Domagoj Starčević - Boulder dimension calculation

# Introduction

The objective of this task is to design a robust methodology and create a Python script that can efficiently process two input data layers: a vector layer containing boulder polygons and a gridded surface of bathymetric average values encoded in GeoTIFF format. The script is expected to produce an output vector layer with points approximating the centroids of the input boulder polygons. Each point in the output layer should be accompanied by essential attribute fields, including Poly\_ID, Target ID, Block, Easting, Northing, Water depth, Length, Width, and Height, measured in meters. This document provides a detailed account of the methodology, thought process, and implementation steps involved in the development of the Python script.

# initial setup

## Development and testing environment

To develop and test the code we will use Visual Studio Code, with the installed extension for Jupyter notebook. This will simplify testing and plotting.

We install the packages required for the imported libraries via the built-in terminal.

## Imports

To get started we need to import the libraries that will be used:

import rasterio

from rasterstats import zonal\_stats

import geopandas as gpd

from shapely.geometry import LineString

import matplotlib.pyplot as plt

from rasterio.plot import show

import openpyxl

RasterIO is the library we will use to load the .tif GeoTIFF file with elevation sample values, and analyze the data within it.

Rasterstats will allow gather statistics on the elevation data within zones, which are areas clipped by the boulder polygons.

GeoPandas is a library for working with geospatial data in python. It allows us to load, analyze and manipulate the data from the .shp shapefile containing the boulder polygons.

LineString from the Shapely library will be facilitate finding the boulder polygon length and width.

matplotlib.pyplot and rasterio.plot’s show function will be used to visualize the data during development.

OpenPyXL will help us export the data to Excel during development for checking.

## User defined field and filepaths

We first set the “block” attribute field, this is user-chosen.

#user defined block name

block\_attribute="02"

We now input the filepath for the files we will use:

#path to GeoTiff file

file\_path\_tif='MBES grid/Test\_Encoded\_Depths\_File.tif'

#path to SHP file

file\_path\_shp="Boulder polygons/Test\_Manually\_Picked\_Boulders.shp"

#path to output file

file\_path\_output\_shp="Boulder polygons/Test\_Manually\_Picked\_Boulders\_Centroids.shp"

file\_path\_tif and file\_path\_shp are the input files with elevation values and boulder polygons.

file\_path\_output\_shp is the path for the output file we will generate, containing the boulder centroids and required boulder properties. It will be saved in the same folder as the shapefile with boulder polygons.

# Loading and visualizing the data

## Loading the files

First, we will open the input files and load the data:

mbes\_raster = rasterio.open(file\_path\_tif)

mbes\_array = mbes\_raster.read(1)

mbes\_affine = mbes\_raster.transform

boulders = gpd.read\_file(file\_path\_shp)

mbes\_raster is the object containing the dataset of the raster with elevation values.

The data from a raster can have multiple bands. This one has only the elevation values, which are stored in the first and only band, which we load into the array mbes\_array

A dataset’s [transform](https://rasterio.readthedocs.io/en/stable/api/rasterio.io.html#rasterio.io.DatasetReader.transform) is an affine transformation matrix that maps pixel locations in (row, col) coordinates to (x, y) spatial positions. We create the mbes\_affine which will help us reference the locations in the raster by coordinates.

We read the boulders shapefile and store the data containing the boulder polygons in the ‘boulders’ object.

## Visualization

At this point, we can plot the data in order to make sure we’ve loaded everything correctly (later we comment out this code):

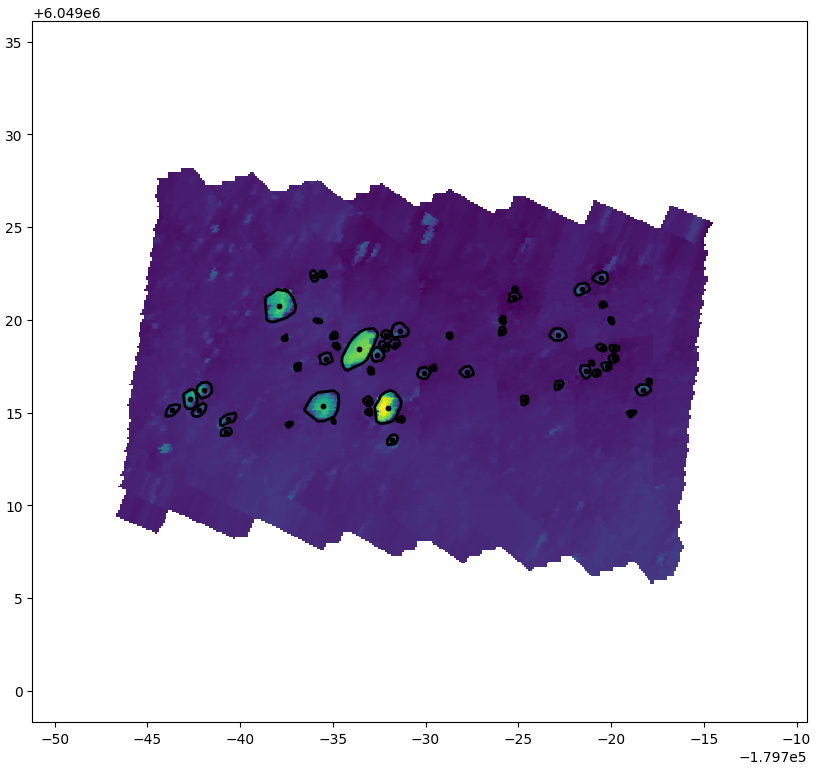
fig, ax = plt.subplots(figsize=(10,10))

show(mbes\_raster,1,ax=ax)

boulders.plot(ax=ax,facecolor='None', linewidth=2)

boulders.centroid.plot(ax=ax,color='black', markersize=10)

We create a figure and on it we display the elevation values, the boulder outlines and the boulder centroids. This will produce the following plot:



As we can see, the zones where the elevation values have sudden changes match with the outlines of the polygons. We see that the raster and the polygons have matching coordinate systems. We also see the polygon centroids where we are expecting them, meaning we are properly accessing them.

# Determining and Setting the boulder attributes

## Setting the “straightforward” attributes

We can determine some of the attributes without any calculation:

boulders['Block']=block\_attribute

boulders['Poly\_ID']=boulders.index

boulders['Target ID']="MBES\_"+block\_attribute+"\_"+boulders['Poly\_ID'].apply(str)

boulders['Easting']=boulders.centroid.x

boulders['Northing']=boulders.centroid.y

Block is the attribute with the value we initially set.

Poly\_ID is the index of each polygon. This is useful for preserving the index data when exporting it.

Target ID is the is a string constructed from “MBES” and the previous two attributes. We need to convert Poly\_ID to a string before concatenating it.

Easting and Northing are the geographic cartesian coordinates for the polygon centroids, so we access them via the x,y coordinates of the centroids.

## Calculating the water depth

We will now find the water depth at the center of each centroid:

depths=[]

for centroid in boulders.centroid:

    depths.append(mbes\_array[mbes\_raster.index(centroid.x,centroid.y)])

boulders['WaterDepth']=depths

First, we create an empty array to contain the depth values.

We then use a for loop to cycle through the boulder centroids.

To get the depth value at the coordinates of the centroid, we input those coordinates in the raster index function, which returns the index in the raster array with the elevation value for those coordinates.

We append the elevation value to the depths array with each cycle.

We then add an attribute for the water depth to the boulders object.

## Calculating the length and width

To calculate the length and width of each boulder, we will first create a minimum bounding rectangle around each polygon.

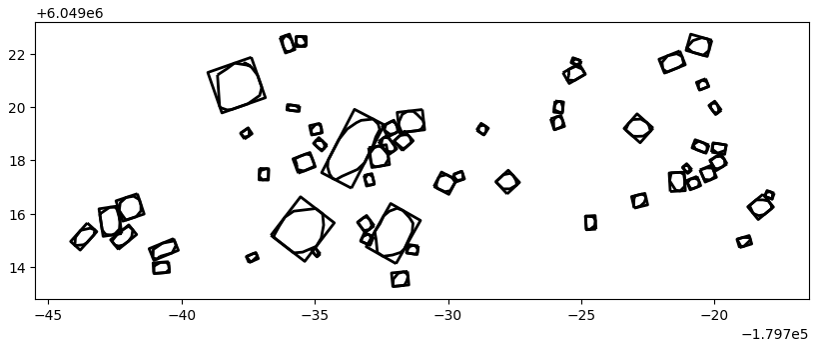
Note: the length calculated this way is not exactly as stated in the task. It is not the largest possible distance within a polygon. E.g. a rectangle polygon’s largest possible distance would be its diagonal. The width would then be a line perpendicular to the diagonal. However, if we consider the length as one of the long sides of the minimum bounding rectangle, and the width as one of the short sides, this resembles more closely what we would intuitively consider length and width dimensions, so that is what we will be calculating. Also, calculating the largest possible distance in a polygon could be quite challenging.

Now we create the rectangles (and we comment out the plotting when done):

rects=boulders.minimum\_rotated\_rectangle()

rects.plot(ax=ax, facecolor='None', linewidth=2)

This will produce the following plot:



We see that the rectangles are properly constructed, and that they can give a good approximation of the boulder lengths and widths.

We now need to determine the lengths and widths of the rectangles. To do this, we will construct a function:

def get\_width\_length(rect):

    rect\_geos = gpd.GeoSeries([rect])

    rect\_geometry = rect\_geos.geometry.iloc[0]

    rect\_coords = list(rect\_geometry.exterior.coords)

    side\_lengths = [

        LineString([rect\_coords[i], rect\_coords[i + 1]]).length

        for i in range(len(rect\_coords) - 1)

    ]

    width = min(side\_lengths)

    length = max(side\_lengths)

    return width,length

The input for the function is a single rectangle (rectangular polygon).

To be able to get the rectangle coordinates, we first convert it from a Shapely geometry to a GeoPandas GeoSeries object. We then access the first and only geometry element in the GeoSeries and access its geometries.

We get the coordinates of the exterior geometry (the points that make up the rectangle).

We then iterate over the coordinates and create LineString objects that represent the sides of the rectangle.

We get four side lengths for each rectangle. The shorter ones (which are identical) represent the width, the longer ones represent the length.

The function then returns the width and length for the input rectangle.

We can now calculate and assign the widths and lengths:

rects\_widths, rects\_lengths = [],[]

for rect in rects:

    width, length = get\_width\_length(rect)

    rects\_widths.append(width)

    rects\_lengths.append(length)

boulders['Width']=rects\_widths

boulders['Length']=rects\_lengths

We create the widths and lengths arrays.

We iterate over each rectangle, and determine its width and length with the help of the function we created. We then append those values to the arrays.

We then add the arrays as attributes to the boulders object.

## Calculating the height

To calculate the height of a boulder we need to find the difference in the depth of the seabed around the boulder and the highest point of the boulder.

To do this, we will use the rasterstats library’s, which allows us to summarize raster datasets based on vector geometries:

boulders\_zonal\_stats = zonal\_stats(boulders, mbes\_array, affine=mbes\_affine, stats=['min', 'max'])

heights=[]

for zone\_stats in boulders\_zonal\_stats:

    heights.append(float(zone\_stats['max']-zone\_stats['min']))

boulders['Height']=heights

The zonal\_stats function takes as input the boulder polygons, the raster array, the affine matrix, and the declaration of which stats we want to calculate. We call this function and store the values in boulder\_zonal\_stats, which is a list of dictionaries with the values for ‘min’ and ‘max’.

In this case, ‘min’ will give us the lowest elevation point within the boulder polygon, which we can assume is somewhere along its edge and represents the depth of the seabed around the boulder, while ‘max’ the highest point on the boulder.

We create the heights empty array, and start iterating through the dictionaries in boulder\_zonal\_stats, appending the difference between the max and min value to the heights array. After that, we store the array as an attribute to the boulders object.

# Output

We now have all the data we need, and are ready to output it:

output\_gdf = gpd.GeoDataFrame(boulders[['geometry','Poly\_ID','Target ID', 'Block','Easting','Northing','WaterDepth','Length','Width','Height']], geometry=boulders.centroid)

output\_gdf.to\_file(file\_path\_output\_shp)

We first create an object output\_gdf which is a GeoDataFrame containing the boulder centroid points as geometry, and the attributes that we calculated in the script.

We write the data to a file in the same folder as the input file.

To check the data, we can also write it to excel (commented out in the script):

output\_gdf.to\_excel("Boulder polygons/Test\_Manually\_Picked\_Boulders\_Centroids.xlsx")

# Conclusion

In summary, the Python script successfully automates boulder dimension calculations from MBES data. Leveraging libraries like Rasterio and Geopandas, it accurately determines centroids, water depth, and dimensions. Tested with the give dataset, the script reliably facilitates boulder analysis.