reasonable thing to do, but still it's very restrictive and might also result in less than optimal final mesh due to either missing important features or having too fine a mesh in some areas.

The problem of accurately specifying element size is often overcome by using a cascaded mesh size grid specification where in refinement areas a hmat grid with finer resolution is cascaded on top of the coarse background hmat grid.

OCSMesh takes a different approach. DEM tiles are often small enough that multiple tiles can be loaded in memory; even if they don't fit, most readily available raster tools provide "windowing" over the tiles so that they easily fit.

OCSMesh specifies size on rasters for each raster tile and then meshes each single raster with the specified size to get an unstructured mesh. Then the values of the element-size is interpolated onto the unstructured mesh (OCSMesh calls this hfun form). This way the mesh size function is optimized: More values are specified in areas where the elements need to be smaller. The unstructured mesh size function form (hfun) is much smaller than the raster and more flexible than the hmat approach. In the last step, if multi DEM meshing is needed, the unstructured hfun from all the tiles are combined to form a final size function. Multi-tile meshing is explained in more detail in the Collector Type section.

In short, the element size is passed to the mesh engine as an unstructured mesh, defined in Jigsaw msh_t format in order to conserve memory.

Raster-based

Raster based size function applies user specified input sizes on the raster grid. Note that similar to geometry, when a size-function is created from a raster object, it doesn't copy the raster object, but just holds a reference to the existing raster. However, there's one big difference: the size-function additionally creates a separate copy of the raster grid to store the size specifications for the mesh. In other words, when a raster-based size function is created, internally it holds onto references to two separate raster objects, one is the original input raster (and is used for extracting geospatial data), the other is a copy of the original (in terms of raster grid coordinates) and has mesh size as its value instead of elevation data.

Example code below demonstrates how to create a raster based size function:

```
from ocsmesh import Raster, Hfun

raster = Raster("/path/to/raster.tif")
hfun = Hfun(raster, hmin=200, hmax=5000)

# Add refinements
hfun.add_contour(level=0, expansion_rate=0.01, target_size=200)
hfun.add_contour(level=-300, expansion_rate=0.01, target_size=500)
```

Code Snippet 15 Create a raster based size function and add contour two refinements

As you can see in Code Snippet 15, the main input arguments are global minimum and maximum of element size. If these global values are not provided, then subsequent "refinement" specifications are applied without bounds.

For example, assume the minimum and maximum element size are specified as 200m and 5000m, respectively. If the user specifies element size of 50m on coastline (contour at level=0m), then the size function uses 200m elements on coastline and expands with the specified rate to the maximum of 5000m. However, if the minimum and maximum are not specified, the size function uses 50m on coastline expanding to whatever maximum value (equal to linear distance from coastline * rate * 50m) in the domain. It is therefore advised to always specify global minimum and maximum to avoid unexpected results.

As described in the **Error! Reference source not found.** section, the size-function is specified on an unstructured mesh and then a number of them can be combined to cover a larger area. In the case of raster based size function, this local mesh is generated on the bounding box of the raster. It means that even regions that are not meshed may get mesh size specification. Because of this, it is a good idea to specify an upper elevation bound when possible, to avoid unnecessary size specification in area that are not in the domain.

Just like geometry, to get the size function in Jigsaw format the following method can be used:

```
hfun_msh_t = hfun.msh_t()
```

Code Snippet 16 Get Jigsaw mesh object from size function

OCSMesh Jigsaw driver automatically calls this method on hfun object when passing size information to the mesh engine.

Mesh-based

Mesh based size function specifies element size on the nodes of a mesh. Unlike raster based size functions this type of size function doesn't make a copy of the underlying mesh. Instead it directly specifies size values on the input mesh object, modifying it in the process.

There are two cases where using a mesh-based size function is very helpful:

1. When mesh size function output is generated, it can be stored on disk (in 2dm or Grd format) and used later for multiple meshing jobs. To reuse the exported size function mesh-based size-function can be utilized (Code Snippet 17).

```
from ocsmesh import Mesh, Hfun

mesh = Mesh.open("/path/to/saved_hfun.14")
hfun = Hfun(mesh)
```

Code Snippet 17 Create mesh based size function

2. When there's an existing mesh that needs to be locally updated (e.g., refined). The user can create a size function from that mesh and then calculate the mesh size at the nodes based on the length of connected edges. After that local refinement can be applied on the size function.

```
mesh = Mesh.open("/path/to/old_mesh.14")
hfun = Hfun(mesh)
hfun.size_from_mesh()
```

Code Snippet 18 Calculate the size function based on the elements of an input mesh

Note that during the creation of the mesh-based size function, no minimum and maximum size is provided; these extremums are inferred from the underlying mesh values. That means if one just opens a mesh with no value stored on it, or a mesh with elevation values and creates a size function from it, then the size function on its own is meaningless. To get the underlying mesh size, size_from_mesh() can be utilized to calculate sizes from the edges of the input mesh.

Collector Type

Most real world use cases would require multiple tiles of DEM for the domain and the size function. Just like geometry, combining multi-tile DEM size-functions (and potentially mesh size function) requires decent understanding of the inner workings of OCSMesh and the packages it relies on. In order to make the process easier, the collector size function has been implemented.

```
from ocsmesh import Hfun, Raster, Mesh

mesh = Mesh.open("/path/to/old_mesh.14")

raster1 = Raster("/path/to/raster1.tif")
raster2 = Raster("/path/to/raster2.tif")
raster3 = Raster("/path/to/raster3.tif")

hfun = Hfun([mesh, raster1, raster2, raster3], hmin=20, hmax=20000)
```

Code Snippet 19 Create a size function collector

Creation of combined size function is available through CLI as well. Still, some of the advanced features are only accessible through the Python API.

Combine Process

The size function combine process executes the following steps:

1. All raster based inputs are clipped to the bounding box of the base mesh or base polygon (if provided)
2. Contour refinements specification is processed
    - Contours are extracted from all the DEMs (or select DEMs if specified).
    - All the extracted contours are applied on all the raster and mesh based size functions.
3. Subtidal flow limiter refinement are calculated on the raster-based size functions only and applied on the same rasters.
4. Constant value refinements are applied on the specified (or all) raster based size functions
5. Patches are applied on all the specified mesh and raster based size function
    - Patch for size functions means specified custom regions with a constant size, expanding with the specified expansion rate outside that region
6. Channel detection refinements are applied on all the mesh and raster based size function inputs
7. For all the inputs (raster or mesh based size functions) the hfun form is retrieved.
    - This involves getting underlying mesh from the mesh based size function and creating a mesh on each raster and reprojecting the element size values on the mesh objects
    - The result of this step is a mesh with overlapping elements
8. Overlaps are resolved by removing triangles whose every node falls within the bounding box of the higher priority size functions (Figure 3).
    - If there's an input base mesh, it is used as the lowest priority input size function.

- Element clipping priority is based on the collector input arguments order. The last input has the highest priority, meaning none of its elements are clipped.

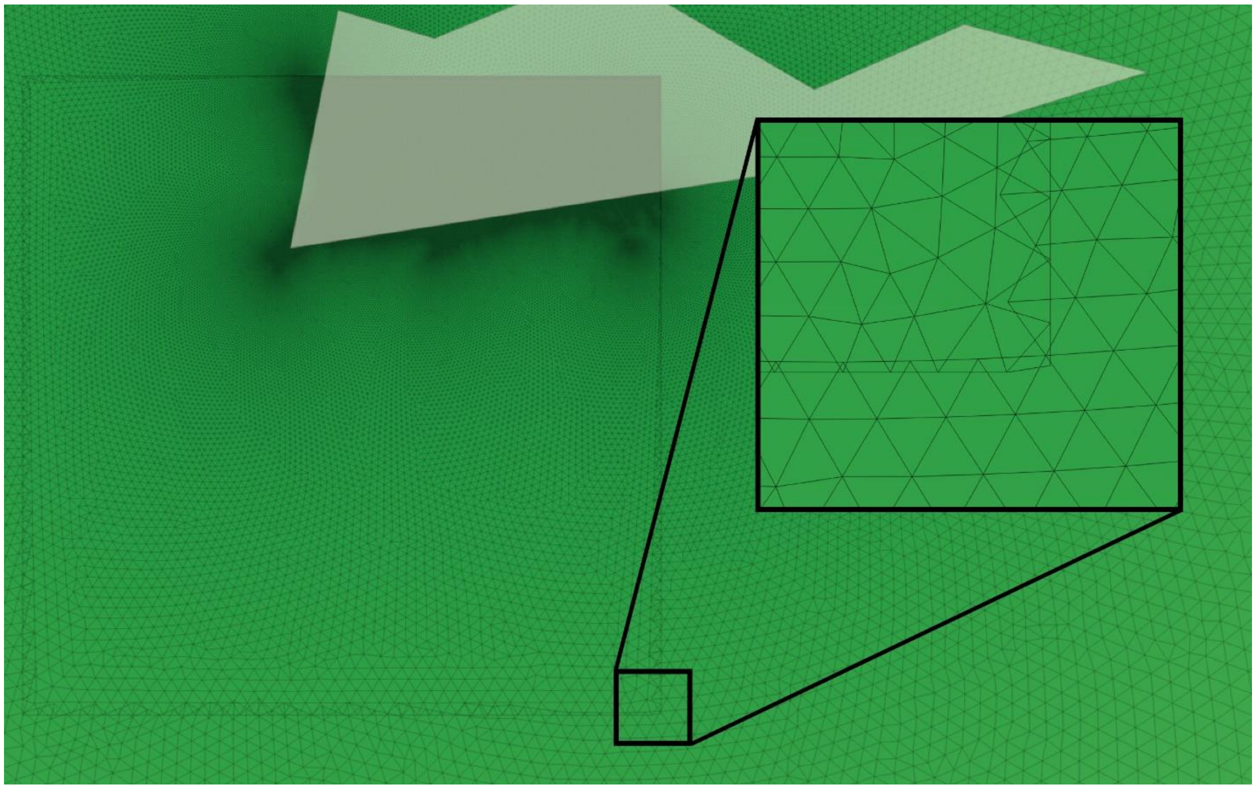

Figure 3 Combined size function mesh from two different overlapping DEM tiles covering US Virgin Islands

Applying Contours on All Hfuns

As specified in the combine process description, features (elevation contours or user specified lines) are applied on all the size functions within the collector. This is to ensure a smooth transition of mesh size across different DEM tiles.

Overlapping Triangles

After the process of triangle removal, the size function is still a non-conformal mesh along the edges of tiles. In fact, some overlapping triangles remain on the edges of the bounding box of the tiles. Jigsaw can handle an input size function like this and generate a conformal smooth mesh from it.

Exact vs Fast Methods

The described method to combine features across tiles, is accurate and retains the details. However due to its exponential time nature, it can be a computational bottleneck.

To enable fast prototyping of size functions in cases that the specified global minimum size is much larger than DEM resolution, another faster approach for combining the tiles is implemented. This approach sacrifices some details for much faster (and more memory hungry) computation. Because of this balance, the method is named the “fast” method, vs the more precise method which is named “exact”. The method that the collector uses is specified as an argument during construction of the object. By default the “exact” method is used.

The process for fast combine is slightly different from the one described above. In this method the following steps are taken:

1. A large raster is created that covers the bounding box of all raster inputs.
   a. The resolution of this “big raster” is set to one half of the specified global minimum mesh size (hmin/2)
   b. An estimate of available memory is calculated based on the user inputs to apply windowing over the rasters
      i. The user inputs that affect this window calculation are the number of DEMs, the resolution of highest resolution DEM, and the number of processes specified for the collector
2. If needed all the input rasters are projected onto this big raster
   a. Projection is required if some refinements such as constant size or topography based refinement are applied by the user
   b. Specifying contour refinement doesn’t require projection because the contours are still calculated on the original DEMs
3. All other steps related to “exact” collector size function are followed except that everything is applied on the big raster instead.
   a. The mesh-based size function inputs are still processed as before
4. In the final step, the big raster hfun is clipped by the shape of all input raster-type DEMs.
   a. Big raster hfun gets the highest priority irrespective of the input order of input DEMs or meshes.

```
from ocsmesh import Hfun, Raster, Mesh

mesh = Mesh.open("/path/to/old_mesh.14")

raster1 = Raster("/path/to/raster1.tif")
raster2 = Raster("/path/to/raster2.tif")
raster3 = Raster("/path/to/raster3.tif")

hfun = Hfun([mesh, raster1, raster2, raster3], hmin=20, hmax=20000, method='fast')
```

Code Snippet 20 Create collector size function with faster calculation algorithm

The process of creating and using the collector with a fast method doesn't differ from the "exact" method, except for the input argument.

Caution should be taken when using fast method, as the final sizes might not be the same as the ones calculated by the exact method.

Refinement Types

Most of the features in OCSMesh are implemented for raster-based types first and then extended to other types. The same is true for different types of refinement specifications for mesh size function. Collector size function supports all the refinement specifications supported by raster-based size functions and delegates them to the underlying raster inputs.

The refinements work accumulatively. In other words, if two different refinements are applied the resulting size function gets the minimum of the two at any point.

Sizes specified in refinements need to be bound by the global extremums. All refinement types discussed in this section can be used together.

Contour-based

Expanding from Specified Levels

Contour based refinement type is the most natural way of applying refinement in coastal modeling mesh generation. Contour refinement specifies elements to have a size equal to a "target size" on the contour line at the specified "level" and that the size be expanded with a given "expansion rate" with distance from that contour.

```
hfun.add_contour(level=0, expansion_rate=0.001, target_size=200)
```

Code Snippet 21 Add contour refinement to size function

The code above specifies that the mesh size on the contour line of level 0m must be equal to 200 meters with an expansion rate of 0.001. The expanded values for points are then calculated as:

```
value = expansion_rate * target_size * distances + target_size
```

Code Snippet 22 Calculations for element size during application of feature refinement with a given expansion rate

Where `distances` is the distance of a given point to the contour curve for specified level. At the end of the process the calculated mesh sizes are truncated by the global values.

The calculation of distance for all points is computationally expensive, especially when applying contours on many rasters.

Multiple contour refinements can be combined sequentially as well. (Code Snippet 23).

```
hfun.add_contour(level=0, expansion_rate=0.001, target_size=200)
hfun.add_contour(level=-1000, expansion_rate=0.002, target_size=2000)
hfun.add_contour(level=-4000, expansion_rate=0.005, target_size=6000)
```

Code Snippet 23 Specifying multiple contour refinement specification on the same size function

For the more advanced collector type additional arguments may be provided to specify where to extract the contour from.

Constant between Levels

Another way to specify a size function is to restrict the element size to be a constant value in a region bounded from one or two sides by contours of specified levels. This refinement type is much less computationally expensive than calculating point distance for contours.

```
hfun.add_constant_value(value=200, lower_bound=0, upper_bound=10)
```

Code Snippet 24 Add constant value refinement specifications to a size function

Code Snippet 24 specifies a mesh size of 200m from elevation 0 to 10 meters. Multiple levels of constant value can be specified to get a step-wise size function (Code Snippet 25).

```
hfun.add_constant_value(value=200, lower_bound=0, upper_bound=10)
hfun.add_constant_value(value=2000, lower_bound=-1000, upper_bound=0)
hfun.add_constant_value(value=6000, lower_bound=-4000, upper_bound=-1000)
```

Code Snippet 25 Specify multiple constant value size regions on size function to create a stepwise hfun

Constant value can also be combined with contour refinement to provide an expand-but-not-greater-than logic for the size function.

As mentioned above, the upper and lower bounds don't necessarily need to be specified; for example if one wishes to mesh everything above 0m depth (floodplain) with a constant size of 200m, the following code will do the trick:

```
hfun.add_constant_value(value=200, lower_bound=0)
```

Code Snippet 26 Add unbounded region constant value refinement to a size function object

Note that in case the upper bound of the domain is actually at level=10 then this would be equivalent to the earlier expression.

Custom Feature-based

Feature lines created or extracted from other sources of data can be incorporated into the size function as well. This provides the flexibility of including domain expert knowledge in the meshing process.

```
from shapely.geometry import LineString

feature = LineString([[0, 0], [1, 1]])
hfun.add_feature(feature, expansion_rate=0.001, target_size=200)
```

Code Snippet 27 Add custom feature refinement to a size function object

Where target size and expansion rate have the same role as in contour refinements. In fact contour refinement is nothing but calling add-feature on the contour lines extracted from the DEM.

A feature can be either a shapely linestring or multi-linestring. Note that the input for this method doesn't accept CRS information as argument, so the input feature must be in the same CRS as the size-function.

- For raster based size function `hfun.crs` returns the CRS of the size function. If no warping operation is done, then this would be the same as the input raster's CRS.
- For collector type size-function the CRS is assumed to always be EPSG 4326. Future versions of OCSMesh might support different CRS values for collector types.

Shape-based

A natural extension of feature-line refinement is to specify refinement based on a custom polygon or multipolygon. Shape based refinement accepts a shapely polygon or multipolygon object along with a target value and an optional expansion rate. It uses constant size for all points within the polygon and then if expansion is specified expands the mesh size for points outside the shape using the exterior contour of the polygon as distance reference.

The same CRS considerations as in custom feature-line based refinement applies for shape based refinement.

Topography-gradient based

As opposed to previous types of refinements where the distance to a given feature or a region defines what the mesh size is, topography based refinement only considers topography for mesh size calculation. Currently OCSMesh implements only a gradient based topographical refinement called subtidal flow limiter.

```
hfun.add_subtidal_flowlimiter(
    hmin=300, hmax=15000, lower_bound=-1000, upper_bound=0)
```

Code Snippet 28 Add subtidal flow limiter refinement to a size function

The code above specifies subtidal refinement with a minimum mesh size of 300m and maximum size of 15000 meters for the region between 0 m and 1000 m depths. Specifying bounds for the region is not required. In fact one can simply call this method as follows:

```
hfun.add_subtidal_flowlimiter()
```

Code Snippet 29 Add a unbounded subtidal flow limiter refinement to a size function

The value for subtidal flow limiter refinement is internally calculated as:

```
values = (1/3) * abs(topobathy/dz)
```

Code Snippet 30 Calculation for subtidal flow limiter refinement size

Where `z` is positive-up elevation and `dz` is the magnitude of elevation gradient. Essentially, what happens is that the local mesh size becomes a function of the ratio of elevation to the magnitude of elevation gradient at each point. In other words, in locations with small elevations as well as locations with high elevation gradients the element sizes are small. Using this refinement, one can capture channels that occur at the bottom of the domain topography.

Local Domain-width based (Channels)

OCSMesh can automatically detect and refine the channels and narrow passages within the domain. The logic is based on the geometry of the domain only. This refinement type is implemented for raster and collector size-functions.

```
hfun.add_channel(level=0, width=1500, target_size=200, expansion_rate=0.01)
```

Code Snippet 31 Add channel refinement specification for a size function

Code Snippet 31 instructs OCSMesh to

1. Extract the polygon/multipolygon bounded by elevation 0 m
2. Find passages of width 1500 meters or less, including sections of the polygon that have distance less than the specified width, such as high curvature corners. We call this the detected region.
3. Use the “detected region” as a patch refinement with a target mesh size of 200 m and expansion rate of 0.01.

Note that to calculate the channel based on the geometry, buffering operation is used followed by a filtering operation for shapes that have areas smaller than $width^2 \times (1 - \frac{\pi}{4})$ because of this some small features might be missed when finding channels with large widths.

Currently channel detection mechanism is still in development, can be computationally expensive, and needs to be used it with caution.

Lazy Calculation

All the calculations for the collector size function are done lazily. That means no actual calculation is done before the user calls the `msh_t()` method. Note that unlike the geometry object, the collector size function object has some level of statefulness, so that some of the extracted contours are stored in memory. For "exact" collector size function the calculations related to applying features are also stateful, but still the calculation of Jigsaw mesh objects is on the fly.

Enforcement of Max and Min

Currently size function's minimum and maximum values are enforced, only when some type of refinement is applied. If not, OCSMesh simply ignores the size function values and passes the global minimum and maximum directly to Jigsaw.

Factory Constructor

Similar to geometry objects, size-functions of different types are created using a factory class "Hfun" by providing different input arguments.

Mesh Post Processing

One of the very important steps after the mesh is generated is to check the quality of the mesh and throw out elements that can negatively impact the simulation results.

OCSMesh has a build-in mesh finalization step during which isolate nodes, pinched nodes and small patches of elements are removed from the mesh.

Isolate Nodes

Isolate nodes are the vertices that have no edge connected to them. These vertices are result of either mesh generator iterations not cleaning up vertices properly, invalid input mesh vertices which ended up in the output mesh or a result of other cleanup operations which delete elements, but not necessarily the vertices.

This type of nodes are detected and removed by gathering all the node indexes that are used by all the elements, removing isolates, and then renumbering the indices.

Pinch Nodes

Pinch nodes are vertices that are at the intersection of boundary edges such that they are simultaneously in contact with flow from two different sides. In other words these are the nodes that reside on more than two boundary edges.

In order to find these vertices OCSMesh finds nodes that have more than two boundary edges attached. Right now the cleanup approach in OCSMesh is to remove all the elements that contain the pinched node.

Sieve Small Patches

Another problem that might occur in the raw mesh is the existence of small patches of elements (such as islands) scattered in the domain. These patches could be a result of removing elements with pinch nodes thus disconnecting a small patch from the main mesh.

OCSMesh finds all polygons from mesh elements and then based on the polygon area either removes or keeps them during the finalization step. By default, OCSMesh only keeps the largest polygon of the mesh. However, the user can pass a threshold so that all the mesh regions larger than the threshold are retained.

Interpolation

After the mesh cleanup, the remaining steps to get the mesh model ready is to interpolate the DEM elevations on the mesh and then optionally detect boundaries. The mesh object created as a result of running Jigsaw via OCSMesh provides the necessary API to achieve this.

```
...
mesh = driver.run()
mesh.interpolate(list_of_rasters)
```

Code Snippet 32 Method to interpolate a list of DEMs on mesh

As you can see in Code Snippet 32, the `interpolate` method accepts a list of rasters. Note that these rasters do not need to be the same rasters used to create the geometry or size-function objects. The priority of rasters is increasing in the list; in other words in case of overlap, the last raster in the list wins over the rest.

The boundary generation method is discussed later in the

Tools section.

6. Remeshing

Currently for the cases where a mesh needs to be locally refined OCSMesh provides tools necessary to prepare mesh and size function for remeshing. However, it doesn't yet provide a full Python API support for this process. Two helper utility scripts are currently implemented for refinement. One uses the input DEMs to identify the refinement area and then uses the same DEMs to create a new size function for the refinement; the other accepts input shape for the refinement area.

Both re-meshing algorithms are tightly coupled with how Jigsaw supports remeshing:

- First, the region that is targeted for refinement is clipped out of the initial input mesh. Then all the nodes that are not clipped out are tagged as "fixed".
- After that a new full domain size function is created by laying the refinement region's high resolution size function on top of the size function calculated from the input mesh (using `size_from_mesh` method).
- In the last step the domain and size-function along with the clipped and tagged initial mesh are passed to Jigsaw for re-meshing.

Jigsaw honors the tagged vertices and only meshes the regions of the domain that doesn't have mesh (due to clipping).

One drawback to such an approach is that sometimes some of the elements and vertices in the fixed area of the mesh are moved slightly by Jigsaw.

In the current version of OCSMesh, the remeshing size function described above can be created in two different ways:

1. The first method involves creating a single collector size function by passing the remeshing zones DEMs as well as the initial mesh (as base mesh). This collector automatically calculates the size function from the base mesh and then creates raster size function from the input DEMs and lays them on top of the base mesh
2. Another way to create such a size-function is to first create two separate size functions, one from the DEMs and another from the input mesh, then pass them as the input list to a collector size function constructor.

If refinements are concentrated in a small area of the input mesh, the method for creating the size function from an old mesh and new DEM data is very fast and can be used on the fly for remeshing.

7. Command Line Interface

Some of the common functionalities of OCSMesh are also accessible through its CLI. Currently the CLI provides functionality for creation of geometry and size function from a list of input DEMs and optionally a base mesh (i.e., CLI interface to collector types). Other CLI entry points are still in development.

8. Tools

OCSMesh provides some utility functions and helper classes for common mesh and raster operations.

Raster

Raster is the base class used for handling GeoTIFF and NetCDF DEM files. It's also the base class for raster-based size function. Raster class wraps some of the commonly used RasterIO functionality. This include methods for operations such as:

- CRS handling
- Clipping
- Reprojecting and resampling
- Windowing
- Calculating multipolygon from the raster data
- Calculating contours from the raster data
- Narrow area detection

Since Raster is the base class for raster-based size function class and is referred to by geometry object, all the functionality above can be readily used on the geometry and size-function objects created from raster files.

Windowing

Sometimes the amount of memory available on a machine is limited, or the raster file is large and as a result raster operations don't fit on the memory. The Raster class wraps the chunking functionality of the RasterIO and automatically uses it during operations. When a raster object is created from file path, one can provide a "chunk_size" and "overlap" arguments to create the Raster object with windowing enabled (Code Snippet 33).

```
raster = Raster(raster_path, chunk_size=2000, overlap=50)
hfun = Hfun(raster, hmin=100, hmax=8000)
jig_hfun = hfun.msh_t()
```

Code Snippet 33 Enable raster windowed processing

The instructions in Code Snippet 33 result in creation of a raster object that processes the raster data in square windows of size 2000 pixels such that the windows have an overlap of 50 pixels. When the Jigsaw mesh (msh_t) object is retrieved from the size-function above, the calculations happen in chunks of given size and overlap.

Parsers

OCSMesh supports many of the common input data formats through its underlying packages. These data formats including reading ESRI Shapefiles (Environmental Systems Research Institute, Inc., 1998), other vector formats, NetCDF, GeoTIFF, and Jigsaw mesh.

Similarly OCSMesh supports common output formats. This includes writing Shapely geometries to disk as Shapefile, VTK or Feather files and writing mesh as VTK or Jigsaw msh_t.

In addition to these formats, there are two custom mesh parsers implemented in OCSMesh to support formats used by coastal modelers (GRD and SMS 2dm). These parsers are defined under ocsmesh/mesh/parsers.

Boundary Extraction

As noted in the Overview section, the last step in the meshing process is to extract boundary segments based on the interpolated elevations. These boundaries are then written to GRD file when exporting the mesh and can be used to set up SCHISM boundary conditions. The API that provides the boundary extraction is currently exposed through OCSMesh mesh objects. Code Snippet **34** demonstrates how this feature can be used.

```
...
mesh = driver.run()
mesh.interpolate(list_of_rasters)
mesh.boundaries.auto_generate(threshold=-10)
```

Code Snippet 34 **Automatically extracting and storing mesh boundaries**

9. Examples

Shinnecock Inlet

This example demonstrates some of the capabilities of OCSMesh. The code in this section is not intended or optimized for high-performance requirements. Instead it aims to demonstrate the workflow and capabilities in each step of the meshing process for a real-world use case.

Raster Inputs

To generate the domain and size function, five (5) NCEI 1/9th of arc-sec Continuously Updated DEMs (Cooperative Institute for Research in Environmental Sciences (CIRES) at the University of Colorado, Boulder, 2014) in Shinnecock inlet are used along with 1 GEBCO DEM (GEBCO Bathymetric Compilation Group 2020, 2020) that covers the area of interest (Figure 4).

In this example no automation is used to find or select the DEMs. The figures are generated using QGIS (QGIS Development Team, 2009).

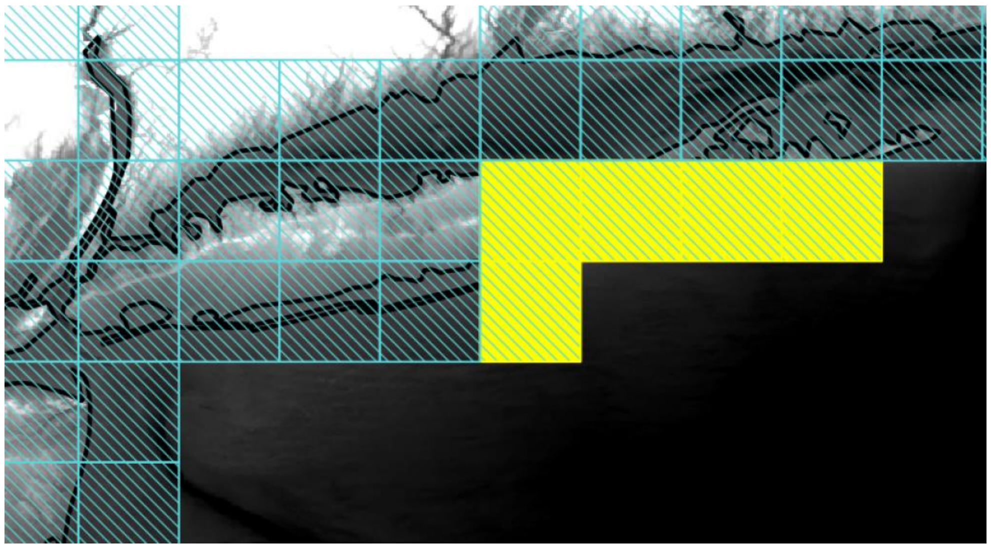

Figure 4 Raster data tiles (highlighted NCEI DEM + background GEBCO)

Specifications

The domain considered for this example is the intersection of a circular area with radius 30km around the inlet with the polygon bounded by 0m above sea level contour. The domain is extracted from the input DEMs.

Global mesh minimum size is set to 200m and maximum size to 8km. A refinement contour of 200 m size at level 0 m is specified.

Script

In the first section of the code (Code Snippet 35) modules used in this example are imported.

```
import pathlib

import numpy as np

import geopandas as gpd
from pyproj import CRS
from shapely.geometry import Point

from ocsmesh import Raster, Geom, Hfun, Mesh, JigsawDriver, utils
```

Code Snippet 35 Import necessary Python modules

`pathlib` and `numpy` (Harris, Millman, & van der Walt, 2020) are used for general file path and array handling respectively. `geopandas` (Jordahl, 2014), `pyproj` (Snow, 2021) and `shapely` (Gillies, Shapely: manipulation and analysis of geometric objects, 2007) are used to define the extent of the domain of meshing.

First, a list of DEM paths is created for later reference:

```
dem_dir = pathlib.Path('/path/to/dems/directory/')
dem_paths = [
    dem_dir / 'GEBCO/gebco_2020_n90.0_s0.0_w-90.0_e0.0.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n40x75_w073x00_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w073x00_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w072x75_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w072x50_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w072x25_2015v1.tif',
]
```

Code Snippet 36 Create a list of DEM paths

Note that the list order is such that NCEI rasters, being towards the end of the list, have higher priorities (in cases where priority is important such as during interpolation). The last element in the list has the highest and the first element the lowest priority.

A circular region is created around the inlet by buffering the point location of the inlet by 30 km in a local azimuthal equidistant projection around the inlet.

```
lon, lat = -72.476, 40.843
custom_crs = CRS.from_string(
    f'proj=aeqd +lat_0={lat} +lon_0={lon} +datum=WGS84 +units=m')

base_gs = gpd.GeoSeries(Point([0, 0]).buffer(30e3), crs=custom_crs)
```

Code Snippet 37 Defining the circular region around the inlet

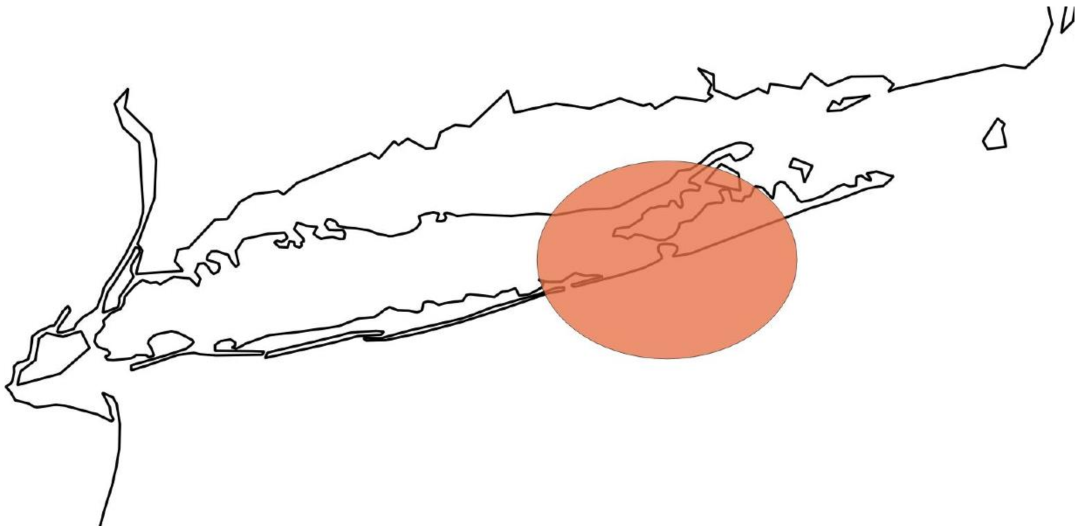

Figure 5 Circular region around the inlet location (EPSG 4326)

In Figure 5 the result is shown in EPSG 4326 and not in the azimuthal projection, hence the shape does not look completely circular. Note that the rendering of the shape is transparent. In order to export and visualize the base shape, one can use `to_file('desired/path')` method on the `base_gs` GeoSeries object.

After creating the circular region, the geometry object is created using the input rasters and the circular base shape.

```
geom_rasters = list()
for f in dem_paths:
    geom_rasters.append(Raster(f))
geom = Geom(
        geom_rasters,
        base_shape=base_gs.unary_union,
        base_shape_crs=base_gs.crs,
        zmax=0)
```

Code Snippet 38 Creating the geometry object defining the domain

Note that the coordinates system needs to be passed along with the base shape, otherwise EPSG 4326 is assumed, which in this case would be incorrect. In Code Snippet 38 maximum elevation of 0 m is specified, which means that extraction algorithm should consider the domain to be the region (within base shape) bounded from above by 0m elevation contour.

Since collector objects are evaluated lazily, nothing is calculated at this point. The polygon extraction happens when get_multipolygon is called. OCSMesh automatically calls this method to retrieve the domain information.

As a reminder, due to stateless lazy evaluation, if the user calls get_multipolygon and then passes the same geom object to the mesh driver (which calls get_multipolygon internally again) it will result in calculating the domain twice. There are ways to avoid double calculation, however the discussion of those methods is outside the scope of this document.

The return value of get_multipolygon is a Shapely multi-polygon which can be saved to disk.

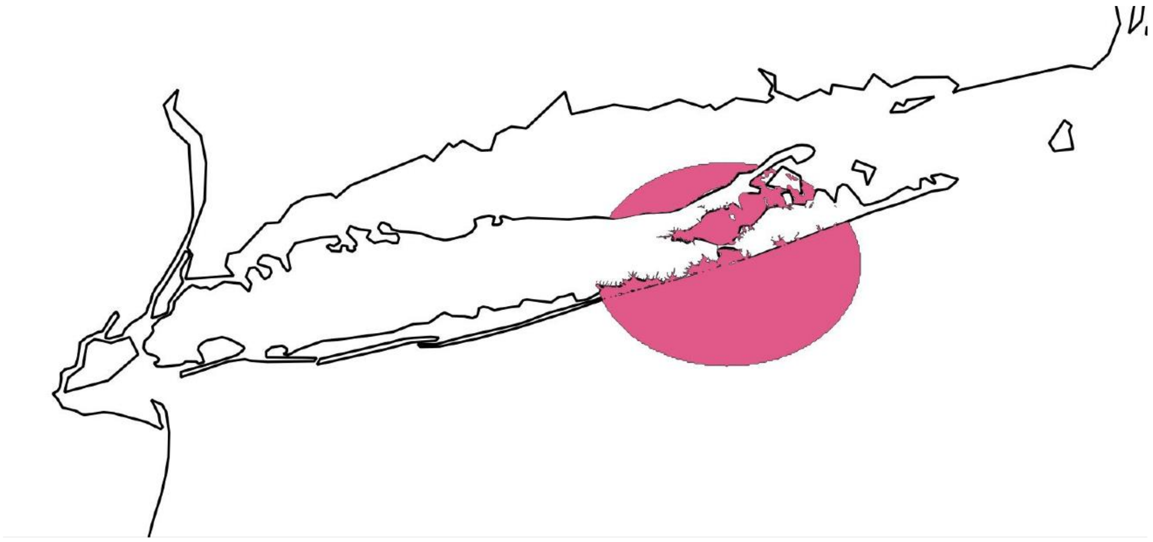

Figure 6 Multi-polygon calculated from the rasters and specifications

Note that the calculated multi-polygon in Figure 6 has multiple disconnected sections, some of which should not be meshed. One way to remove the excess region is to come up with a better base shape before creating the geometry object. Another method is to clean up the calculated geometry and then create a polygon-based geometry from the cleaned up polygon. For the purpose of this example, because the main section of the extracted domain is the largest segment and that OCSMesh by default discards all the smaller connected mesh segments (see Sieve Small Patches), the calculated geometry is not cleaned up before proceeding to the next steps.

The next step is defining the size function.

```
hfun_rasters = list()
for f in dem_paths:
    hfun_rasters.append(Raster(f))
hfun = Hfun(
   hfun_rasters,
   base_shape=base_gs.unary_union,
   base_shape_crs=base_gs.crs,
   hmin=200, hmax=8000,
   method='fast')

hfun.add_contour(level=0, expansion_rate=0.004, target_size=200)
```

Code Snippet 39 Define size function based on the specifications

Note that to define size function new raster objects are used to avoid the implicit mutation issue mentioned in

Quick Start.

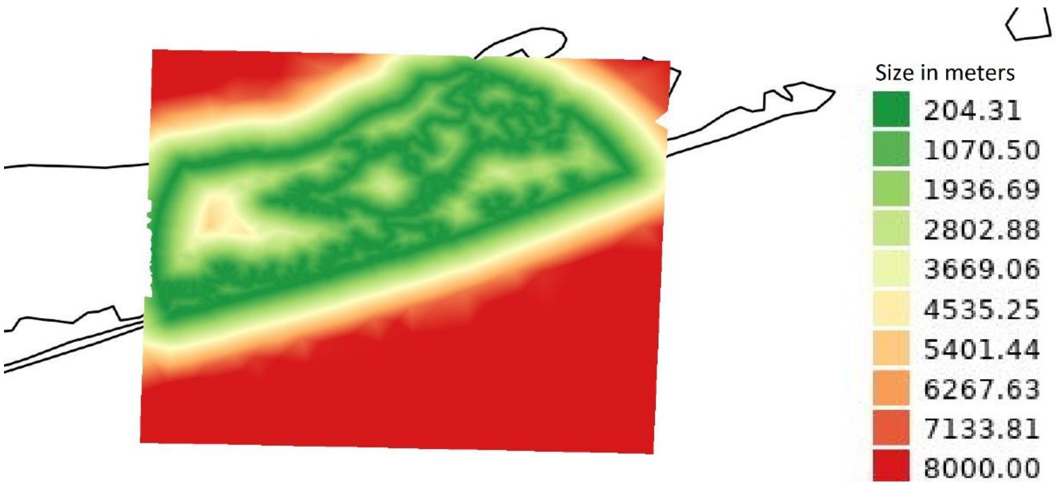

Figure 7 Calculated size function

Similar to geometry polygon, size function is not calculated until the `msh_t()` method gets called. This method returns an unstructured mesh with element sizes as its values.

Notice that the size function is calculated on the bounding box of the input base shape, rather than on the shape itself. To create the visualization in Figure 7, one can create an OCSMesh mesh object from the calculated Jigsaw mesh object and write it in 2dm format which is readable by QGIS.

```
hfun_jig = hfun.msh_t()
Mesh(hfun_jig).write('/desired/path/hfun.2dm', format='2dm', overwrite=True)
```

Code Snippet 40 Saving calculated size function to disk

OCSMesh’s Jigsaw driver uses geometry and size function objects to retrieve the domain and element size and passes them to Jigsaw. This is done by executing Code Snippet 41.

```
driver = JigsawDriver(geom=geom, hfun=hfun)
mesh = driver.run()
```

Code Snippet 41 Create Jigsaw driver and mesh the domain

The output mesh from Code Snippet 41 is a conformal cleaned-up mesh with no elevation data. Visualizing it results in Figure 8:

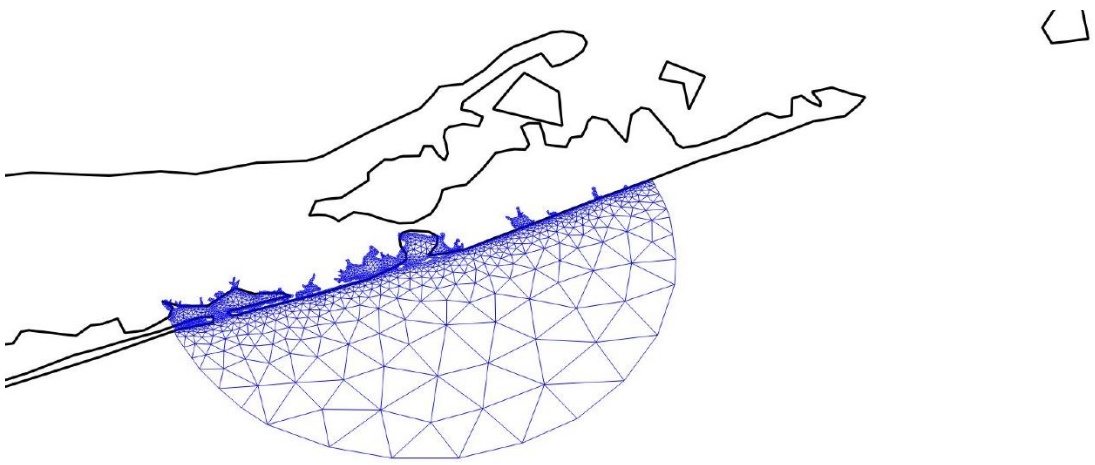

Figure 8 Generated and cleaned-up mesh with no elevation data

The value from the input DEMs can be interpolated on the nodes of the generated mesh using Code Snippet 42:

```
interp_rasters = list()
for f in dem_paths:
    interp_rasters.append(Raster(f))
mesh.interpolate(interp_rasters)
```

Code Snippet 42 Interpolating DEMs on the finalized mesh

Figure 9 depicts the result of the interpolation. The thick black lines show estimated coastlines from QGIS and the color bar is such that nodes above the coastline fall in the brown color zone. Note that although the domain cutoff is at 0m, after meshing some nodes might fall on land and interpolating DEM on those nodes results in elevations higher than 0m.

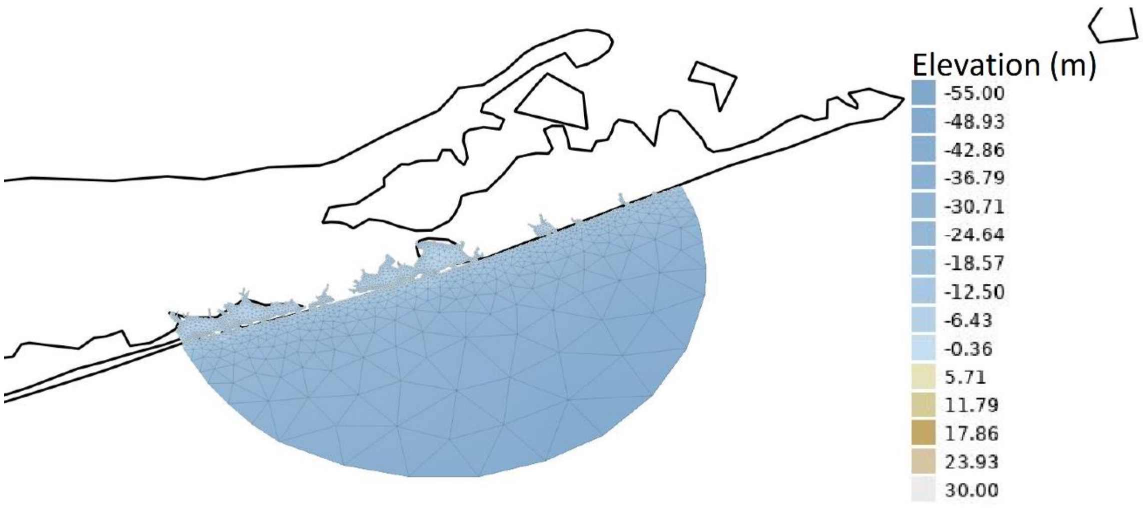

Figure 9 Mesh with interpolated DEM data

The final step for the mesh to be model ready is to detect the boundary segments. The boundary segment detection is automated based on the elevation input: edge segments with both attached nodes below a given threshold are assumed to be ocean boundary and the rest of the external boundaries are taken as land. Islands (internal boundaries) are also detected and assigned a separate boundary type.

```
mesh.boundaries.auto_generate(-10)
```

Code Snippet 43 Auto extraction of boundary segments

Code Snippet 43 shows how to generate boundaries for an OCSMesh mesh object. The mesh object can be either newly generated or imported from a file.

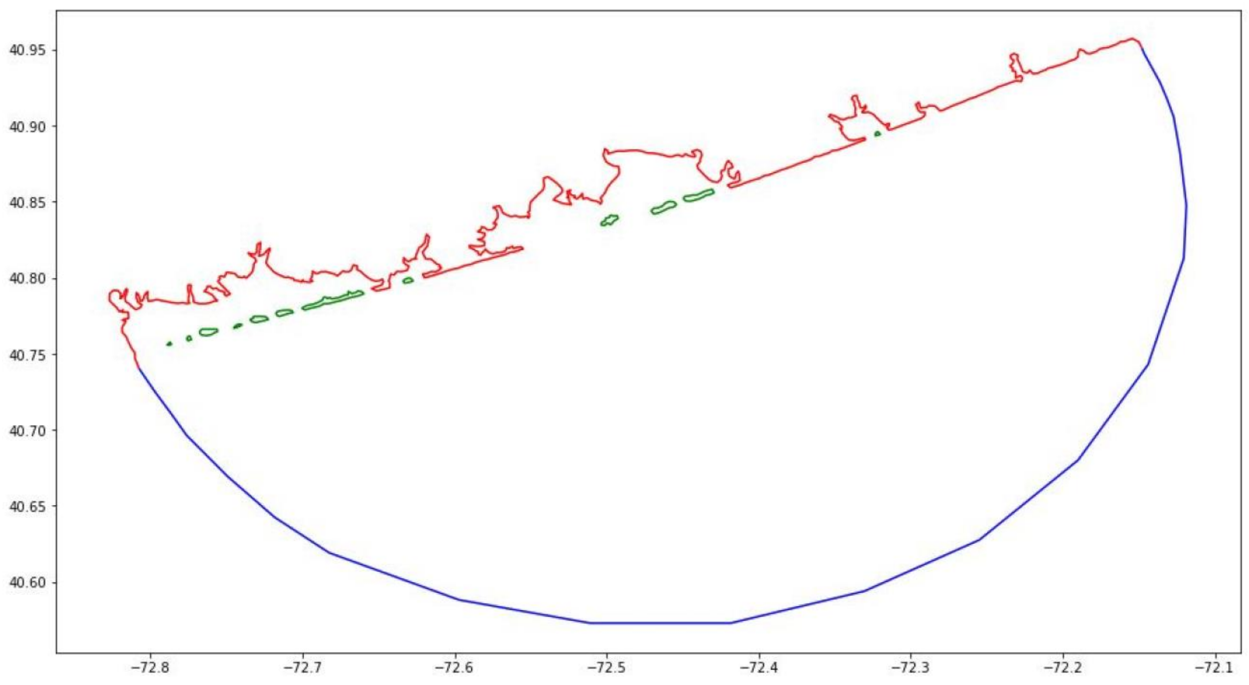

Figure 10 Automatically extracted boundary segments (ocean: blue, land: red, islands: green)

Note that in Figure 10 some of the nodes that are in fact in water are counted as land boundary. This is due to the fact that the calculation is elevation based. Since interpolation and approximation is involved during the mesh generation process, the elevation of some of the nodes that are actually on land might fall below 0 m or vice versa. Because of this, choosing 0 m as the threshold for boundary detection results in nodes to be assigned as ocean when they're in fact on land. This could result in disaster for the simulation. It would be better to choose a lower elevation as the threshold, especially if the mesh is coarse in some of the coastal boundaries (e.g., sometimes the threshold needs to be as large as -200 m for very coarse mesh on large domains).

To preserve the boundary type accuracy it is possible to programmatically force some nodes to be on land or in the ocean. However, this is outside the scope of this document.

Local Refinement

In this section another use-case for OCSMesh is demonstrated. Some of the details touched upon in the Shinnecock Inlet example are going to be left out in this example and instead the focus is on setting up local refinement of a mesh.

Setup

Let’s assume there’s an existing coarse mesh and we want to refine it in the New York City (NYC) area in order to get better results. The coarse mesh is created in the first part of this example, then in the second part the coarse mesh is refined locally.

On a side note, it is worth mentioning that in certain cases it would be beneficial (time-wise) to generate size functions in iterative steps and at each step add more locally concentrated refinements. This iterative approach for size function could be time saving for a large domain with concentrated local refinements.

Domain, data and specifications

The domain of meshing is the northern Atlantic Ocean basin on the east coast of the U.S. The process of generating the coarse mesh starts with a rough sketch of the domain (e.g., hand-drawn in QGIS) as shown in Figure 11. This rough domain sketch serves the same purpose as the circular region created in Shinnecock Inlet example.

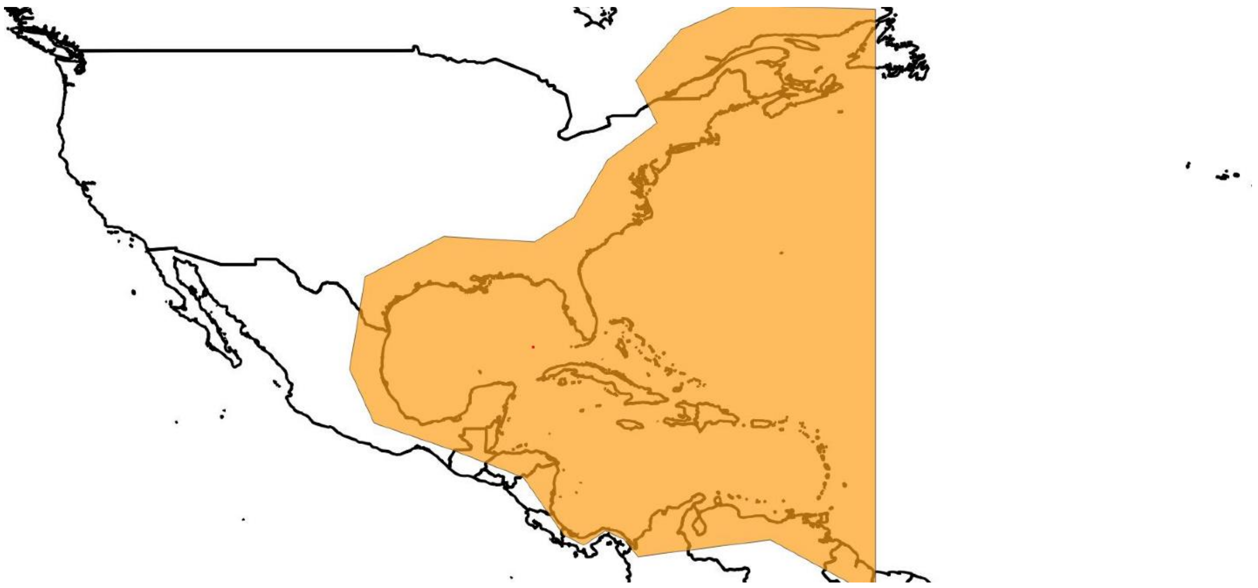

Figure 11 Rough sketch of the domain of meshing

The domain extracted from this basin is bounded by the contour of 10m above sea level. The DEM data used in this case is from the 2 GEBCO rasters intersecting with the basin shape and 4 NYC Manhattan area NCEI 1/9th of arc-sec Continuously Updated DEMs shown in Figure 12.

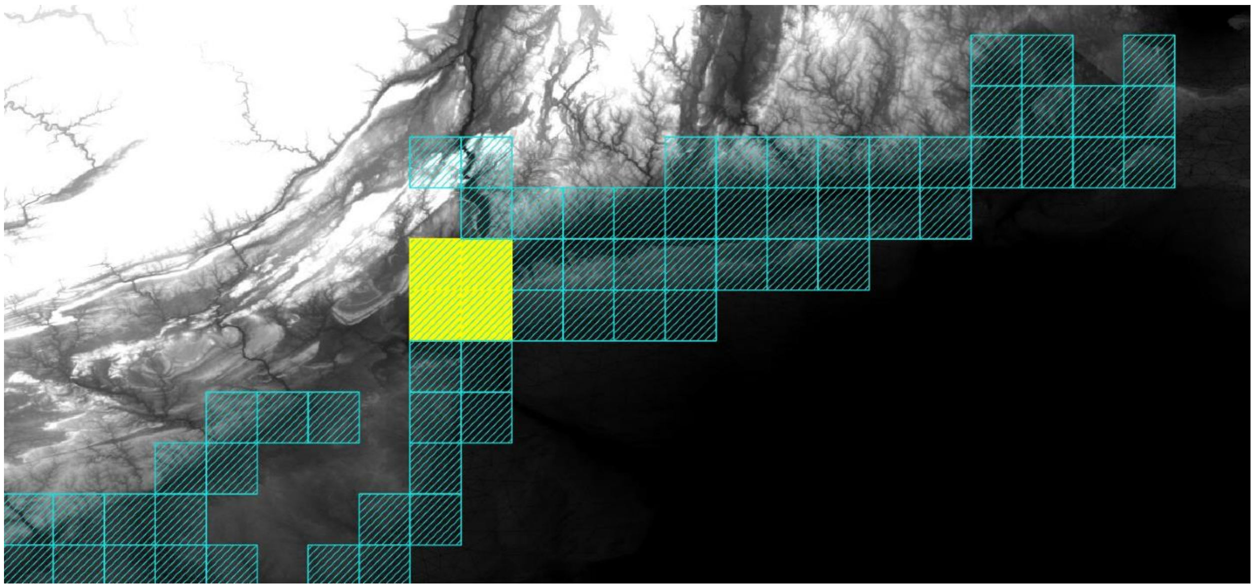

Figure 12 NCEI raster elevation data used (highlighted) for refinement region

Coarse mesh

Creating the coarse mesh is not a part of the mesh refinement. In an actual local mesh refinement case this step is not required. It is included so that one can easily follow the example without any prerequisites.

To define the domain, we start from the rough domain sketch in Figure 11. Then create a geometry object from it with maximum elevation equal to 10 m (Code Snippet 44, Code Snippet 45, and Figure 13).

```
import pathlib
import os
from copy import deepcopy

import numpy as np
import geopandas as gpd
from shapely import geometry
import jigsawpy

from ocsmesh import Raster, Geom, Hfun, Mesh, JigsawDriver, utils

dem_dir = pathlib.Path('path/to/dems/directory/')
gebco_paths = [
    dem_dir / 'GEBCO/gebco_2020_n90.0_s0.0_w-90.0_e0.0.tif',
    dem_dir / 'GEBCO/gebco_2020_n90.0_s0.0_w-180.0_e-90.0.tif',
]
ncei_paths = [
    dem_dir / 'NCEI_1_9th/ncei19_n40x75_w074x00_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w074x00_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n40x75_w074x25_2015v1.tif',
    dem_dir / 'NCEI_1_9th/ncei19_n41x00_w074x25_2015v1.tif'
]
dem_paths = [*gebco_paths, *ncei_paths]

domain_path = 'path/to/rough/sketch/atlantic_domain.shp'
```

Code Snippet 44 Importing modules and setting paths

Note that NCEI and GEBCO raster file paths are separated in order to reference the high resolution rasters separately in the Fine local remesh section.

```
base_gdf = gpd.read_file(domain_path)

rast_list_1 = list()
for f in gebco_paths:
    rast_list_1.append(Raster(f))

base_geom = Geom(
    rast_list_1,
    base_shape=base_gdf.unary_union,
    base_shape_crs=base_gdf.crs,
    zmax=10)
```

Code Snippet 45 Creating geometry for the domain

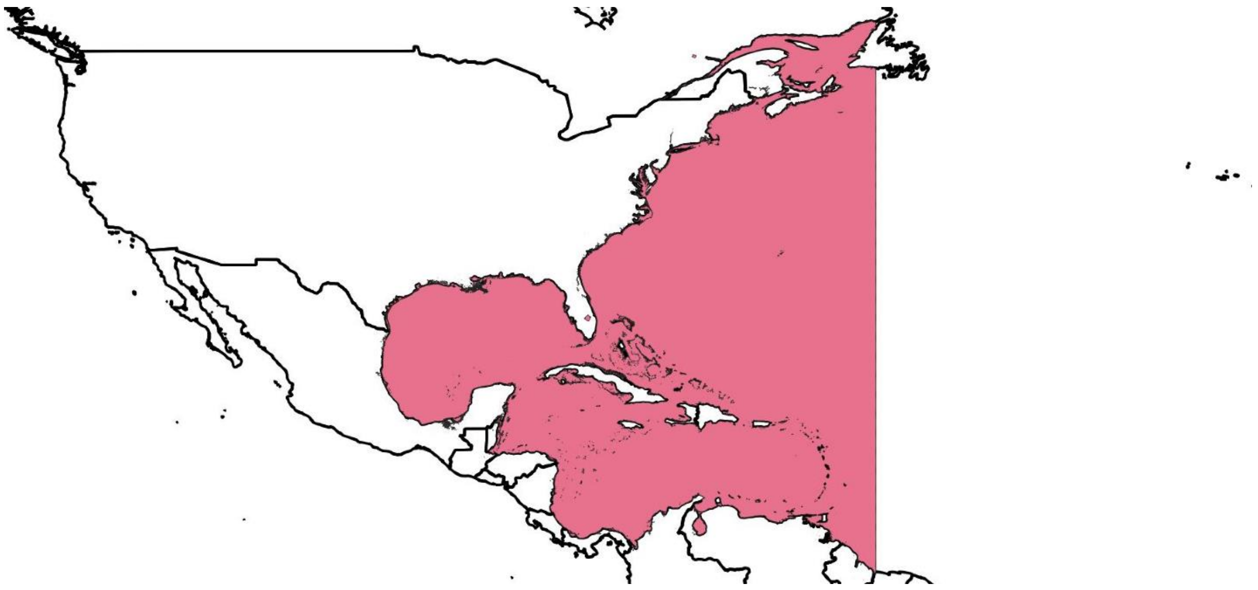

Figure 13 Geometry for base mesh at bounded at 10m contour level

Note in order to avoid the implicit mutation issue discussed in

Quick Start, in this example every time a list of rasters is passed to create an object or execute some operation, a new list is created. This is not necessarily needed in all the cases, but it is kept this way to make it easier to follow the example.

The next step is to create the size function (Code Snippet 46). The minimum mesh size used for the coarse mesh is 1000 m.

```
rast_list_2 = list()
for f in gebco_paths:
    rast_list_2.append(Raster(f))
base_hfun = Hfun(
    rast_list_2,
    base_shape=base_gdf.unary_union,
    base_shape_crs=base_gdf.crs,
    hmin=1000, hmax=15000,
    method='fast')

base_hfun.add_contour(level=0, expansion_rate=0.0005, target_size=1000)
base_hfun.add_constant_value(value=1000, lower_bound=0, upper_bound=10)
```

Code Snippet 46 Creating mesh size function for the base mesh

Note that not only a contour refinement of 1000 m at level equal to 0m is used, but also the mesh size is kept constant at 1000 m for all the points above 0 m by using add_constant_value. The final step for creating the base mesh is the meshing and interpolation (Code Snippet 47).

```
driver = JigsawDriver(geom=base_geom, hfun=base_hfun)
base_mesh = driver.run()

utils.reproject(base_mesh.msh_t, "EPSG:4326")

rast_list_3 = list()
for f in gebco_paths:
    rast_list_3.append(Raster(f))
base_mesh.interpolate(rast_list_3)

base_mesh.write('/desired/path/basemesh.2dm', format='2dm', overwrite=True)
```

Code Snippet 47 Generate the base mesh and interpolate raster elevation data

Fine local remesh

This section describes the steps for the local remeshing.

The process of creating the geometry and size function is similar to the previous example. One important difference is that instead of passing in a base shape, a base mesh is passed to the object constructors (Code Snippet 48).

```
base_mesh = Mesh.open('/path/to/basemesh.2dm', crs="EPSG:4326")

rast_list_4 = list()
for f in ncei_paths:
    rast_list_4.append(Raster(f))
geom = Geom(
    rast_list_4,
    base_mesh=deepcopy(base_mesh),
    zmax=10)

rast_list_5 = list()
for f in ncei_paths:
    rast_list_5.append(Raster(f))
hfun_fine = Hfun(
    rast_list_5,
    base_mesh=deepcopy(base_mesh),
    hmin=200, hmax=15000,
    method='fast')

hfun_fine.add_contour(level=0, expansion_rate=0.001, target_size=200)
hfun_fine.add_constant_value(value=200, lower_bound=0, upper_bound=10)
```

Code Snippet 48 Creating remeshing geometry and size function based on base mesh

Note that only NCEI DEMs are passed as input rasters to both geometry and size function constructors. The refined mesh size has a minimum of 200 m. The maximum size for the refinement size function should be equal to the maximum size of the base mesh. This is necessary in order to keep the large elements outside of the refinement region when remeshing using Jigsaw, otherwise Jigsaw ignores the fixed element tags (Code Snippet 50) and generates mesh for the whole domain. The maximum value can be retrieved automatically as in Code Snippet 49.

```
hfun_base = Hfun(base_mesh)
hfun_base.size_from_mesh()

rast_list_5 = list()
for f in ncei_paths:
    rast_list_5.append(Raster(f))
hfun_fine = Hfun(
    [hfun_base, *rast_list_5],
    base_mesh=deepcopy(base_mesh),
    hmin=200, hmax=hfun_base.hmax,
    method='fast')
```

Code Snippet 49 Retrieve the base mesh maximum element size automatically

The refinement geometry and size function in the vicinity of NYC are shown in Figure 14 and Figure 15.

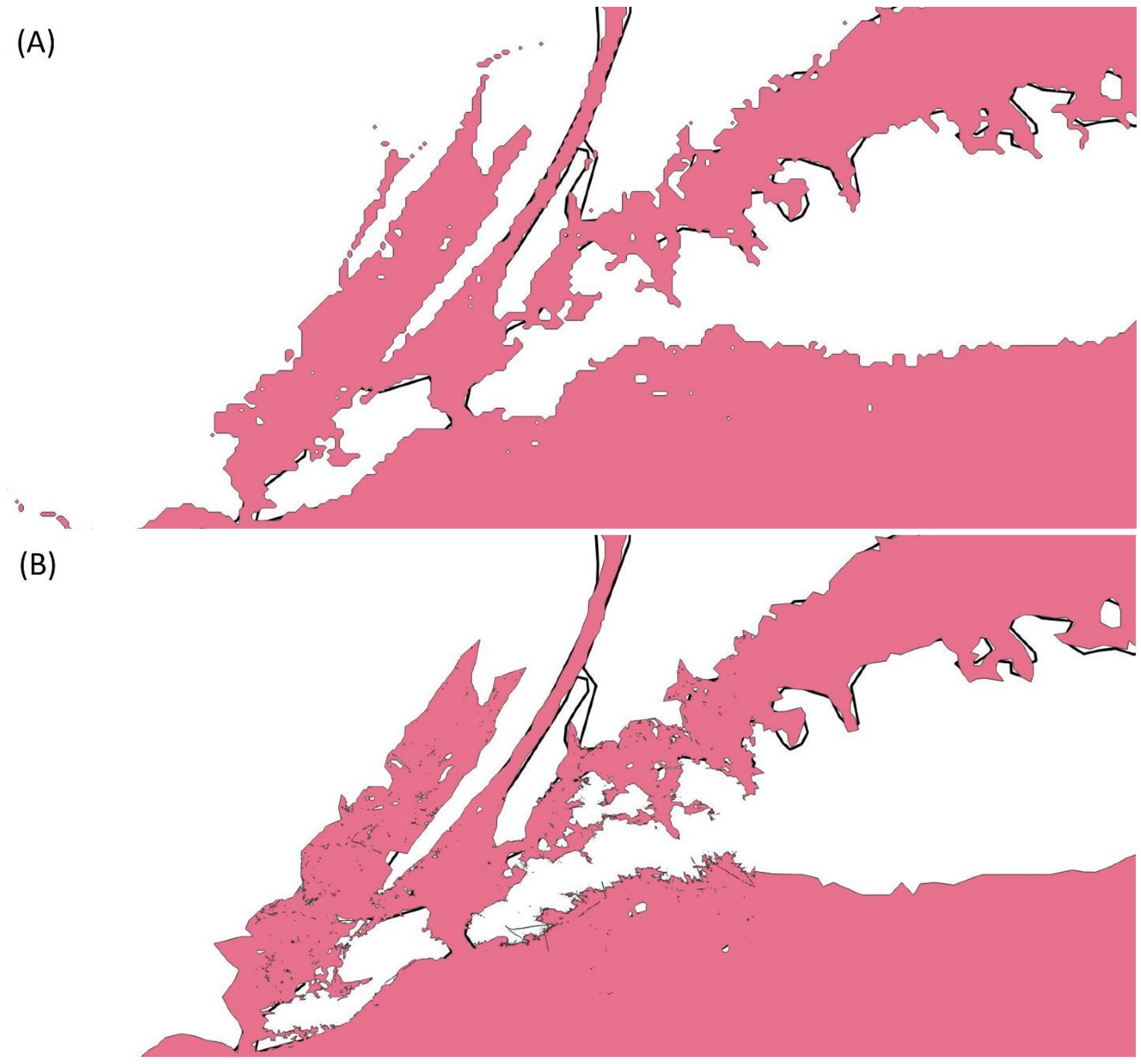

Figure 14 Extracted geometry for (A) base mesh and (B) refinement zoomed in NYC area

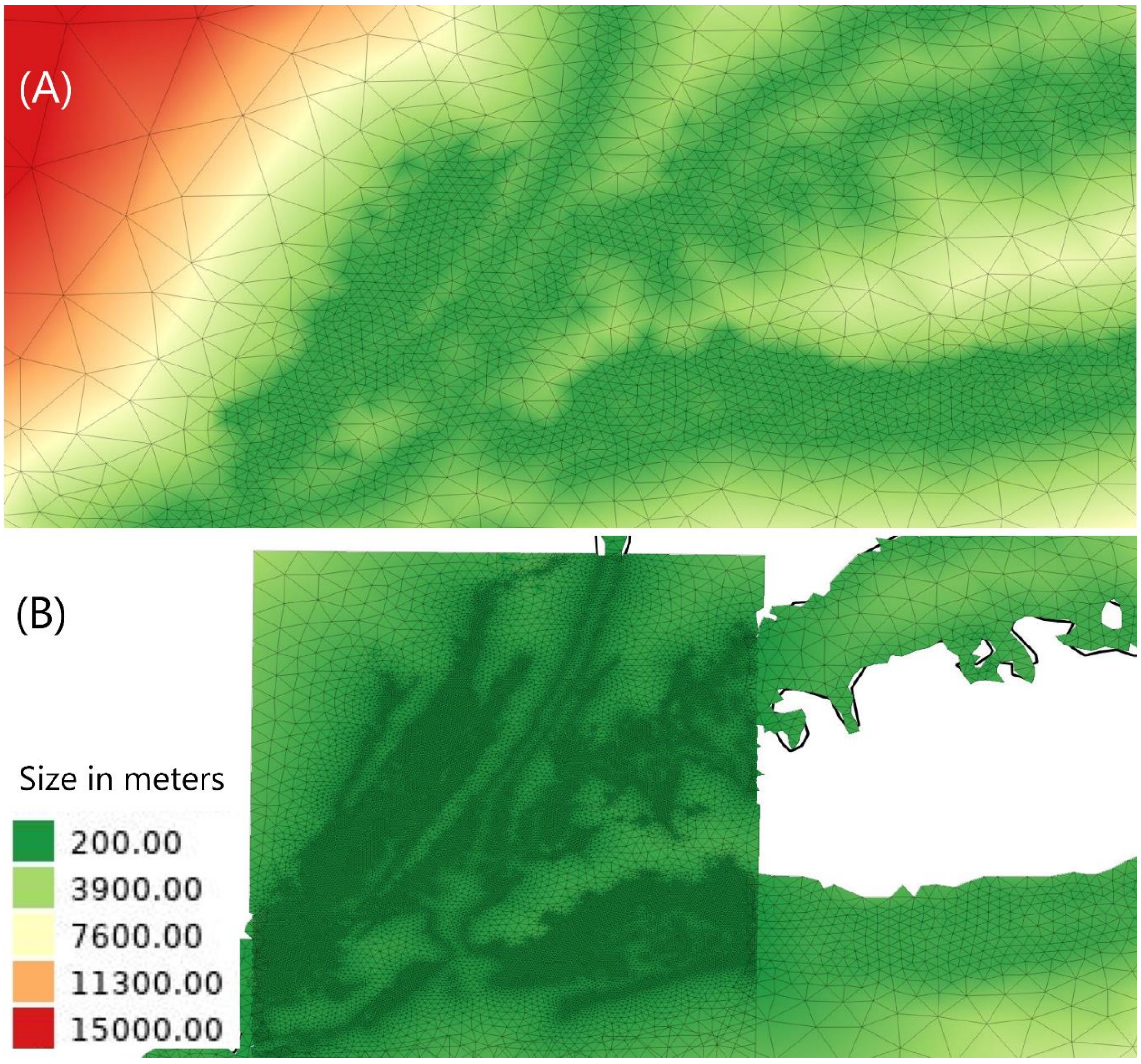

Figure 15 Extracted size function for (A) base mesh and (B) refinement zoomed in NYC area

After creating the geometry and size function, it’s time to locally refine the mesh. Although OCSMesh does provide necessary tools to prepare geometry, size function and initial mesh for remeshing. OCSMesh Jigsaw driver doesn’t support the local remeshing feature.

One way to circumvent this is to just pass in the size function and geometry to the driver and we should get a mesh which is similar to the input mesh, but is a complete remesh of the whole domain.

The better approach, which is demonstrated in this example, is to use all the available tools in OCSMesh, prepare the objects and then pass them to JigsawPy directly. To prepare the initial mesh for remeshing, first the remeshing regions are clipped out and then the remaining mesh edges and vertices are tagged as fixed. For simplicity, the refinement region is taken to be the union of bounding boxes of the input NCEI rasters.

```
geom_jig = geom.msh_t()
base_jig = deepcopy(base_mesh).msh_t

if geom_jig.crs != hfun_fine_jig.crs:
    utils.reproject(hfun_fine_jig, geom_jig.crs)
if geom_jig.crs != base_jig.crs:
    utils.reproject(base_jig, geom_jig.crs)

remeshed_jig = jigsawpy.jigsaw_msh_t()
remeshed_jig.ndims = +2
remeshed_jig.crs = geom_jig.crs

rast_list_6 = list()
for f in ncei_paths:
    rast_list_6.append(Raster(f))

boxes = [i.get_bbox(crs=geom_jig.crs) for i in rast_list_6]
region_of_interest = geometry.MultiPolygon(boxes)

fixed_mesh_w_hole = utils.clip_mesh_by_shape(
    base_jig, region_of_interest, fit_inside=True, inverse=True)

fixed_mesh_w_hole.vert2['IDtag'][:] = -1
fixed_mesh_w_hole.edge2['IDtag'][:] = -1
```

Code Snippet 50 Preparing initial mesh for remeshing to pass to JigsawPy

When a Jigsaw mesh object is created from the geometry, the CRS of the mesh type doesn't necessarily match that of the geometry itself. If the geometry object is in EPSG 4326, the object returned by `msh_t()` method will be in the local UTM projection estimated based on the center point of geometry extent. Because of this, the input mesh and size function need to be projected to the CRS of the geometry to make sure coordinates match.

Most of these details are masked from the user when using OCSMesh API. However since OCSMesh driver currently doesn't have support for remeshing, CRS needs to be handled in the client code for remeshing.

The next step is to pass all the inputs to JigsawPy and generate the mesh (Code Snippet 51).

```
refine_opts = jigsawpy.jigsaw_jig_t()
refine_opts.hfun_scal = "absolute"
refine_opts.hfun_hmin = np.min(hfun_fine_jig.value)
refine_opts.hfun_hmax = np.max(hfun_fine_jig.value)
refine_opts.mesh_dims = +2

jigsawpy.lib.jigsaw(
        refine_opts,
        geom_jig,
        remeshed_jig,
        init=fixed_mesh_w_hole,
        hfun=hfun_fine_jig)

utils.finalize_mesh(remeshed_jig)
```

Code Snippet 51 Pass prepared inputs to the JigsawPy for remeshing operation and cleanup raw mesh

After the raw mesh is generated, it needs to be manually cleaned up. For normal meshing, OCSMesh driver automatically calls the cleanup function, but since driver is not used for this example, the `finalize_mesh()` function needs to be called manually (Code Snippet 51).

When the mesh is cleaned up, we can interpolate raster and optionally old mesh elevation values on the new vertices to get the final mesh.

```
final_mesh = Mesh(remeshed_jig)

utils.interpolate_euclidean_mesh_to_euclidean_mesh(
        base_mesh, final_mesh.msh_t)

rast_list_7 = list()
for f in dem_paths:
    rast_list_7.append(Raster(f))
final_mesh.interpolate(rast_list_7)

final_mesh.write('/desired/path/final_mesh.2dm', format='2dm', overwrite=True)
```

Code Snippet 52 Interpolating old mesh values as well as new DEM raster elevation data on the locally refined mesh

The mesh can be compared to the base mesh to see the result of refinement (Figure 16). Base (coarse) mesh is in blue and refined mesh is in red. Panel A shows the overview of NY; elements of both meshes match in most of the regions except in areas closer to NYC. In panels B and C the difference in NYC mesh becomes more evident.

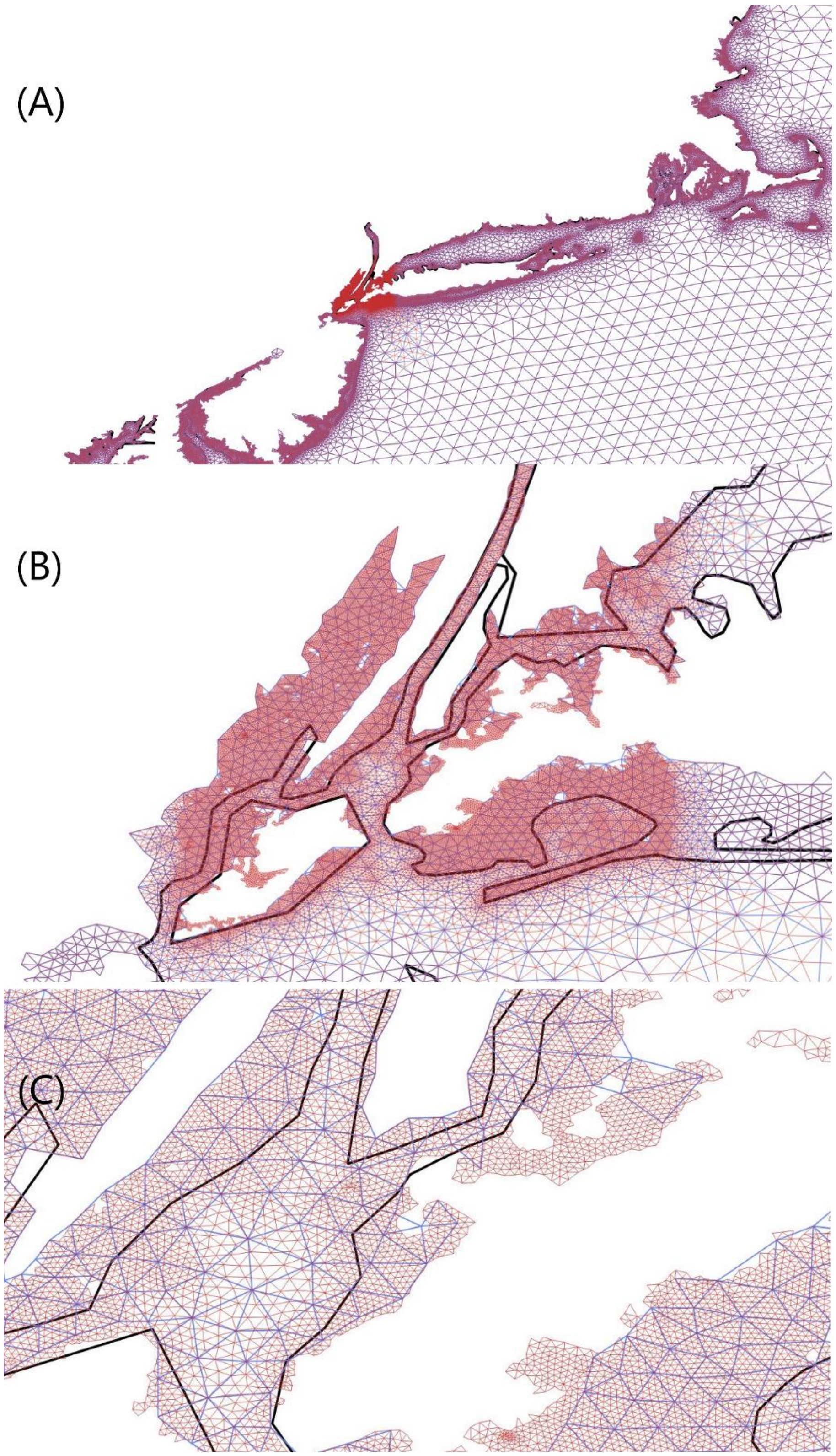

Figure 16 Overlay of base (blue) and refined (red) mesh in NYC area in 3 different zoom levels (A) showing Long Island at the center, Gulf of Maine at the top right and Chesapeake Bay at the

bottom left (B) showing Manhattan at the center, Brooklyn extending from Newark to Queens from left to right (C) Centered on Upper Bay

After the interpolation, the boundaries can be automatically extracted as before and the mesh can be written to .grd format in order to be used by SCHISM.

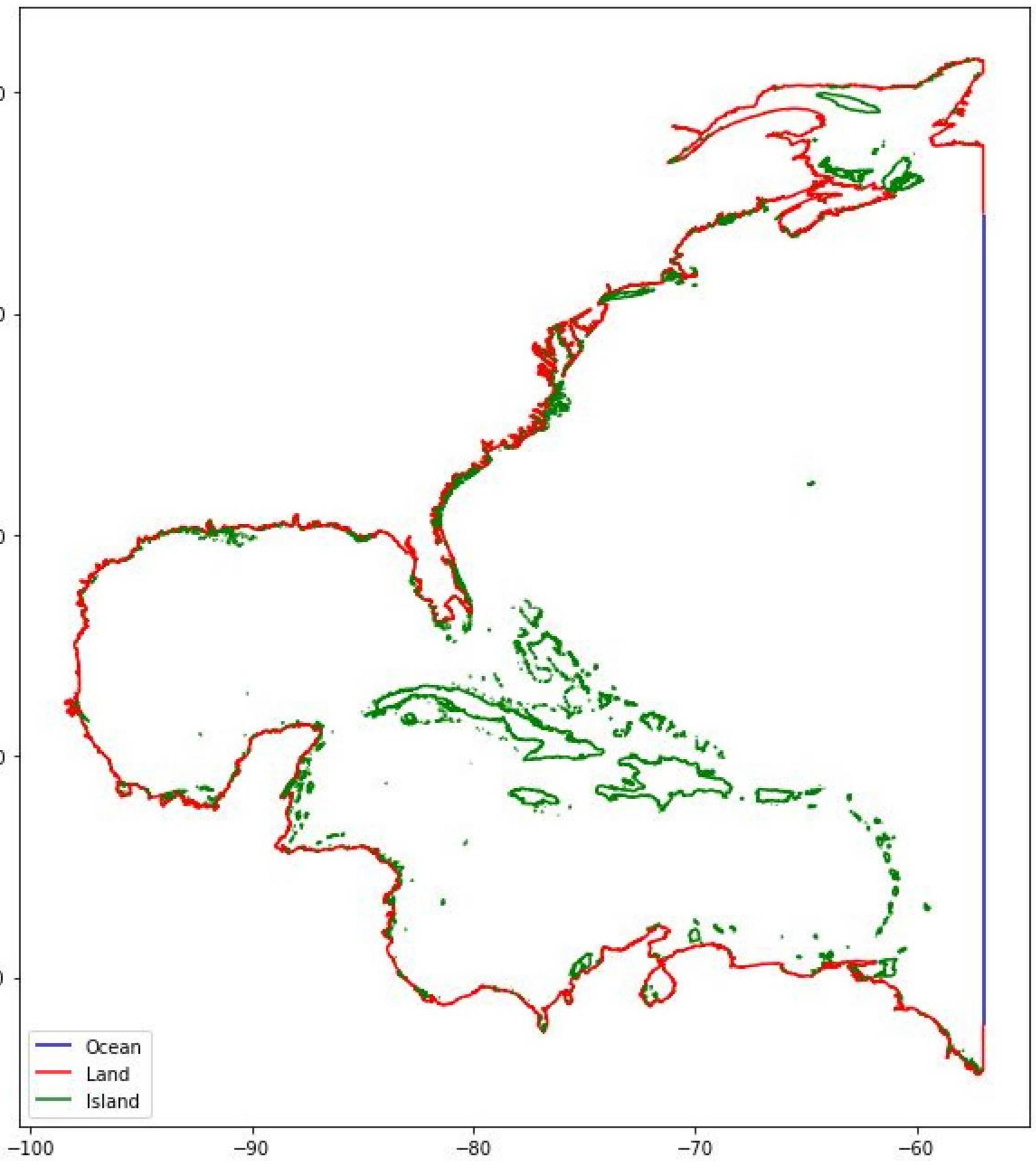

Figure 17 Extracted boundaries for Atlantic domain with threshold of -600m (due to coarse mesh)

Validation

To assess the mesh, a tidal hindcast simulation is done (using SCHISM) and the results are compared with the actual historical measurements. To setup the simulation, PySCHISM (Calzada, 2021) is used. PySCHISM is a preprocessor tool that supports generating input files for many different types of simulation on SCHISM model. The details of how to use PySCHISM is beyond the scope of this document, but the validation results are presented.

Setup

The tidal validation is done by hindcasting tides for part of October 2012. To compare the elevation, moving averages of measurements and simulation results are deducted from their respective values and then the de-trended values are plotted together. The de-trending is necessary due to the model bias.

Since the refined area is only around NYC, the best NOAA stations to use for comparing measurements with simulation results are at The Battery, NY and Bergen Point West Reach, NY (NOAA Center for Operational Oceanographic Products Services (CO-OPS)).

The simulation starts from 10/23/2012 continuing for 6 days with a time step of 150 seconds. The boundaries are only forced with tides. No meteorological forcing is used. Time-series (station) outputs are defined for the two aforementioned stations in NY. The output is written to disk every 30 simulation minutes.

Results

To de-trend simulation and measurements data for comparison a bandwidth of 500 steps is used to calculate the moving average. As shown in Figure 18 and Figure 19 the coarse mesh cannot capture the change in elevation in either of the two stations. However after the refinement, results near both stations are improved; the results in the Battery station matches the measurements almost exactly.

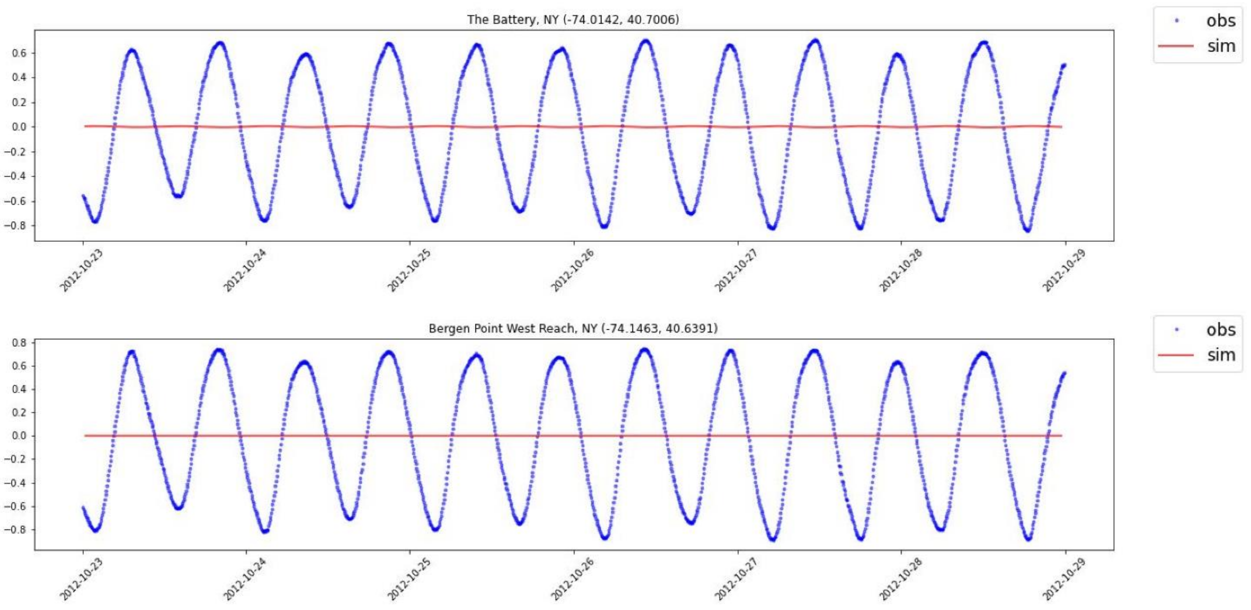

Figure 18 Comparison of water surface elevation before hurricane Sandy (2012) in a tide-only simulation at NOAA tide stations in The Battery, NY and Bergen Point West Reach, NY for the coarse (base) mesh

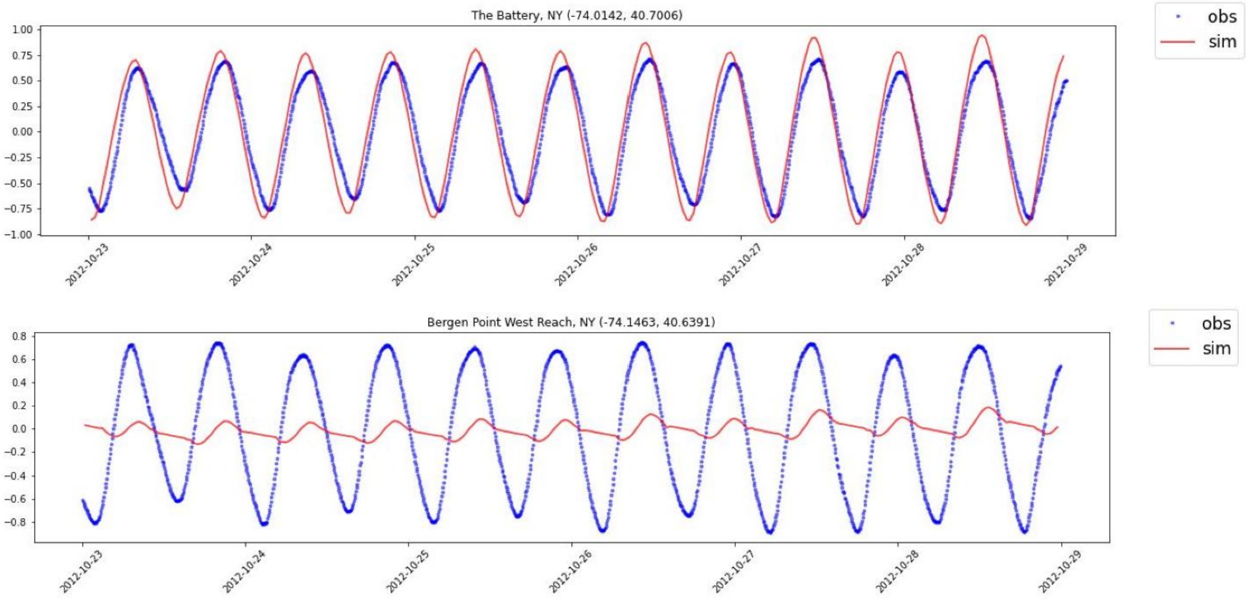

Figure 19 Comparison of water surface elevation before hurricane Sandy (2012) in a tide-only simulation at NOAA tide stations in The Battery, NY and Bergen Point West Reach, NY for the locally refined mesh

10. Summary

In this document we introduced OCSMesh. This free and open source Python unstructured mesh preparation tool relies on Jigsaw library for mesh generation. OCSMesh is capable of handling various input and output formats including rasters, Shapefiles, SCHISM/ADCIRC GRD, and SMS 2DM. OCSMesh workflow is mainly designed around mesh generation from rasters, but custom workflows are also supported to enable, among other things, multi-stage meshing and mesh refinement.

Two common example use cases of OCSMesh are then demonstrated in this document. These examples show the intuitive API for mesh generation and validates simulation improvement due to local mesh refinement.

Similar to any piece of software, OCSMesh would benefit from improvements and new features. The following feature additions and enhancements are being considered for future development:

- Code reorganization to achieve better parallelization
- Use of Dask (Dask Development Team, 2016) for improved distributed workflows and enabling on-cloud implementation.
- Generation of quadrilateral elements (e.g., along detected channels); currently OCSMesh only supports generation of triangular mesh
    - Quad generation to be achieved through either a custom algorithm or use of other mesh engines
- More efficient algorithms to detect thalwegs and channels from the bathymetric data
- Capability of mesh subsetting and merging to provide flexibility in mesh editing
- Automation for fetching DEMs from different sources
- Improve command line interface and extend its capabilities
- Support for configuration file-based mesh declaration
- Support for size function calculation based on other features such as wavelength or feature line (e.g., shoreline) curvatures
- More intelligent meshing strategy for jetties and breakwaters
- Handle cross date line and global domains automatically (currently there are manual workarounds to handle such domains)
- Removing unexpected programmatic side effects

OCSMesh is currently being used in the development of an on cloud project for automated on-demand mesh generation by the authors of this document.

Acknowledgement

We’d like to thank Darren Engwirda from Los Alamos National Laboratory for providing technical support for using JigsawPy.

REFERENCES

Aquaveo. (2017, September 18). *SMS:2D Mesh Files *.2dm*. Retrieved 6 12, 2020 from XMSWiki: https://www.xmswiki.com/wiki/SMS:2D_Mesh_Files_*.2dm

Calzada, J. R. (2021). PySCHISM: A Python interface for SCHISM model runs. GitHub. From https://github.com/schism-dev/pyschism

Cooperative Institute for Research in Environmental Sciences (CIRES) at the University of Colorado, Boulder. (2014). Continuously Updated Digital Elevation Model (CUDEM) - 1/9 Arc-Second Resolution Bathymetric-Topographic Tiles. doi:10.25921/DS9V-KY35

Dask Development Team. (2016). Dask: Library for dynamic task scheduling. From https://dask.org

Engwirda, D. (2014, 11 01). *Locally optimal Delaunay-refinement and optimisation-based mesh generation.* doi:10.1016/j.proeng.2015.10.143

Environmental Systems Research Institute, Inc. (1998). *ESRI Shapefile Technical Description.* Environmental Systems Research Institute, Inc. From https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf

GEBCO Bathymetric Compilation Group 2020. (2020). The GEBCO_2020 Grid - a continuous terrain model of the global oceans and land. doi:10.5285/A29C5465-B138-234D-E053-6C86ABC040B9

Gillies, S. (2007). Shapely: manipulation and analysis of geometric objects. toblerity.org. From https://github.com/Toblerity/Shapely

Gillies, S. (2021). Rasterio. GitHub. From https://github.com/mapbox/rasterio

GNU General Public License, version 3. (2007, June). From http://www.gnu.org/licenses/gpl.html

Harris, C., Millman, K., & van der Walt, S. e. (2020, 9). Array programming with NumPy. *Nature, 585*, pp. 357-362. doi:10.1038/s41586-020-2649-2

Jordahl, K. (2014). GeoPandas: Python tools for geographic data. From https://github.com/geopandas/geopandas

NOAA Center for Operational Oceanographic Products Services (CO-OPS). (n.d.). CO-OPS Map - NOAA Tides & Currents. NOAA COOPS. Retrieved 2021 from https://tidesandcurrents.noaa.gov/map/index.html

QGIS Development Team. (2009). QGIS Geographic Information System. Open Source Geospatial Foundation.

Snow, A. D. (2021). PyProj: Python interface to PROJ. GitHub. From https://github.com/pyproj4/pyproj

Van Rossum, G., & Drake, F. L. (2009). *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace.

Zhang, Y., & Baptista, A. M. (2008). SELFE A semi-implicit Eulerian-Lagrangian finite-element model for cross-scale ocean circulation. *Ocean Modelling, 21(3-4)*, pp. 71-96.

Zhang, Y., Ye, F., Stanev, E., & Grashorn, S. (2016). Seamless cross-scale modeling with SCHISM. *Ocean Modelling, 102*, pp. 64-81.