Guide and Suggested Use for Software for Polygonal Crater Analysis

from
"What Is a Polygonal Impact Crater? A Proposed Framework Towards Quantifying Crater Shapes"
by S.J. Robbins and J.D. Riggs, published in *Earth and Space Science*, doi: 10.1029/2023EA002863

version 1.0.20230425

Step 1: Software

This code is currently available on Zenodo at https://... . It is also available on GitHub at https://github.com/CraterAnalysis/PolygonalCraterIdentification, and either link can be used to download the software.

At the time of this writing, there is the primary code file `PolygonalCraterFitter.py` and the additional required folder `geo`. Both should be placed in your working directory.

The code requires five additional Python packages. Most are standard so you might already have them installed. They are: pandas, matplotlib, scipy, numpy, and shapely. These must be installed into your Python environment (such as using "pip install [package]" or "conda install [package]").

Step 2: Preparing Your Crater Data

The code is set up to analyze one impact crater rim at a time. The file format is a .CSV file with two columns separated by a comma. The first row should be a header row: `lon,lat`

Each subsequent row should be an *x*,*y* point of longitude and latitude in units of decimal degrees. For example, the single point of London, UK, would be: `-0.12574,51.50853`

The .CSV file should list as many rim points as were used to trace the crater rim; ideally, this will be significantly more than 10: As with any shape-analysis work, the more points that are used to define the shape, the better the analysis will be.

For simplicity, the .CSV file can be placed in the same directory as the code from Step 1.

Step 3: Determining the Command-Line Arguments

This code is customizable based on command-line arguments. A total of seven can be used, though the code has built-in defaults for most of them.

- `--data` ~CSV file with crater rim trace
- `--minPoly` ~minimum *n*-gon to try
- `--maxPoly` ~maximum *n*-gon to try
- `--minArcs` ~minimum number of sides to be arcs (including 0)
- `--maxArcs` ~total number of sides to be arcs, though there is a hard-coded that will overrule this
- `--repeat` ~how many times the code will try to fit a polygon with the SAME parameters, where "1" is to do it once, while larger values will fit multiple times so the user can test for robustness
- `--ellipsoid` ~valid values are Mercury, Venus, Earth, WGS84, Moon, Mars, Vesta, Ceres; any other bodies and you will need to add support

For example, if your rim file is named TestCrater.csv, you would like to test a five-sided polygon fit where no sides can be arcs (all must be straight), you want to run it once, and this is a crater that was on Ceres, then you could run this command:

```
> python PolygonalCraterFitter.py --data TestCrater.csv --minPoly 5
  --maxPoly 5 --minArcs 0 --maxArcs 0 --repeat 1 --ellipsoid Ceres
```

Note: An accurate --ellipsoid value must be passed for accurate angles to be reported by the code.

Note: If --minPoly is set to 0, then a circle fit will be done; if --maxPoly is then set to ≥3, the code will skip any polygon with 1 or 2 sides (since no such polygone exists).

Step 4: Understanding the On-Screen Information

Upon starting the command, an initial circle fit will be displayed on the screen. It will remain for one second, and then it will be replaced with an initial polygon fit. This fit will iterate as the code improves it (if it can be improved) before disappearing after a final shape is fit. If the argument --repeat is set to >1, then this will repeat multiple times.

Step 5: Understanding the Outputs

Multiple .TXT files will be output from this code into the same directory where the code resides.

a) The first file output will be the name of the input crater .CSV with "-chi-sq" appended to the file name (*e.g.*, TestCrater-chi-sq.txt). This file will be two columns wide, separated by a tab, and will list in the first column the shape iteration number, and in the second column it will list the chi-squared (χ^2) value of the fit. Only when the χ^2 value was improved will a line be output. If --repeat is set to >1, then each iteration will be in the same file with an extra blank line between them.

b) The second file output will be the name of the input crater .CSV with "-circdata" appended to the file name (*e.g.*, TestCrater-circdata.txt). For each fit shape that is done, one line will be output to this file. The line will have three columns, separated by tabs. The first column is the central longitude, then central latitude, then radius (*not* diameter) of the crater. For any shape that was fit that did *not* include arcs, these columns will have the data for a separate circle fit that the code conducted. For any shape that was fit that *did* include arcs, these columns will have data corresponding to a circle fit that includes each arc in the polygon (the code is restricted such that each arc must be drawn from a common circle).

c) The third file output will be the name of the input crater .CSV with "-edges-circle" appended to the file name (*e.g.*, TestCrater-edges-circle.txt). <u>This file will only be output if --minPoly is set to 0.</u> The output is two-column, tab-separated values, which will have the same number of datapoints as the input .CSV rim trace. It contains the longitude and latitude of the circle-fit points and its sole purpose is for plotting in your own software for comparison purposes.

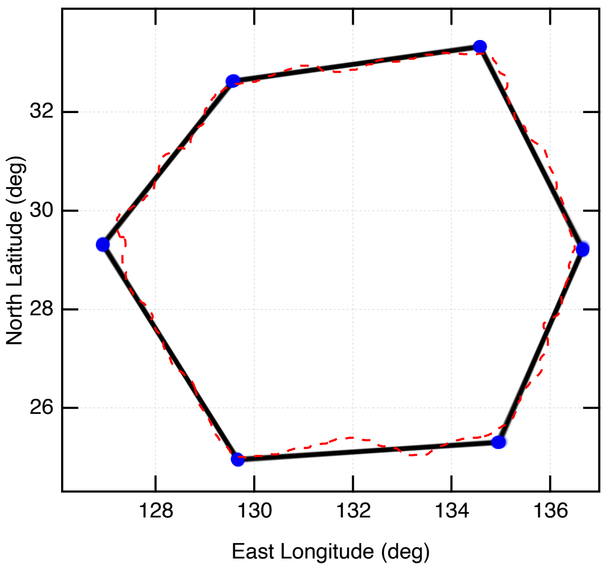
d) The fourth file output will be the name of the input crater .CSV with "-edges" appended to the file name (*e.g.*, TestCrater-edges.txt). This is intended to be a compact form of representing the polygon crater shape that is fit. It is a four-column tab-separated values file where each row is one vertex in a polygon fit. The first column is the longitude of the vertex point, and the second column is the latitude. The third column is whether the vertex point is connected via a straight line (value = 0) or an arc (value = 1) to the next vertex point. The fourth column is the azimuth of the line going from that vertex point to the next one. Angles are clockwise from North, and vertex points go clockwise around the crater. If this is the last row (last vertex), then the third and fourth columns refer to connecting that vertex to the first one. If an arc is included (value = 1 in any row in the third column), then the arc information is in the -circdata file. If --repeat is set to >1, then each iteration will be in the same file with an extra blank line between them.

e) The fifth file output will be the name of the input crater .CSV with "-shape_forDisplay" appended to the file name (*e.g.*, TestCrater-shape_forDisplay.txt). The output is two-column, tab-separated values. It contains the longitude and latitude of the shape for purposes of plotting in your own software for comparison purposes. Since this is for display purposes, it includes a "wrap-around" row where the last row will be the same as the first. If there are straight lines connecting vertex points, then those lines are *not* drawn out in this file since graphing software will simply draw a straight line between them. However, if there are arcs, those will be listed with, by default, 2° increments of arc fidelity (*e.g.*, if an arc would cover ≈30% of the rim, then there will be ≈60 points included for your display purposes). If --repeat is set to >1, then each iteration will be in the same file with an extra blank line between them.

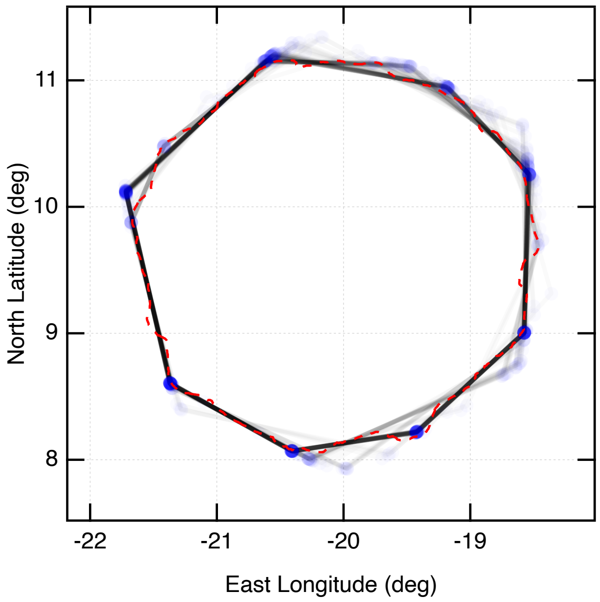
Step 6: Interpreting the Outputs

This code is *not* deterministic, in that the exact same input could lead to different "best" fits.

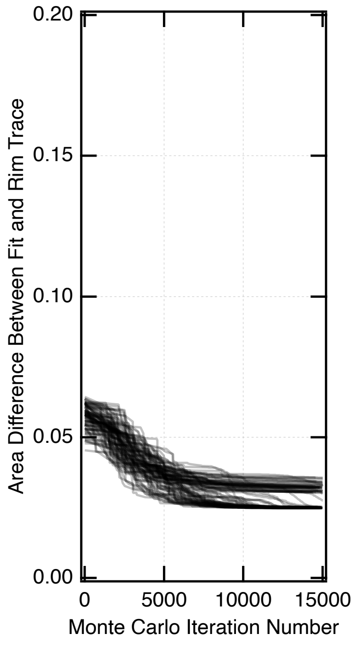
The reason behind this is explained by its driving engine, which is a Monte Carlo approach: The code initially attempts to fit many hundreds to thousands of different possible versions of the shape in question (*e.g.*, a five-sided polygon with five straight sides). The one that minimizes the difference between the test shape and the rim trace is used as a seed for the next step. That next step takes the shape and perturbs it slightly to try to improve the fit (decrease the difference between the fit shape and the input rim). This is iterated until the pre-set maximum number of iterations is reached, at which point the last best shape is considered the final fit.



If there is a very strong underlying shape to the rim that drives the fit, the fit will converge to the same solution each time. This is the case with Fejokoo crater on Ceres (included in the GitHub distribution as a test example) where, when fit with six straight sides, the code converges to the same shape practically every time with nearly identical vertex points. <u>This is interpreted as a case where a hexagon is clearly "the" correct shape to fit to this crater.</u> This example is to the right, where the rim is shown as a dashed red line, the vertices from 100 fits are blue dots, and the lines connecting them show as black. There is some very slight variation in the southern rim vertices, which is why those blue dots are not quite perfect circles, but overall the fit is incredibly consistent.



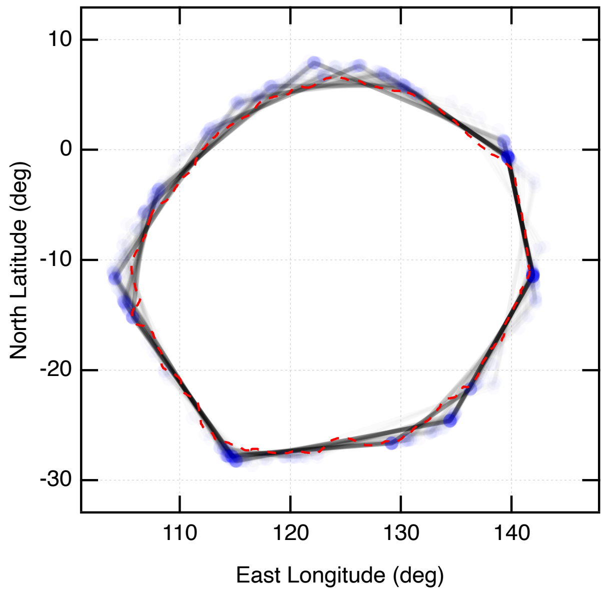
Alternatively, there could be no strong underlying driving shape, despite how some researchers might have interpreted the feature in the published literature. Such an example is Copernicus crater on Earth's moon, shown to the right, with an eight-sided fit. The colors are the same, but this time the 100 different fits have been overlaid with just 2% transparency. That means if there are consistent vertex points and/or edges, those will appear as more solid, while lighter ones indicate other possible (if more rare or less likely) potential solutions. This type of display also underscores the use of the fifth output file type ("-shape_forDisplay"), setting the --repeat value to >1, and visualizing the results.



In this case, it is true that there is a shape corresponding to the true minimum of the area difference between the fit and the rim trace. That is illustrated by the graph to the right of the "-chi-sq" file data. In this case, the true minimum does correspond to the darkest (most repeated) regions of the top graph, so one could interpret this as supporting the idea that Copernicus could be fit by an octagon.

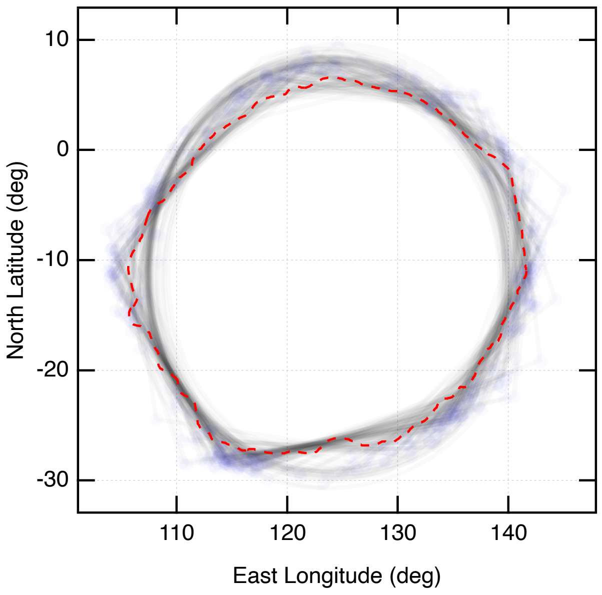
Alternatively, one could also interpret this as an indication ghat there is not really a strong driver here of a shape being fit, which is why there are many different possible final shapes from this code.

An illustrative example that goes further is that of Kerwan crater on Ceres, shown below on the left. This is the result of 100 8-sided fits. While there is some strong structural control to the east, south, and southwest, the remainder of the rim shows a very wide range of possible solutions where none appear significantly more often than any others.



In this case, we would interpret this shape as perhaps better fit by six sides with an arc connecting one of those sides (*e.g.*, the west to northeast rim section).

However, in trying such a fit (to the right), the code converges upon no strong solution whatsoever. So, what went wrong? The code is constrained such that any fitted arcs must be parts of a circle. Connecting the western rim with the northeast with an arc that is a part of a circle takes that arc far north of the actual rim trace, therefore vitiating the fit as a minimum. Shifting the vertex points farther south and connecting those with arcs worsens the fit elsewhere, and so in the case of a six-sided polygon where one side is an arc, there is not any good solution.



Therefore, how might one reasonably interpret Kerwan's shape, given that most researchers "by eye" consider it to be a polygon of some sort? This is a case where the code itself cannot necessarily provide a good, self-contained solution. Instead, the code can be used as a tool to identify the stronger, more repeatable sides, and those sides and vertices used in isolation from the rest of the solution. In this example, we would interpret Kerwan's current form to have at least three – and possibly five – straight edges, and the rest of Kerwan's rim to be uncertain.

While this is not necessarily a satisfying result, one must remember that if this were easy, it is a problem that would have been solved long ago; similarly, if Kerwan had a strong, underlying, "obvious" shape, then there would not be significant disagreement in the literature about how many sides it has.