CGAL Library Testing

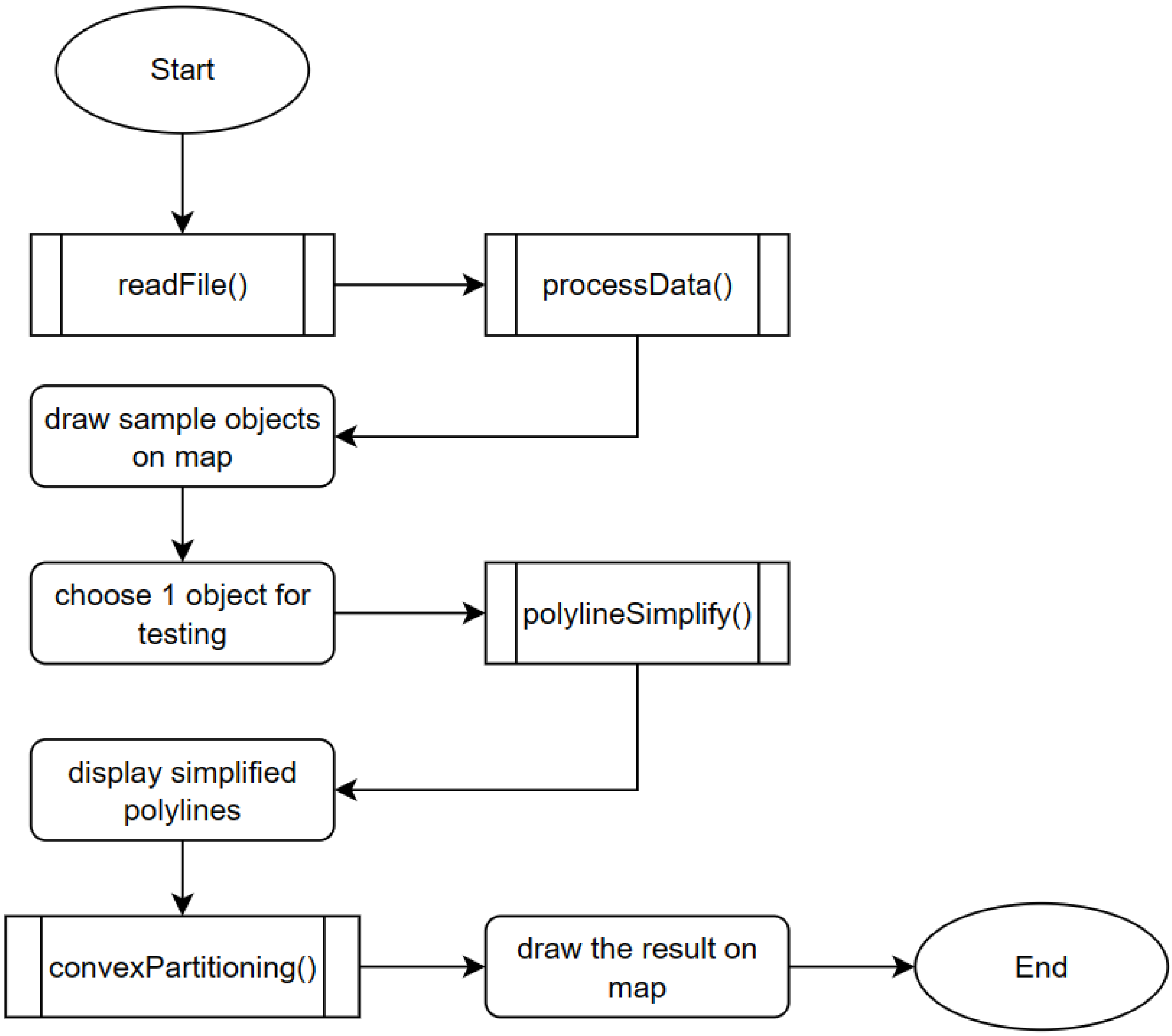
1. Setup:

- My configuration:
  - Window 11
  - Visual Studio 2022 (using msvcpp2022 (v143) as C++ compiler)
  - Boost 1.84
  - CGAL 5.6.1
- First, you need to build your boost library to have .lib files that is compatible with msvcpp2022 (named libboost_...-vc143-mt-gd-x64-1_84.lib or libboost_...-vc143-mt-gd-x32-1_84.lib).
- Copy 2 files .xml from folder 2022_Compilers to the directory `C:\Program Files\Polyspace\R2020a\bin\win64\mexopts`.
- Open MatLab, change the directory to this project folder.
- Run cmd: `mex -setup:'C:\Program Files\Polyspace\R2020a\bin\win64\mexopts\msvcpp2022.xml' C++ -v` to change the mex compiler to msvcpp2022 of VS2022.
- In case you want to rebuild mex functions, change the path variables (to your path) in the script: `build_code.m` and then run it from cmd window of MatLab.
- Run `test.m` to see the test result.

2. Project Structure:

```
├─2022_Compilers
│       ├─msvc2022.xml
│       ├─msvcpp2022.xml
├─PolygonDecomposition
│       ├─convexPartitioning.cpp
├─PolylineSimplification
│       ├─convexPartitioning.cpp
├─Enc.txt
├─build_log.txt
├─build_code.m
├─test.m
├─convexPartitioning.mexw64
├─polylineSimplify.mexw64
├─README.md
└─.gitignore
```

3. Workflow:



4. Polyline Simplification:

1. *Introduction:*

- Polyline simplification is the process of reducing the number of vertices used in a set of polylines while keeping the overall shape as much as possible.
- Vertices are removed according to a priority given for the vertex by a user-supplied cost function which calculates the simplification error. The cost function is a measure of the deviation between the original polyline and the current polyline without the vertex.
- The algorithm terminates when a user-supplied stop predicate returns true, for instance, upon reaching a desired number of vertices or reaching a maximum simplification error.
- CGAL pre-defined 3 cost functions for users to choose different strategies of simplification:
  - Squared_distance_cost: The cost is the maximum value of the squared Euclidean distances between each point on the original polyline.

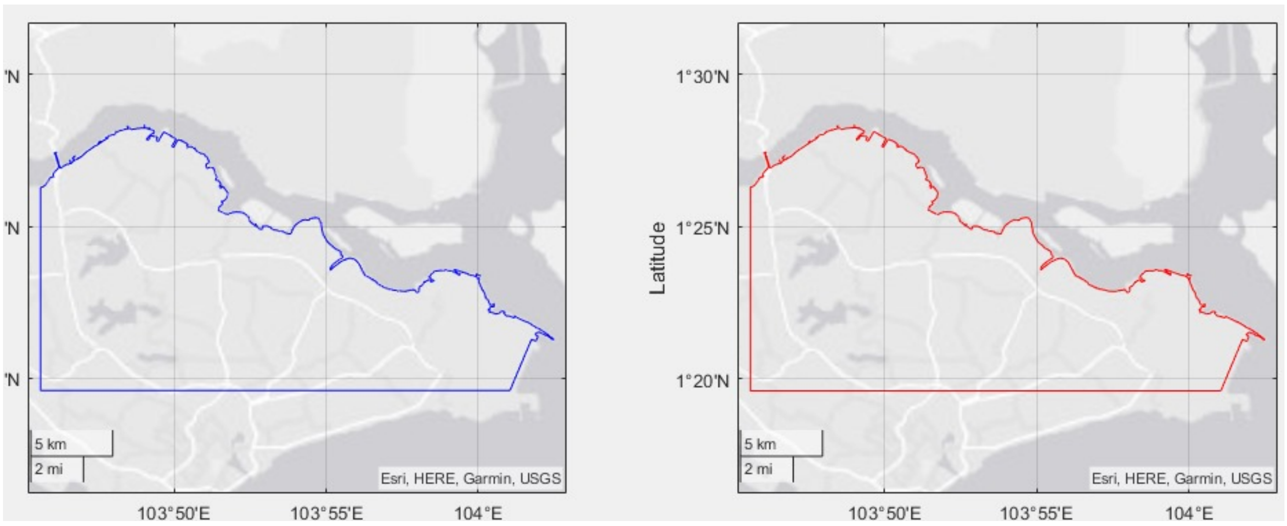
  - Scaled_squared_distance_cost: Use in case we simplify multiple polylines that are close to each other at the same time, when it's important to preserve the separation of adjacent polylines.
  - Hybrid_squared_distance_cost: a combined version of two cost functions above.
  - In addition, we can define our own cost function in case we want to control this simplifying operation.
- CGAL also pre-defined 3 stop conditions:
  - Stop_below_count_ratio_threshold: stop when the percentage of remaining vertices is smaller than a certain threshold.
  - Stop_above_cost_threshold: stop when the cost for simplifying a vertex is greater than a certain threshold.
  - Stop_below_count_threshold: stop when the number of vertices is smaller than a certain threshold.
- Refer to this link for more information.

2. *Implementation:*

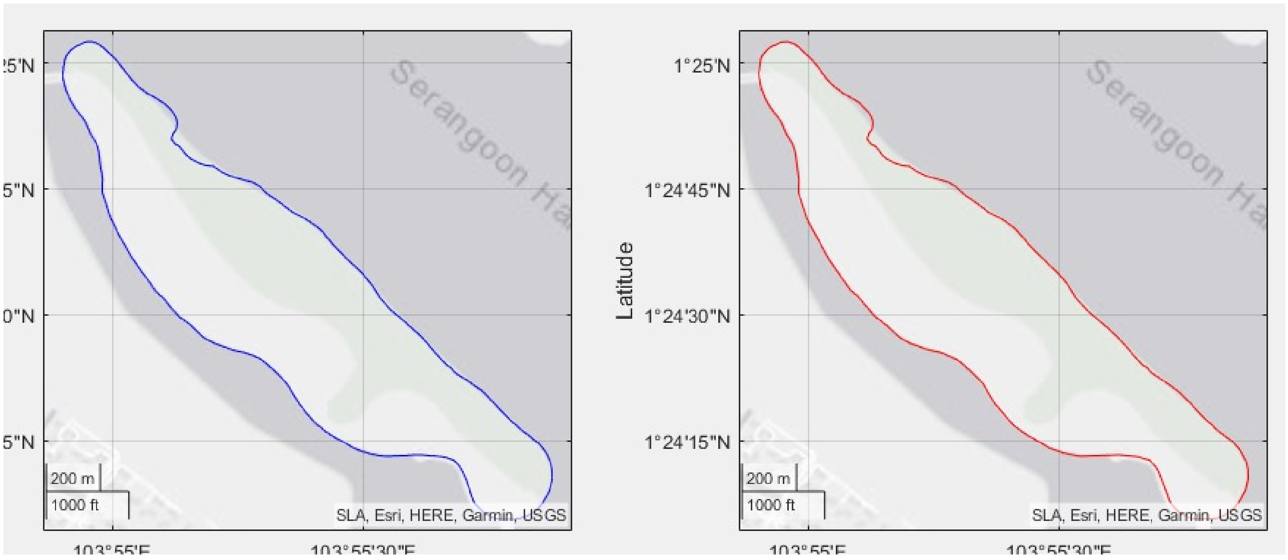
- In this project, I implemented a C++ script called **polylineSimplify.cpp** to perform simplification on closed-loop polygon objects. Then it is compiled into polylineSimplify.mexw64 file by using msvcpp2022 to be able to be invoke from MatLab.
- How to use this function from MatLab?
  - Template of the function: **polylineSimplify(1D-Array y, 1D-Array x, double threshold)**.
  - In which:
    - y is the latitude double array
    - x is the longitude double array
    - threshold is the percentage of remaining vertices for stopping the simplification.
- What do this function actually do inside?
  - Check whether inputs is valid or not
  - Construct the constraints of the polyline by inserting sequentially each coordinate point (built according to latitude and longitude respectively) into a constraint list.
  - Simplify the constructed polyline with pre-defined cost function and stop condition.
    - In the implementation of this function, I used **Squared_distance_cost** and **Stop_below_count_ratio_threshold**.
    - The threshold parameter will be passed to this stop condition.
  - Construct the outputs

3. *Result:*

- Singapore object:
  - simplified with Stop_below_count_ratio_threshold(0.1) and Squared_distance_cost

- 

-
  ```
  The number of points in original object: 4448
  The number of points in simplified object: 445
  ```

- Coney Island object:
  - simplified with Stop_below_count_ratio_threshold(0.2) and Squared_distance_cost
  - 

  -
    ```
    The number of points in original object: 486
    The number of points in simplified object: 98
    ```

5. Convex Partitioning:

1. *Introduction:*

- A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. The partitions are produced without introducing new vertices.
- Convex partitioning
- The points passed to the partitioning functions are assumed to define a **simple** polygon whose vertices are in **counterclockwise** order.
- CGAL provides three functions for producing convex partitions of polygons.
  - One produces a partition that is optimal in the number of pieces.

        - Function: **optimal_convex_partition_2()** is an implementation of Greene's dynamic programming algorithm for optimal partitioning. This algorithm requires ***O(n^4)*** time and ***O(n^3)*** space in the worst case.
    - The other two functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; the first uses a triangulation and the second a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. The triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.
        - Function **approx_convex_partition_2()** implements the simple approximation algorithm of Hertel and Mehlhorn that produces a convex partitioning of a polygon from a triangulation by throwing out unnecessary triangulation edges. This convex partitioning algorithm requires **O(n)** time and space to construct a decomposition into no more than four times the optimal number of convex pieces.
        - Function **greene_approx_convex_partition_2()** is an implementation of the sweep-line approximation algorithm of Greene, which given a monotone partition of a polygon, produces a convex partition in ***O(nlogn)*** time and ***O(n)*** space.
    - **Note**: All 3 algorithms above require object input is a **simple, counterclockwise-oriented** polygon. So the data we use is important, it should be considerd before using.
- Refer to this link for more information.

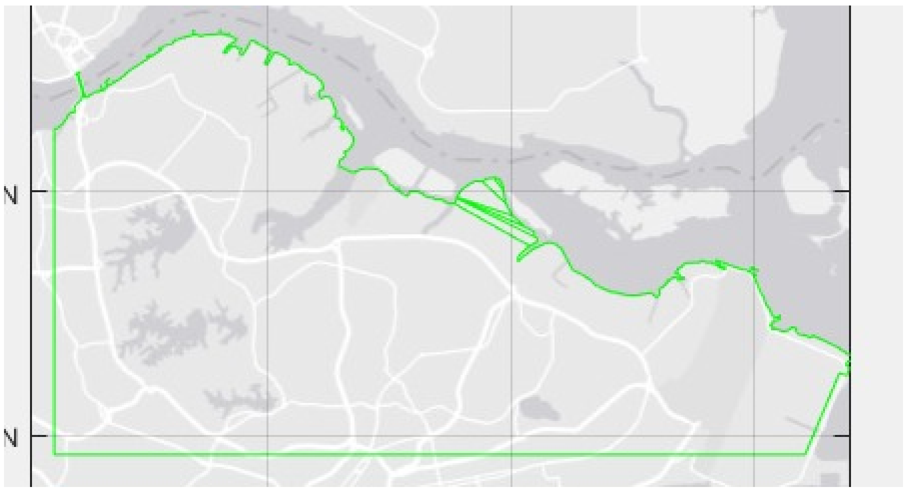
2. *Implementation:*

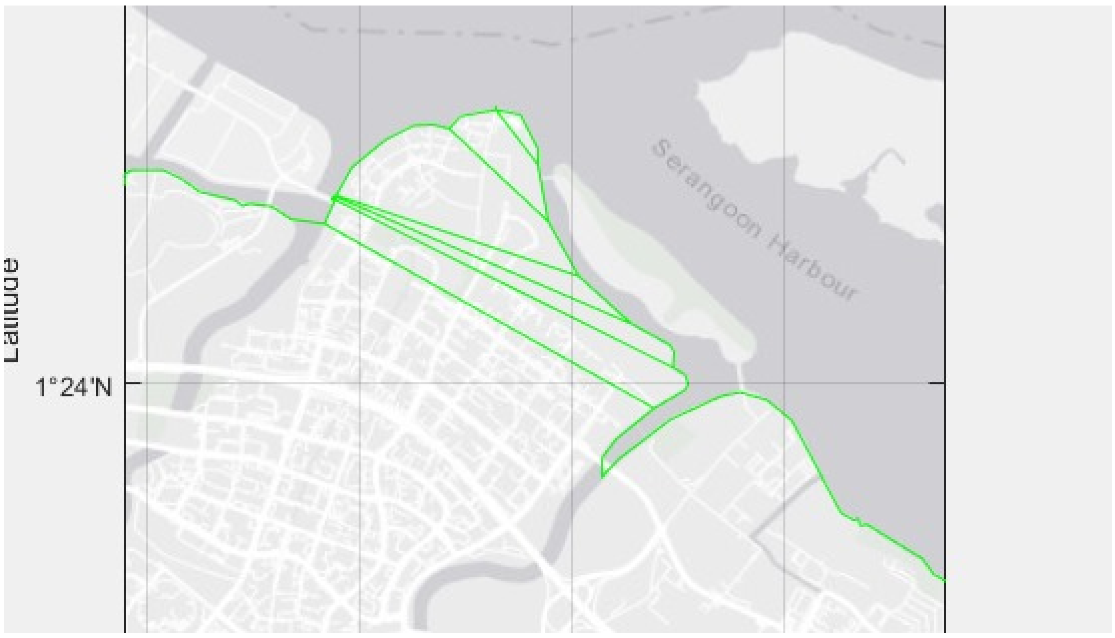
- In this project, I implemented a C++ script called **convexPartitioning.cpp** to perform partitioning on closed-loop polygon objects. Then it is compiled into convexPartitioning.mexw64 file by using msvcpp2022 to be able to be invoke from MatLab.
- How to use this function from MatLab?
    - Template of the function: **convexPartitioning(1D-Array y, 1D-Array x, int mode)**.
    - In which:
        - y is the latitude double array
        - x is the longitude double array
        - mode is the option to chose which algorithm will be used to perform partitioning the input polygon (constructed by latitude and longitude arrays). There are 3 modes:
            - 0: Use Greene's dynamic programming algorithm for optimal convex partitioning
            - 1: Use Hertel and Mehlhorn algorithm for approx convex partitioning
            - 2: Use sweep-line approximation algorithm of Greene for approx convex partitioning
- What do this function actually do inside?
    - Check whether inputs is valid or not
    - Construct the valid polygon for partitioning step by:

        - Pushing sequentially each coordinate point (built according to latitude and longitude respectively) into a pre-defined polygon data structure.
        - Check whether constructed polygon is counter-clockwise oriented or not, if not, reverse the orientation of the polygon.
        - Check whether the constructed polygon is a simple polygon or not, if not, return function without anything.
        - Check whether the constructed polygon is convex or not, if yes, return function without anything.
    - Partition the polygon with the mode passed by user.
    - Check whether each polygon inside the output list of polygons after partitioning is convex and valid or not, if not, return function without anything.
    - Construct the outputs

3. ***Result:***

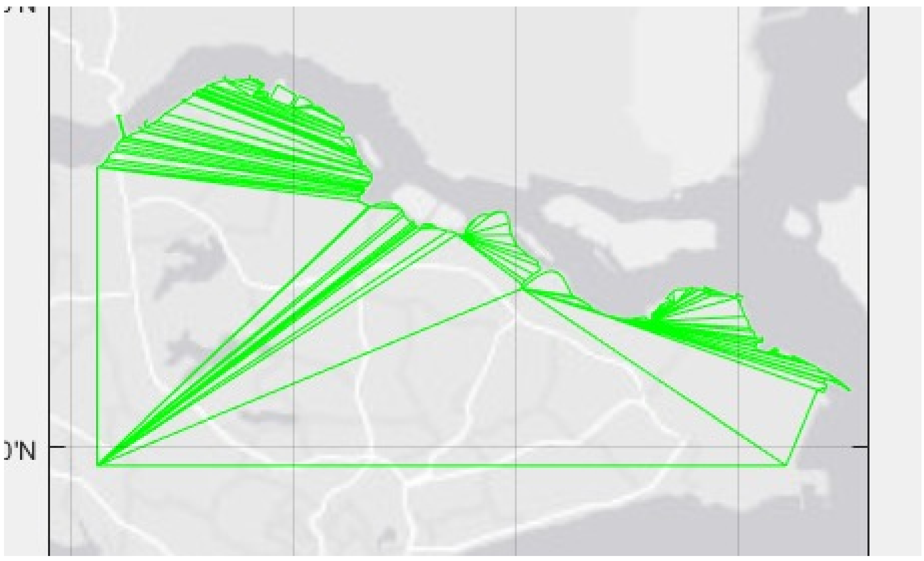
I performed the test with two options: optimal_convex_partition (0) and approx_convex_partition (1) for both Singapore object and Coney Island object.

- Singapore object:
    - optimal_convex_partition: 9 polygons
        - 

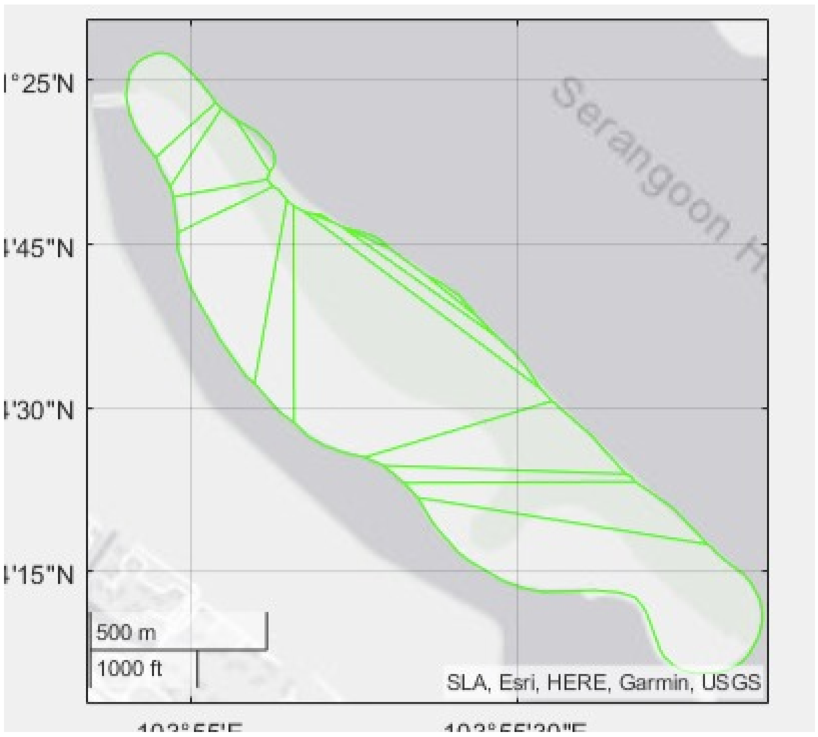
- 

- It seems like the option 0 - optimal_convex_partition does not work correctly on the large objects/polygons.

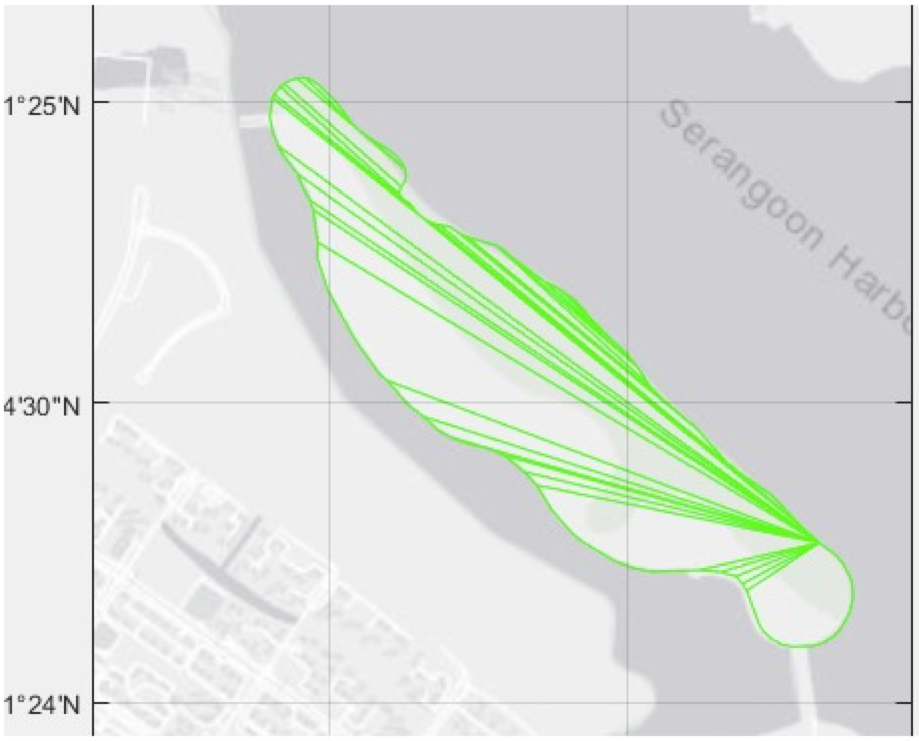
    - approx_convex_partition: 251 polygons

        - 

- Coney Island object:
    - optimal_convex_partition: 17 polygons



- approx_convex_partition: 43 polygons


1°25'N
4'30"N
1°24'N
Serangoon Harb