UNIVERSITY OF EXETER

Computer Science Department

M.Sci. Computer Science
Computer Science Department

Pycart: A Library for Cartogram generation in Python

Candidate

Alexander Rundle

Student ID 690037391

Supervisor

Dr. Rudy Arthur

University of Exeter

Academic Year
2022/2023

Abstract

This report covers the design, development and deployment of a Python library, Pycart, that facilitates the generation of Cartograms, using datastructures integrated from well-known geospatial analysis libraries. The Cartogram aids in geospatial visualisation by altering a map's shape or topology, proportionally to a statistic of interest. The previously undertaken Literature review and Project Specification are first summarised, followed by a review of the design and data pre-processing stages. The implementation of each Cartogram generation method is covered in detail, followed by coverage of documentation, deployment and examples of generation.

Although setting out to implement four generation methods, only two are successfully implemented owing to time constraints. The Pycart library is deployed publicly to Python's largest repository of libraries, allowing easy installation; further, expansive documentation is written that allows easy use of the library. Future avenues of research include the implementation of the Gastner-Newman diffusion method and a Rectangular cartogram method, alongside further optimisation of the current implementations.

	Yes	No
I certify that all material in this dissertation which is not my own work has been identified.	☑	☐
I give the permission to the Department of Computer Science of the University of Exeter to include this manuscript in the institutional repository, exclusively for academic purposes.	☑	☐

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

FFTs Fast Fourier Transforms

CSV Comma Separated Values

ONS Office for National Statistics

CUAs Counties and Unitary Authorities

PyPI Python Package Index

Esri Environmental Systems Research Institute

ISO International Organization for Standardization

VCS Version Control System

CLI Command Line Interface

API Application Programming Interface

JIT Just-in-Time

1 Introduction and Motivation

Visualisation is important within the field of geospatial analysis, enabling researchers and non-researchers alike to recognise and display interesting geospatial datasets. The Cartogram aids geospatial visualisation by altering a map's shape or topology to appear proportionally to a variable of interest, such as Population or Gross Domestic Product. Due to their aesthetic appeal, cartograms have become increasingly useful tools for the representation of political or socioeconomic data and, although useful, there are markedly few examples of code libraries which can create cartograms for generic data and fewer still that utilise Python.

This project explores the development and delivery of the Python library Pycart, or `pycart`, which enables the generation of Cartograms using a variety of well-known cartogram algorithms. The intention behind `pycart` is that it is an 'easy-to-use' library that enables researchers to generate cartograms for generic datasets, using datatypes and representations from common geospatial analysis libraries.

This paper consists of a comprehensive report, starting with a summary of a prior Literature Review in Section 1.1, followed by the initial requirements for this project and the updated requirements under which I have worked. Following this, Chapter 2 covers the pre-processing of datasets used to test the `pycart` library, alongside a brief section on the technologies used during development. Chapter 3 explains the implementation and technical details of the `pycart` library. As the aims of this project are based around the delivery of a Python library, in Chapter 4 I present a gallery of examples generated by `pycart` alongside details of documentation and public deployment, with the aim of highlighting the steps taken to make the `pycart` library meet the 'easy-to-use' requirement.

1.1 Summary of Literature Review

Firstly, I presented prior approaches to showing statistical information over the top of a geographical map, via the use of Augmented Map visualisations. Augmented Map visualisations use multiple correlated views to show a geographic map overlaid with data plots, such as histograms, pie-charts or scatter-plots. Well-known examples include the necklace map [33], in

which underlying data is projected onto a one-dimensional 'necklace' curve around a map, in the form of area-proportional symbols. Augmented Map visualisations are advantageous in the fact that both uni- and multi-variate datasets can be visualised onto the map [10]; however, Augmented Map visualisations create only very weak connections between regions and their underlying data [1].

Two main types of cartogram exist: *topological* and *deformation* cartograms. Topological cartograms extract the topology of the map into highly schematised layouts, using scaled low-complexity polygons to represent regions; topological cartograms are often easier to compute, but sacrifice some recognisability to achieve this [1]. Deformation cartograms, on the other hand, modify the original map itself by pushing, pulling and warping boundaries of regions to achieve area proportionality to given geospatial data [11, 12]. They retain recognisability unlike topological cartograms, but are often more computationally complex [1].

Topological cartograms have, in fact, been used for over 100 years with Levasseur demonstrating one of the earliest known, published examples of a cartogram in 1876 [5]. Haack and Wiechel constructed early deformation cartograms in 1903, creating the "Kartogramm zur Reichstagswahl" (*Cartogram for the Reicshtag election* [18]), where they deformed the constituencies of the German Empire based on population size, as of the 1898 Reichstag parliament election [15, 18].

Algorithms and Techniques

Next, I introduced some well-known cartogram generation algorithms

The first algorithm, the eponymous Dorling circular cartogram introduced by Dorling [7], represents a geospatial dataset as circles of varying radius for each region of the map. In the Dorling algorithm, the circles are generated with area proportional to the data and are subjected to forces as if they were objects in a gravity model. The region circles are repelled by objects with which they overlap and are attracted to objects with which they held adjacencies in the original map. To simulate the movement of the circles, the gravity-like forces are calculated using velocity and acceleration values and are exerted on the circles with a certain amount of friction (thus to make the movement act 'as if it were occurring in treacle' [7]).

I also discussed Rectangular cartograms, which differ from Dorling cartograms in the fact that they schematise regions as rectangles instead of as circles. One popular algorithm is RecMap [17], which attempts to approximate regions using rectangles that are placed as close to the original position as possible, and as close as possible to formerly adjacent neighbours. Heilmann et al. [17] propose a genetic algorithm to solve two placement heuristics. First, the MP1 heuristic aims to maximise the area of a rectangle region w.r.t the original area of the region, alongside minimizing the amount of empty space found in the resulting map. The second heuristic, MP2, aims to maximise the area of a rectangle region w.r.t the original area of the region, whilst also ensuring that the rectangle regions retain their relative position on the map [17, 25].

Next, I covered the first of two deformation cartogram types, the Non-Contiguous cartogram. Non-Contiguous cartograms differ from the Dorling [7] or Rectangular [17] cartograms

in that they scale regions in the geographical map to fit a desired area, without deforming or stretching borders. These cartograms are often very easy to generate and, although they experience a loss of topology, they are still recognisable as they preserve relative geographical positions [24, 25]. Olson [29] introduces a method in which regions are scaled in-place, about their geographical centroids. The density, or data value divided by geographical area, is calculated for all regions and a region with density above a given threshold is chosen as the 'anchor' of the operation. Said anchor determines whether a region is enlarged or shrunk; geographically smaller regions with larger densities are enlarged and regions with smaller densities are shrunk [29].

Finally, I covered the second of the deformation cartograms: the Contiguous cartogram. These cartograms, unlike the Non-Contiguous cartogram, deform the shape of regions themselves so that desired areas are obtained. Further to this, contiguous cartograms place emphasis on the preservation of topology, such that regions maintain any and all adjacencies. Amongst the most popular techniques is the Diffusion-based density equalizing algorithm from Gastner and Newman [11]. Given a grid of population densities calculated from a geographic map, population flows from high-density areas to low-density areas until density is equalised throughout, echoing linear diffusion processes from elementary physics [19]. Through the use of an initial velocity grid $v(r, t)$ and multiple Fast Fourier Transforms (FFTs) [6], Gastner and Newman propose that the cumulative displacement, $r(t)$, of any point on the map at time t represents the operation needed to deform a map and create the cartogram, and is found as in Equation 1.1:

$$r(t) = r(0) + \int_0^t v(r, t')dt' \tag{1.1}$$

Compared to the diffusion method, the Fast-flow method introduced by Gastner et al. [12] only calculates the velocity field once, rather than varying it over t time. After computing the FFTs, all remaining calculations can be replaced with computationally fast calculations (i.e. addition, subtraction, division and simple multiplication) [12]. The fast flow-based method itself produces overall similar results to the diffusion method of Gastner-Newman [11, 12], however requires only 2.5% of the computation time required of the Gastner-Newman diffusion method [12].

Both methods, though, see region adjacencies perfectly preserved. Given this, there is often a trade-off between geographical accuracy and cartographic accuracy due to the inherent deformation that occurs; geographical properties such as shape or relative position cannot be perfectly preserved whilst also ensuring a perfect representation of the underlying geospatial data [25].

1.1.1 Project Specification

Original and Updated Requirements

The functional requirements, as originally presented, stated that I would develop this library for Python 3.9 on a Windows 10 machine. Further, a major technical requirement was the inclusion and integration of the GeoPandas library [21], which intends to make working with geospatial data in Python easier by extending the datatypes used by `pandas` [35] to allow spatial operations on geometries.

I further specified the use of the GeoJSON format for supply of map boundary data [3], alongside two mandatory and two conditional algorithmic requirements. The two mandatory requirements were the inclusion of the Dorling [7] algorithm and Olson's Non-Contiguous cartogram method [29], as these were deemed to be the simplest of the algorithms covered in the Literature Review and techniques that enabled further extension into Rectangular and Contiguous techniques. Second, the two conditional requirements were the inclusions of the RecMap [17] rectangular cartogram algorithm and the Fast Flow-Based algorithm [12]; these algorithms are decisively more complex due to their use of a Genetic algorithm and multiple FFTs, respectively [17, 12]. Their development was based on the fulfilment of the two mandatory requirements, alongside the amount of allocated development time left in the project.

Further notes on the design and development of the `pycart` library can be found in Chapters 2 and 3 respectively; however during the course of development, the initial requirements were found to have needed changes.

The requirement of providing region boundary maps in the GeoJSON format was found to be unnecessary owing to the fact that the main class of `pycart` takes region data and geometries as a single combined GeoPandas `GeoDataFrame` [21], which can be generated regardless of the supplied data's format.

Although I had initially anticipated the inclusion of the Fast Flow-Based algorithm [12], during development I started with implementation of the original Gastner-Newman diffusion algorithm [11]; this algorithm had inspired the Fast Flow-based algorithm [11, 12]. Although initially confident in the implementation, I later found that I had underestimated the amount of time necessary to include this algorithm before deployment to the Python Package Index (PyPI) (see Section 4.2) and later decided to omit it from the final deployment. This was the similar case for the inclusion of RecMap [17]; I had not found enough time during development to include this algorithm and omitted it from the final deployment. In the end, the two conditional requirements were not met owing to a lack of time.

I also stated that I would use the population of the UK dataset, supplied by the Office for National Statistics (ONS) [27], to test the generation of cartograms; more details about the population of the UK dataset [27] can be found in Section 2.1. Further, I stated that a gallery of results would be generated using the implemented algorithms, covering the Counties and Unitary Authorities (CUAs) of England, the districts of South West England and for the boroughs of London.

During development and, after some reflection on the delivery of the `pycart` library, I

identified the need for requirements that helped to determine how I would measure what an 'easy-to-use' library is.

I identified the need for comprehensive documentation that is accessible to any developer, settling on the documentation host Read the Docs [31]. Alongside this, I identified the necessity of deploying `pycart` to PyPI, the largest official host of Python packages and main source of packages for many Python developers.

Following this, I also decided to expand the example datasets from the single, original population of the UK dataset [27] to also include the population of the World [38]; all code in `pycart` was tested on the population of the UK dataset [27], but the inclusion of a second dataset in Section 4.3 demonstrates that `pycart` works on generic datasets.

Further to this, the gallery of examples originally stated the use of data for the CUAs of England, the districts of South West England and the boroughs of London; however, during the pre-processing of the population of the UK dataset in Section 2.1, I discovered that accurately extracting the districts of the South West of England and the boroughs of London would not be possible. Instead, I decided that the gallery of examples should be of the population of the CUAs of the UK and the regions of England from the ONS dataset [27]; also, examples should be generated for the population of Europe and for the World, from the World Bank dataset [38].

The following list shows the updated non-functional requirements:

- The ONS population of the UK dataset [27] must be used to test the cartogram library
- Documentation should be written and hosted by Read the Docs [31]
- The library should be downloadable from PyPI, via `pip`
- A gallery of examples should be generated for the following data:
  - Population of the CUAs of the UK [27]
  - Population of the Regions of England [27]
  - Population of the countries of Europe [38]
  - Population of the countries of the World [38]

2 Design

2.1 Population Datasets

In this section I describe the structure and pre-processing of the two datasets, the ONS Population of the UK dataset [27] and the World Bank Population of the World dataset [38], as mentioned in Section 1.1.1.

The Population of the UK dataset is provided by the ONS in the Comma Separated Values (CSV) format [27] and contains population figures for all regions specified by the UK government in [23]. Population figures are further split by Age, Gender, Births and Deaths; these details are unnecessary for the generation of a simple Population cartogram and as such only the 'Code', "Name', 'Geography' and 'Population' fields are extracted.

Alongside this, the boundaries for the regions of the United Kingdom are supplied by the ONS in the GeoJSON format [3], containing the geometries of each region alongside a variety of key geographical features such as longitude, latitude, region area and region length. For the gallery of examples, a GeoJSON boundary file is provided for the CUAs of the UK [26] and a separate GeoJSON boundary file is provided for the Regions of England [28].

When supplying the dataset to the `pycart` library, the Population field is attached to the fields from the ONS GeoJSON files; the Code fields contain the same codes across the Population CSV and the Boundaries GeoJSON, so the two sets of data can be merged based on the singular common field. Now that the data contains both geometric and statistical data, it can be passed as a `GeoDataFrame` into the `pycart` library.

Similarly, the Population of the World dataset is provided by the World Bank in a CSV format and contains population figures for nearly all countries in the world, giving figures for each year between 1960 and 2021 [38]. For this report, I have chosen to use only the latest data from 2021, so only the 'Country Name', 'Country Code' and '2021' fields are extracted. In the extracted data, Country Code is the International Organization for Standardization (ISO) 3166-1 alpha-3 code, a three-digit unique identifier for each country [34].

The boundaries for the countries of the World are supplied by the Environmental Systems Research Institute (Esri) in the GeoJSON format [9], containing the geometries for each country

alongside details such as name, country area, country length and a ISO 3166-1 alpha-2 code, a two-digit identifier unique to each country [34].

Although both the World Bank population data [38] and the Esri boundary data [9] contain ISO 3166-1 codes, the population data uses the three-digit, alpha-3 variation of ISO 3166-1 and the boundary data uses the two-digit, alpha-2 variation of ISO 3166-1 [38, 9, 34]. In this case, I use the DataPrep library [30] to convert the extracted population's alpha-3 code to alpha-2 format.

Following this, the Population data can be attached to the fields from the Esri GeoJSON files, merging on the now-converted alpha-2 ISO 3166-1 codes; this new `GeoDataFrame` can be passed into the `pycart` library.

2.2 Technologies

As mentioned in Section 1.1.1, I stated that I would be developing `pycart` for Python 3.9 on Windows 10; in this section, I discuss the main libraries and dependencies used by `pycart`.

Table 2.1 shows the main dependencies of `pycart`, with the necessary version constraints to ensure the working of `pycart`. Fixed version constraints are used under the assumption that a user will be using a Virtual Environment.

Component	Version Constraints
Python	$\geq 3.9.7, < 3.10$
matplotlib	3.7.1
GeoPandas	0.12.2
geojson	3.0.1
pandas	1.5.3
numpy	1.24.2
libpysal	4.7.0
shapely	2.0.1
mkdocs	1.4.2

Table 2.1: The main dependencies for `pycart`

The Matplotlib library [20] is included so that the cartograms generated by `pycart` can be plotted and saved to file. GeoPandas and `pandas` are a necessity as they provide an important data structure, the `GeoDataFrame` [21], which is an extension of the `pandas` `DataFrame` [35] that enables geographical study and manipulation. Numpy is included as it provides several useful mathematical operations not found in base Python, as well as the Numpy Array datastructure [16]. The `libpysal` library provides spatial weight functionality [32], which allows the detection of bordering regions used in the implementation of the Dorling algorithm [7] in Section 3.3. The `shapely` library [13] provides the functionality necessary for scaling regions in the Non-Contiguous [29] implementation in Section 3.2 and for translating region circles in the Dorling [7] implementation in Section 3.3. The `mkdocs` library allows the automatic generation of documentation files compatible with Read the Docs [36, 31].

3 Development

The `pycart` library is packaged by the `poetry` packaging and dependency management tool [37], which enables automatic dependency tracking and fine-tuned dependency control. When packaged, `pycart` contains the scripts `cartogram.py` and `border_utils.py`; the main `Cartogram` class and associated generation methods are found in the `cartogram.py` script and border calculations used by the `dorling()` function are found in `border_utils.py`.

3.1 The Cartogram Class

The mainstay of the `pycart` library is the `Cartogram` class, which is contained within the `cartogram.py` script. The `Cartogram` class consists of four attributes, `geo_field`, `id_field`, `gdf` and `value_field`, and two main methods, `non_contiguous()` and `dorling()`.

The four attributes are necessary for the correct generation of a cartogram. However, the most important attribute is `gdf`, which is the necessary geospatial data provided as a GeoPandas `GeoDataFrame` (hence, the name `gdf`) [21]; the format of data supplied during testing is covered in Section 2.1, however the `gdf` should at least have, as seen in Table 3.1, a field containing the region geometries, a field containing the value of which the cartogram would be generated for and an identifier field. Preferably, each row should have a unique identifier as, after generation, the returned `GeoDataFrame` will contain only the ID field, value field and newly transformed geometry field; by supplying unique identifiers, the new cartogram geometries can easily be merged with other data via the identifier field.

Geometry Field	Value Field	Identifier Field
e.g. ”geometry”	e.g. ”Population”	e.g. ”ISO” or ”Name”

Table 3.1: The minimum fields required by the `Cartogram` class.

The `Cartogram` class acts as a staging ground for the later generation methods, ensuring that the relevant fields are present; from this point, generation techniques can be applied to the data held within the `Cartogram` object. There are, also, three static helper functions that are found within the `cartogram.py` script, which aid the successful generation of cartograms.

These static functions are: `_paired_distance()`, `_repel()` and `_attract()`; although used within the methods of the `Cartogram` class, these functions are not intended for external use, hence the preceding underscore by standard convention.

For an example of instantiating the `Cartogram` class, see Code 3.1.

```
from pycart.cartogram import Cartogram
import geopandas as gpd

my_geodf = gpd.read_file("path/to/dataset.csv")

cart = Cartogram(gdf=my_geodf, value_field="Population", id_field="ISO", geo_field="
    geometry")
```

Code 3.1: An example of `Cartogram` instantiation

In Sections 3.2 and 3.3, I discuss the implemented Non-Contiguous [29] and Dorling [7] methods in detail.

3.2 The Non-Contiguous Method

The Non-Contiguous method, as proposed by Olson [29], creates a cartogram by scaling regions proportional to their density (i.e. area divided by statistical value). Due to the simplicity of this method, it was an obvious choice for first implementation. The `non_contiguous()` class function applies the non-contiguous method to the data within a `Cartogram` object, returning a scaled `GeoDataFrame` containing only the `value_field`, `id_field` and modified `geometry_field`.

First, the centroid of each region is calculated as the scaling is applied to a region about its centroid; next, the density for each region is calculated as $d = V_t/A_t$, where t is a given region, V_t is the statistical value of t and A_t is the area of the region t.

Given the densities for all regions, the anchor point that determines whether a region is enlarged or shrunk is chosen, usually as the region with the highest density. The density of the anchor point is used to calculate the scaling factor for each region, using Equation 3.1 where t is the current region and n is the anchor region.

$$S = \sqrt{\frac{V_t}{A_t}} \cdot k, \text{where } k = \frac{1}{d} \text{ and } d = \sqrt{\frac{V_n}{A_n}} \tag{3.1}$$

The `non_contiguous()` class method takes one optional parameter, being `size_value`; the size value parameter acts as a simple multiplier to scaling so that a user can accentuate or minimise the scaling to their liking. Affine scaling is carried out using the Shapely function `scale()` [13] in the x and y dimensions; the newly scaled region geometries replace the supplied geometries.

As the `Cartogram` class receives the geospatial data as a combined `GeoDataFrame`, new columns can be added when calculating the centroids, densities and scaling factors of each region; these columns are not needed in the output, so are removed. This leaves only the orig-

inal `id_field` and `value_field`, alongside the scaled `geometry_field` and scaling factors (as the field `scale`). The three fields supplied are preserved for identification and potential merging with other datasets, and the `scale` field is preserved as it can provide some statistical insight into the scaling of regions, as well as enabling users to tweak region scaling factors on an individual basis should they choose.

An example of the usage of the `non_contiguous()` method can be seen in Code 3.2. Code 3.2 uses the same `cart` variable that is instantiated in Code 3.1.

```
import matplotlib.pyplot as plt

non_con = cart.non_contiguous(size_value=1.0)

# Plot the Non-Contiguous cartogram
fig, ax = plt.subplots(1)
non_con.plot(color='r', ax=ax, edgecolor='0', linewidth=0.1, legend=False)
plt.show
```

Code 3.2: An example of usage of `non_contiguous()`

3.3 The Dorling Method

As mentioned previously in Section 1.1, the Dorling circular cartogram algorithm represents regions as circles of a radius proportional to value density. The class method `dorling()` implements the Dorling algorithm, with the help of two helper functions, `_repel()` and `_attract()`, and a utility function, `get_borders()`, discussed in Subsection 3.3.1.

The Dorling algorithm can be summarised as Algorithm 1, with the `dorling()` implementation following a largely similar structure.

The `dorling()` function takes the following parameters: `iterations`, `ratio`, `friction` and `stop`. The parameter `iterations` determines the number of iterations that the Dorling algorithm runs for, mimicking the central **While** loop seen at Line 4 in Algorithm 1, without needing to determine whether forces are negligible; this further allows a user to manually tweak the length of run-time of the algorithm.

3.3.1 Border Utilities

The implementation of `dorling()` starts by calculating the borders using the function `get_borders()` from the `border_utils` script. This function calculates detects the bordering regions of a given region, for all regions, based on the Queen contiguity measure. The Queen contiguity originates from the first-order contiguity weights proposed by Berry and Marble [2], where two regions are defined as neighbours if they share a common edge or vertex [8], similar to the movement of the Queen piece in chess.

To achieve this, the `get_borders()` function uses the Libpysal `Queen` class [32], which calculates the contiguity based on shared vertices, for all regions in a given `GeoDataFrame`. As the

Algorithm 1 The Dorling circular cartogram algorithm [7]

```
for each region do
    Calculate radius of circle so that its area is proportional to population
end for
while forces calculated are not negligible do
    for each region do
        for each region that overlaps region do
            Record a force away from overlap, in proportion to it
        end for
        for each region which originally neighbours region do
            Record force towards it proportional to distance away
        end for
        if forces of repulsion are greater than attraction then
            Scale forces to less than distance to closest circle
        end if
        Combine the two aggregate forces for each circle
    end for
    for each region do
        Apply forces acting on each circle to its centroid
    end for
end while
```

Queen contiguity relies on shared vertices, the `Queen` class cannot provide neighbours for disconnected components (or islands) that appear in the original map; hence, I remove the islands from the `GeoDataFrame` and proceed with calculating weights for all regions with neighbours.

The weights themselves are found as the length of the intersection between the geometry of a given region and the geometry of its neighbours; this allows for the scaling of forces, found later in the Dorling algorithm, to be proportional to the amount of a region's border that a neighbour occupies [7].

3.3.2 Radius Calculation and Scaling

Following the border calculation, the calculations seen in Algorithm 1 can begin. For the remainder of this section, I refer to each step as acting on a given, *focal* region for simplicity; however, such steps are applied in-turn to all regions in the supplied `GeoDataFrame`.

The `dorling()` function starts by calculating the perimeter of the focal region and replacing the geometries of the focal region with its respective geographical centroid. From here, the centroids of the focal region and all of its neighbours are extracted and the pairwise distance between each is calculated. The pairwise distance between the focal region and its neighbours are calculated using the `_paired_distance()` function which calculates the Euclidean distance [4], using the Shapely `distance()` function [13].

The area of the circle that represents the focal region is determined to be equal to its value field, which allows the unscaled radius of the focal region to be calculated. Following this, the Euclidean distance for all region-neighbour pairs is summed to give the total distance D

between all regions; the total radius of all regions R is calculated as in Equation 3.2, where A_f is the area of a focal region's circle and A_n is the area of a neighbour's circle

$$R = \sum_{f} \sum_{n} \sqrt{\frac{A_f}{\pi} + \frac{A_n}{\pi}}. \tag{3.2}$$

Using the total distance D and total unscaled radius R, a single scaling coefficient k is found as $k = D/R$; using the scaling coefficient k, the scaled radius s_f of the focal region is found as in Equation 3.3.

$$s_f = \sqrt{\frac{A_f}{\pi}} \cdot k. \tag{3.3}$$

Now that the radii of all regions have been calculated as per the **for** loop at Line 1 in Algorithm 1, the main **while** loop at Line 4 can begin. As mentioned previously, the **while** loop is replaced with a **for** loop that is bounded by the supplied number of `iterations`. The optional parameter, `stop`, allows the user to halt execution of the **for** loop at a given iteration, without needing to change the number of `iterations`.

Next, all regions that are within the radius $r = r_{\max} + r_f$, where $r_{\max}$ is the widest radius and r_f is the focal region's radius, are retrieved. For this set of distance-capped neighbours, the distance from the focal region to each neighbour, d_n, is again calculated. The overlap O of a neighbour n onto the focal region f is found as in Equation 3.4

$$O(n, f) = (r_n + r_f) - d_n \tag{3.4}$$

If the overlap value of a given neighbour, O, is greater than 0, then the repulsive forces are calculated (as detailed in Subsection 3.3.3), otherwise the attractive forces are calculated (as detailed in Subsection 3.3.4).

3.3.3 Repulsive Forces

The repulsive forces that act on a region are calculated in proportion to the amount of overlap between the focal region and a given neighbour, using the helper function `_repel()`. This function takes the following parameters: `neighbour`, `focal`, `xrepel` and `yrepel`. These parameters correspond to the given neighbour region, the given focal region and the existing x and y component forces respectively. The force itself is split into its x and y components for easier calculation, which are summed across all neighbours of a given focal region so that the force acts in an aggregated direction.

Using the previously calculated distance between neighbour and focal region, d_n, and overlap O, the x and y components of the repulsive force (F_x and F_y respectively) are found as in Equation 3.5, where (x_f, x_n) and (y_f, y_n) are the x and y positions of the focal and neighbour region centroids, respectively.

$$F_x = O \cdot \frac{(x_n - x_f)}{d_n}$$
$$F_y = O \cdot \frac{(y_n - y_f)}{d_n} \tag{3.5}$$

The `_repel()` function subtracts the calculated x and y component forces from the existing `xrepel` and `yrepel` component forces as the repulsive forces are assumed to be negative forces acting on the focal region, pushing said region away from any overlapping neighbours.

3.3.4 Attractive Forces

The attractive forces that act on a focal region are calculated for each neighbour originally found the geographical map, such that it is proportional to the distance that the focal region is away from the given neighbour [7]. The act of calculating attractive forces is carried out by the helper function `_attract()`, which takes the parameters `nb`, `borders`, `idx`, `focal`, `perimeter`, `xattract` and `yattract`.

Similar to the previous `_repel()` function, the `nb` and `focal` parameters represent a given neighbour region and a given focal region, respectively. Further, the `xattract` and `yattract` are the existing x and y component forces acting on the focal region. The `idx` parameter represents the unique index of the focal region, used for identification and location of the region within the `borders` parameter, which is the adjacency list of border weights as calculated in Subsection 3.3.1.

Before calculation of attractive forces can begin, the overlap between the focal and neighbour regions must be scaled. To achieve this, a binary mask is created that denotes whether a focal region and the supplied neighbour border each other in the original map. If this is the case, the overlap is scaled as in Equation 3.6, where O is the original overlap, W_{fn} is the border weight for focal region f and neighbour n and p_f is the perimeter of focal region f.

$$O_{\text{new}} = \frac{|O| * W_{fn}}{p_f} \tag{3.6}$$

From here, the x and y component forces can be calculated. The component forces calculated using Equation 3.5 are summed with the existing `xattract` and `yattract` forces; attractive forces are assumed to be positive forces acting on a focal region, pulling it in an aggregated direction towards its neighbours.

3.3.5 Movement of Circular Regions

Now that the attractive and repulsive component forces have been calculated, the sequence of events following Line 12 of Algorithm 1 can begin.

The distance covered by the attractive and repulsive forces are calculated using the Numpy `hypot` function [16], which calculates the distance as $D = \sqrt{{F_x}^2 + {F_y}^2}$; this is calculated for

both the attractive and repulsive forces.

If the repulsive distance is greater than zero, the total x and y forces are calculated as in Equation 3.7, where ρ is the parameter `ratio`, D_{repel} is the repulsive distance and D_{attract} is the attractive distance. The combined total forces are calculated for attractive and repulsive forces respectively.

$$F_{\text{total}} = (1 - \rho) \cdot F_{\text{repel}} + \rho \cdot (D_{\text{repel}} \cdot \frac{F_{\text{attract}}}{D_{\text{attract}} + 1}) \tag{3.7}$$

The `ratio` parameter determines the ratio of attractive forces to repulsive forces, which allows the user to tweak how effective they want the relevant forces to be. As the `ratio` parameter increases, the effectiveness of the attractive forces increases. The default value $\rho = 0.4$ is selected due to its use in the original method proposed by Dorling [7].

Given the total x and y forces, the vectors by which the focal region's circle will move are calculated as $\vec{x} = \mu \cdot F_{x_{\text{total}}}$ and $\vec{y} = \mu \cdot F_{y_{\text{total}}}$, where $\vec{x}$ is the movement vector's x component, $\vec{y}$ is the movement vector's y component and μ is the `friction` parameter. The `friction` parameter allows the user to artificially increase or decrease the effects of the total forces, and the default value of 0.25 is used as in the original method proposed by Dorling [7].

The movement is acted upon using the Shapely `translate` function [13], which takes the geometries from the `GeoDataFrame` alongside the x and y vector components and, finally, a buffer is added around the centroid of each region equal to the radius defined in Subsection 3.3.2. An example of the usage of the `dorling()` function can be seen in Code 3.3, which again uses the same `cart` variable that is instantiated in Code 3.1.

```
import matplotlib.pyplot as plt

# Run the default Dorling algorithm for 200 iterations
dorling = cart.dorling(iterations=200)

# Plot the Dorling cartogram
fig, ax = plt.subplots(1)
dorling.plot(color='w', ax=ax, alpha=0.8, zorder=0, edgecolor='0', linewidth=0.1,
    legend=False)
plt.show
```

Code 3.3: An example of usage of `dorling()`

4 Results and Evaluation

4.1 Documentation

As stated in the updated requirements found in Section 1.1.1, a non-functional requirement of the `pycart` library would be to host documentation on the popular site Read the Docs [31]; Read the Docs is a free documentation repository that allows users to publicly host their documentation, automatically building from a Version Control System (VCS) of their choice.

I start by writing a main README, containing key information about the library; I further write an in-depth installation and tutorial guide, alongside comprehensive Application Programming Interface (API) reference pages that detail the syntax and workings of the implemented methods. I then use MkDocs to generate a Read the Docs compatible project [36, 31], configuring the project using the configuration file `mkdocs.yml` and the MkDocs Command Line Interface (CLI) to complete generation.

Like most open-source libraries, the VCS of choice for `pycart` is Github [14], which Read the Docs has native support for; in this case, hosting is a simple matter of authenticating a link between the two services and selecting the repository to host. The `pycart` documentation is accessible at the following: `https://ecmm428-pycart.readthedocs.io/en/latest/`.

4.2 Deployment to PyPI

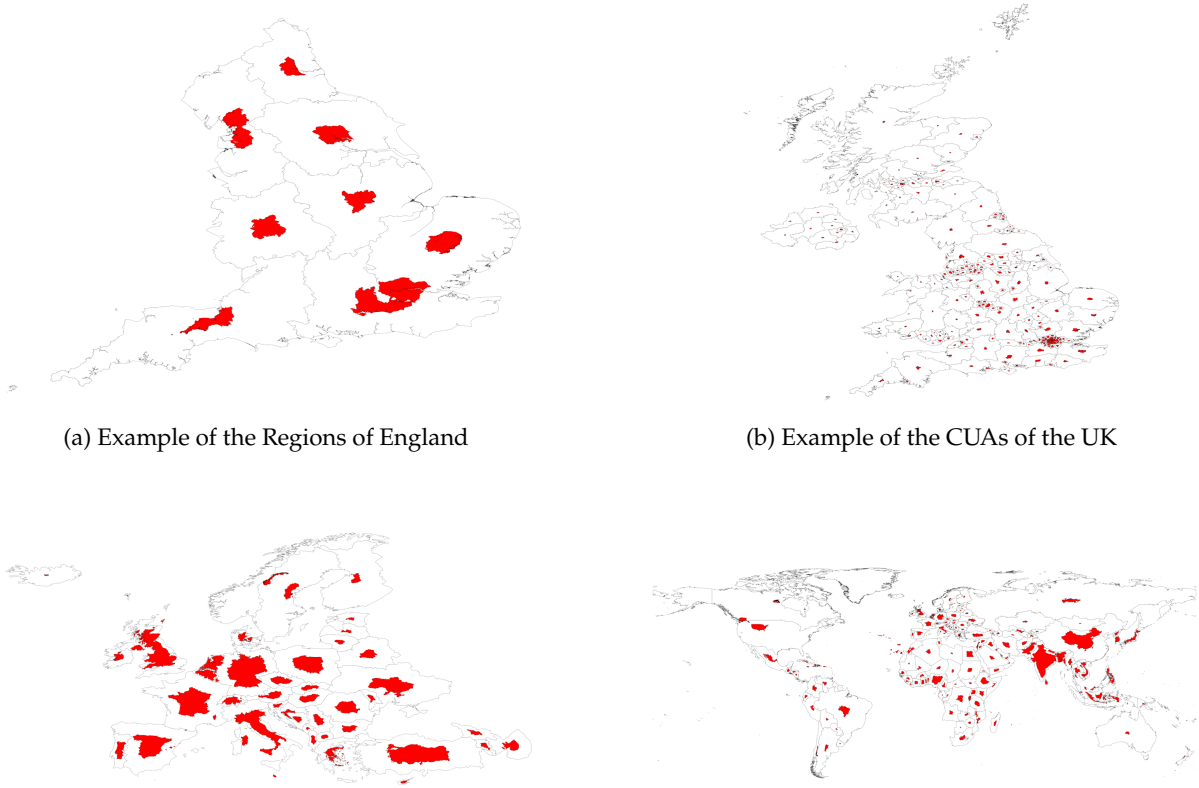
Python is often used by researchers due to its modularity and wealth of libraries with the Python Package Index (PyPI) allowing Python users easy access to thousands of libraries, alongside allowing users to search across various tags and library features.

Given the requirements in Section 1.1.1, I felt it was of importance to enable users to import and use `pycart` with minimal effort, which deploying to `PyPI` ensures. I started by configuring an account with PyPI and generating the necessary API token, allowing me to publish my package remotely from a CLI. I further shared the API token with `poetry` [37], allowing me to publish `pycart` to PyPI via `poetry` [37]. The `pycart` library can be seen on PyPI at the following: `https://pypi.org/project/pycart/`.

4.3 Examples of Generation

In the following section, I display and evaluate the results generated by the `pycart` library using the gallery of examples as defined in Section 1.1.1.

I begin by applying the `non_contiguous()` function to the ONS Population and World Bank datasets [27, 38] as per the requirements in Section 1.1.1, the results of which can be seen in Figure 4.1; higher resolution, full-page examples can be found in the appendix (Section 5.1). First, Sub-figure 4.1a shows the regions of England, with the Non-contiguous method applied to them [29]. As can be seen with the red regions, they have all been shrunk, except for the region of London, which remains the same; this suggests that the region of London has a very high population density.



(a) Example of the Regions of England

(b) Example of the CUAs of the UK

(c) Example of the Countries of Europe

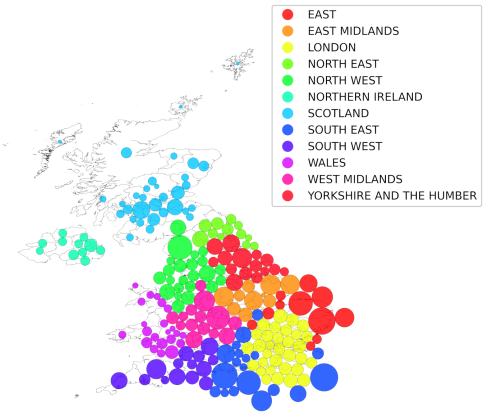
(d) Example of the Countries of the World

Figure 4.1: Examples of the Non-Contiguous method [29] applied as specified in Section 1.1.1

Sub-figure 4.1b shows the Non-Contiguous cartogram of the CUAs of the UK [26], which displays large shrinkage for many regions in the North of Scotland, as well as in Northern Ireland and Wales. Further, the boroughs of London remain largely the same size, meaning they have a high population density. Sub-figure 4.1c shows the Non-Contiguous cartogram for the countries of Europe [38, 9]. In this case, the region with the largest density which appears to have been chosen as the anchor of the algorithm is the Netherlands, with the surrounding regions being shrunk in proportion to distance; further, the Scandinavian countries like Norway, Swe-

den and Finland appear to have very low population densities. Finally, Sub-figure 4.1d shows the Non-Contiguous cartogram of the countries of the World [38, 9], with the India appearing to be the anchor region with the highest population density and other countries being shrunk around it. Some other notable results include Canada, which has been shrunk massively due to its large land area but sparsely distributed population, and Nigeria, which appears to be have the largest population density in Africa.

(a) Example of the Regions of England, after 100 iterations



(b) Example of the CUAs of the UK, after 100 iterations

(c) Example of the Countries of Europe, after 500 iterations

(d) Example of the Countries of the World, after 100 iterations

Figure 4.2: Examples of the Dorling method [7] applied as specified in Section 1.1.1

I follow this by applying the `dorling()` function to the ONS and World Bank population datasets [27, 38]; in the case of the Dorling algorithm, the number of iterations run for each example is found in their respective caption. First, Sub-figure 4.2a shows the regions of England represented as circles, with their names displayed within each (again, higher resolution versions of these examples can be found in the Appendix, Section 5.1). This example, when compared to the Non-contiguous example, seems to suggest that the London area has a similar population density to the South East of England, despite the area of London being only 8% of the size of the South East. This fact can also be seen in Sub-figure 4.2b, which shows the CUAs of the UK colour-coded by their region in England (or by countries outside of England that they occupy); the yellow region represents London and, once again, appears to show that the boroughs of

London have a larger population density than most other regions in the UK.

Sub-figure 4.2c shows the Dorling cartogram for the countries of Europe [7, 38, 9], with each circular region labelled with the ISO-3166 alpha-2 code of the country that it represents; in this cartogram, it can be seen that Germany has the largest population density, followed by the UK, Turkey, France, Italy and Spain. This cartogram, unlike the non-contiguous examples, provides a perfect visual representation of the low geographical accuracy obtained by the Dorling method; one can see many circle regions that appear nowhere near their underlying country's position, although relative position and region adjacencies are well preserved. Finally, Sub-figure 4.2d shows the Dorling cartogram produced for the countries of the World [7, 38], which shows how the countries of India and China have massively higher population densities than any other countries in the world; further, this is another great example of low geographical accuracy as, due to the sheer size of India, much of Asia is placed in locations very far from their respective country's position.

5

Conclusion and Analysis

The aim of this project was to develop a Python library that facilitates the easy creation of Cartograms, for data supplied as `GeoDataFrames`; this resulted in the creation and deployment of the `pycart` library. In this paper, I have discussed the background knowledge that underpins the cartogram and some famous examples of cartogram generation algorithms, as well as a set of functional and non-functional requirements that this project was to meet. Although the implementation of four different generation methods was specified, I did not succeed in the implementation of two methods, a Rectangular cartogram method [17] and the Gastner-Newman diffusion method [11]; these two methods were conditional requirements, subject to the completion of the Non-Contiguous and Dorling methods [29, 7].

Moreover, the design stage of this project covered the structure of the ONS and World Bank population datasets [27, 38], as well as vital pre-processing stages and a short description of the key technologies used by the `pycart` library. The pre-processing of the key datasets was carried out in separation from the library itself, meaning the key methods of `pycart` can be used on general datasets.

Each component of the `pycart` library was covered in detail, starting with the main class `Cartogram` and its structure, followed by an in-depth explanation of the `non_contiguous()` method and the `dorling()` method. The Non-Contiguous method shrinks all regions based on their density and an external anchor point, and the Dorling method schematises regions as circles of density-proportional radius.

The non-functional requirements stated in Section 1.1.1 are all used and implemented successfully, with the `pycart` library providing publicly hosted, in-depth documentation and the ability to be easily downloaded from PyPI, the most-used repository of Python libraries. Further, the gallery of examples were generated and shown in Section 4.3, with Figures 4.1 and 4.2 showing the examples for both methods. These examples show the methods in action, with Figure 4.1 showing regions being shrunk around high-density anchors and Figure 4.2 showing circular regions having moved as per the movement model defined by Dorling [7].

5.1 Further Research

This projects shows that the creation of an 'easy-to-use' Python library that facilitates the generation of cartograms is certainly achievable, but there are many avenues of improvement and expansion that future research could follow.

Firstly, one might continue and finalise the implementation of the RecMap [17] and Gastner-Newman Diffusion [11] methods, as per the original requirements in Section 1.1.1. These could both propose interesting development challenges owing to their individual complexities; the RecMap algorithm utilises Genetic Algorithms to resolve the MP1 and MP2 heuristics that define rectangular cartograms [17], whilst the Gastner-Newman diffusion method utilises multiple FFTs to calculate the velocity and density grids used [11].

One might also commit to the optimisation of the current implementations of the Non-Contiguous and Dorling methods, respectively [29, 7]. Due to the number of nested **For** loops within the Dorling algorithm (see Algorithm 1), the computation time of the Dorling cartogram can take tens of minutes dependent on the dataset size. For example, generating Sub-figure 4.2d requires the processing of 215 regions and takes 10 minutes and 54 seconds, on average. This is not necessarily an unmanageable amount of computation time however, it still remains that one could optimise the method so that it runs in significantly less time; this could be achieved by vectorizing operations through the use of `numpy` [16] or through the use of a Just-in-Time (JIT) compiler such as Numba [22], which uses the `jit` decorator to speed up array and loop operations.

In summary, this project has explored the design, development and deployment of a Python library that facilitates Cartogram generation, in the form of the `pycart` library. The `pycart` library implements only a small handful of the wide array of generation methods that exist, and I greatly anticipate the further implementation of more advanced methods, whether it be in my own work or in the work of others.

References

[1] Md Jawaherul Alam, Stephen G Kobourov, and Sankar Veeramoni. "Quantitative measures for cartogram generation techniques". In: *Computer Graphics Forum*. Vol. 34. 3. Wiley Online Library. 2015, pp. 351–360.

[2] Brian JL Berry and Duane Francis Marble. *Spatial analysis: a reader in statistical geography*. Prentice-Hall, 1968.

[3] Howard Butler et al. *The geojson format*. Tech. rep. 2016.

[4] David Cohen. *„Precalculus: A Problems-Oriented Approach", Cengage Learning*. Tech. rep. ISBN 978-0-534-40212-9, 2004.

[5] *Congrès international de statistique à Budapest: neuvieme session du 29 août au il septembre 1876 : Programme*. Congrès international de statistique à Budapest: neuvieme session du 29 août au il septembre 1876 : Programme v. 1-3. Impr. "Athenneum", 1876. URL: https://books.google.co.uk/books?id=etw7AQAAMAAJ (visited on 11/18/2022).

[6] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[7] DFL Dorling. "Area cartograms: their use and creation". In: *Concepts and techniques in modern geography series*. Environmental Publications, University of East Anglia, 1996.

[8] Robin Dubin, AS Fotheringham, and PA Rogerson. "Spatial weights". In: *The Sage handbook of spatial analysis* (2009), pp. 125–158.

[9] Environmental Systems Research Institute. *World Countries - Generalized*. Redlands, California, United States, 2022. URL: https://hub.arcgis.com/datasets/esri::world-countries-generalized/about (visited on 04/09/2023).

[10] Georg Fuchs and Heidrun Schumann. "Visualizing abstract data on maps". In: *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.* IEEE. 2004, pp. 139–144.

[11] Michael T Gastner and Mark EJ Newman. "Diffusion-based method for producing density-equalizing maps". In: *Proceedings of the National Academy of Sciences* 101.20 (2004), pp. 7499–7504.

[12] Michael T Gastner, Vivien Seguy, and Pratyush More. "Fast flow-based algorithm for creating density-equalizing map projections". In: *Proceedings of the National Academy of Sciences* 115.10 (2018), E2156–E2164.

[13] Sean Gillies et al. *Shapely*. Version 2.0.1. Jan. 2023. DOI: 10.5281/zenodo.5597138. URL: https://github.com/shapely/shapely.

[14] github. *GitHub*. 2023. URL: https://github.com/.

[15] H Haack and H Wiechel. "Kartogramm zur Reichstagswahl". In: *Zwei Wahlkarten des Deutschen Reiches in alter und neuer Darstellung mit politischstatistischen Beiworten und kartographischen Erläuterungen. Gotha: Justus Perthes Verlag* (1903).

[16] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[17] Roland Heilmann et al. "Recmap: Rectangular map approximations". In: *IEEE Symposium on Information Visualization*. IEEE. 2004, pp. 33–40.

[18] BD Hennig. "Kartogramm zur Reichstagswahl: an early electoral cartogram of Germany'". In: *Bulletin of the Society of Cartographers* 52.1–2 (2018), pp. 15–25.

[19] Josef Honerkamp and Hartmann Römer. *Theoretical Physics: A Classical Approach*. Springer Science & Business Media, 2012.

[20] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[21] Kelsey Jordahl et al. *geopandas/geopandas: v0.8.1*. Version v0.8.1. July 2020. DOI: 10.5281/zenodo.3946761. URL: https://doi.org/10.5281/zenodo.3946761 (visited on 11/22/2022).

[22] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[23] Ministry of Housing, Communities & Local Government. "Local government structure and elections". In: (2021). URL: https://www.gov.uk/guidance/local-government-structure-and-elections (visited on 11/18/2022).

[24] Sabrina Nusrat, Md Jawaherul Alam, and Stephen Kobourov. "Evaluating cartogram effectiveness". In: *IEEE Transactions on Visualization and Computer Graphics* 24.2 (2016), pp. 1077–1090.

[25] Sabrina Nusrat and Stephen Kobourov. "The state of the art in cartograms". In: *Computer Graphics Forum*. Vol. 35. 3. Wiley Online Library. 2016, pp. 619–642.

[26] Office for National Statistics. "Counties and Unitary Authorities (Dec 2020) UK BGC". In: (2020). URL: https://geoportal.statistics.gov.uk/datasets/ons::counties-and-unitary-authorities-december-2020-uk-bgc-1/about (visited on 11/18/2022).

[27] Office for National Statistics. "Estimates of the population for the UK, England and Wales, Scotland and Northern Ireland". In: (2020). URL: https://www.ons.gov.uk/peoplepopulationandcommunity/populationandmigration/populationestimates/datasets/populationestimatesforukenglandandwalesscotlandandnorthernireland (visited on 11/18/2022).

[28] Office for National Statistics. "Regions (Dec 2020) EN BGC". In: (2020). URL: https://geoportal.statistics.gov.uk/datasets/ons::regions-dec-2020-en-bgc/about (visited on 11/18/2022).

[29] Judy M Olson. "Noncontiguous area cartograms". In: *The Professional Geographer* 28.4 (1976), pp. 371–380.

[30] Jinglin Peng et al. "DataPrep.EDA: Task-Centric Exploratory Data Analysis for Statistical Modeling in Python". In: *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China*. 2021.

[31] "Read the Docs". In: (2020). URL: https://readthedocs.org/ (visited on 04/10/2023).

[32] Sergio J Rey and Luc Anselin. "PySAL: A Python library of spatial analytical methods". In: *Handbook of applied spatial analysis: Software tools, methods and applications*. Springer, 2009, pp. 175–193.

[33] Bettina Speckmann and Kevin Verbeek. "Necklace maps." In: *IEEE Trans. Vis. Comput. Graph.* 16.6 (2010), pp. 881–889.

[34] International Organization for Standardization. *Codes for the representation of names of countries and their subdivisions – Part 1: Country code*. ISO 3166-1:2020. Geneva, Switzerland, Aug. 2020.

[35] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

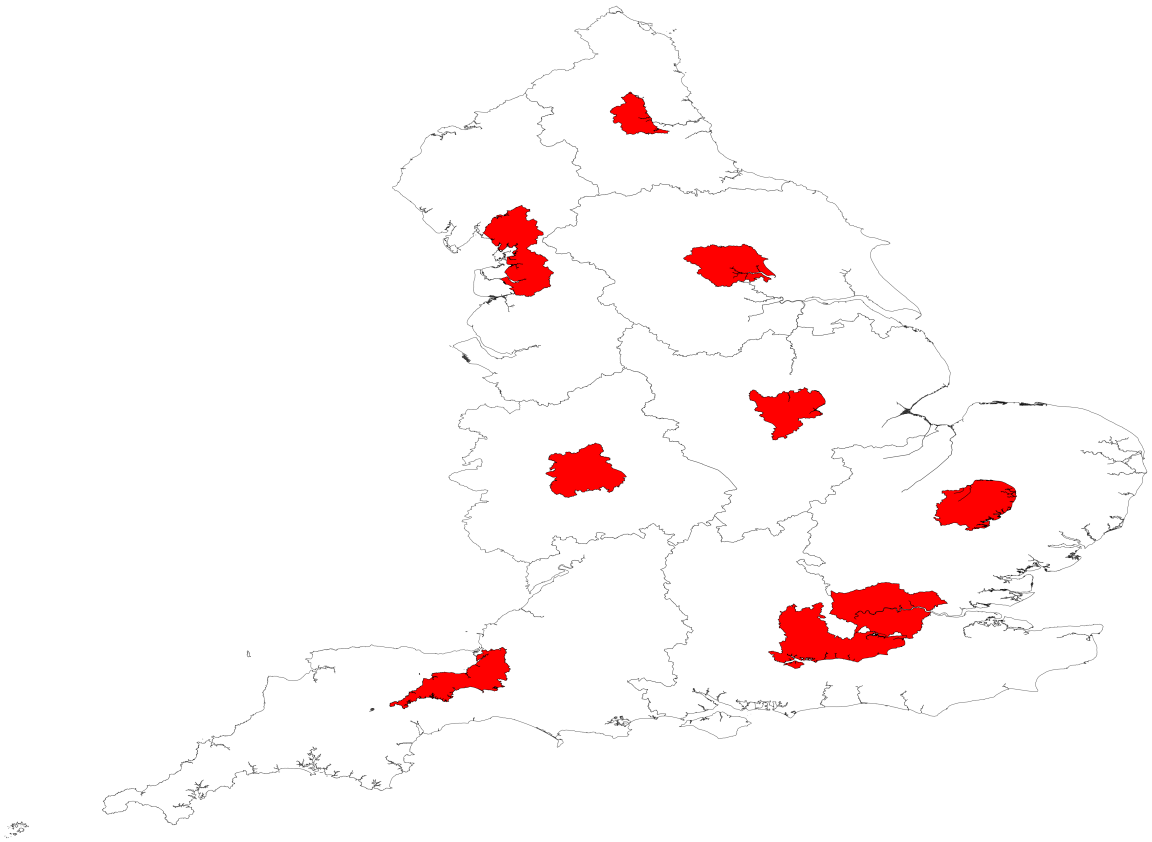
[36] The MkDocs development team. *mkdocs [Source Code]*. Version 1.4.2. Nov. 2022. URL: https://github.com/mkdocs/mkdocs (visited on 04/10/2023).

[37] The Poetry development team. *poetry [Source Code]*. Version 1.4.2. Apr. 2023. URL: https://github.com/python-poetry/poetry (visited on 04/09/2023).

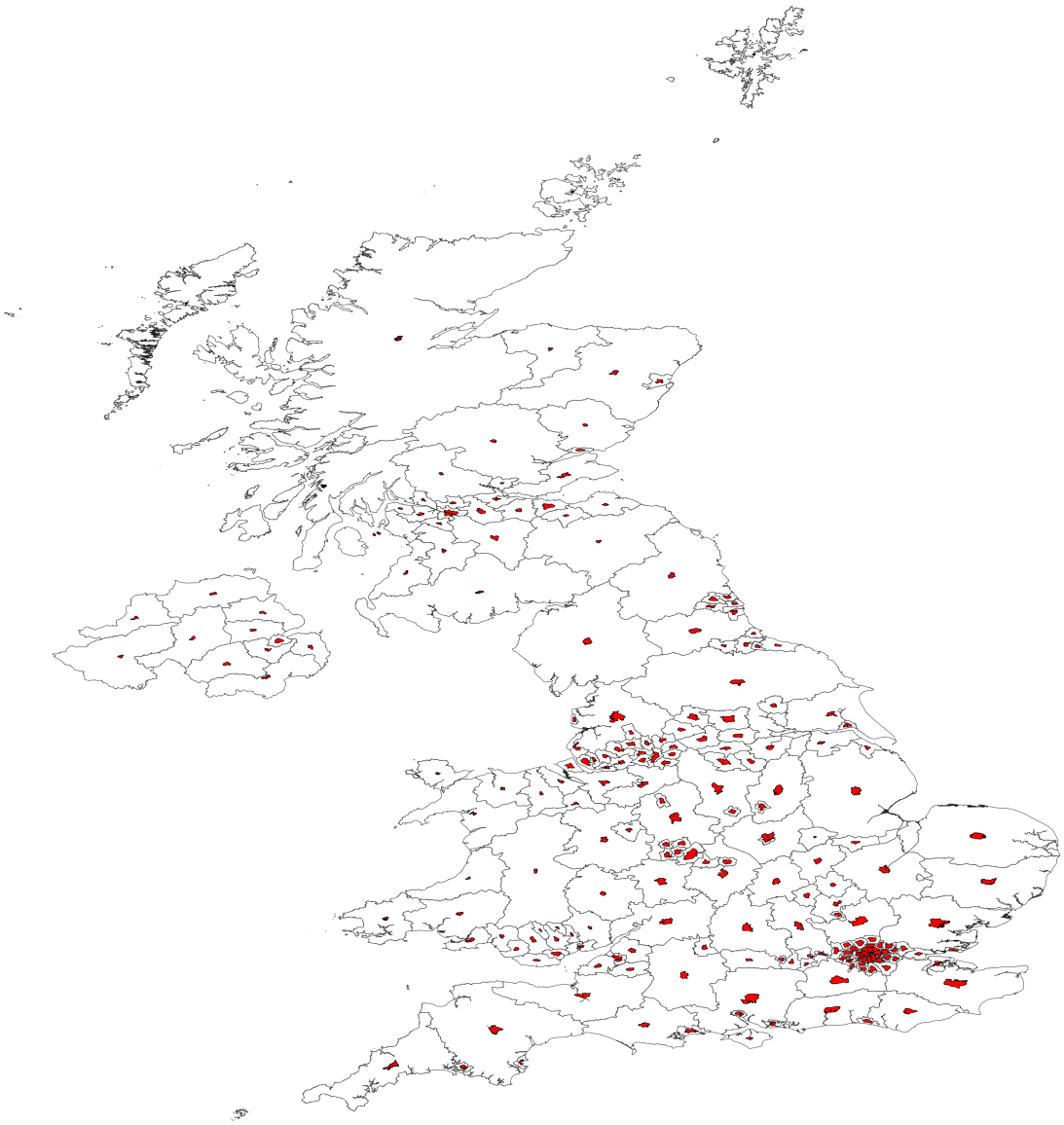
[38] World Bank. *Population, total: Total Population of the World*. Washington, DC, United States, 2021. URL: https://data.worldbank.org/indicator/SP.POP.TOTL?end=2021&name_desc=false&start=1960&type=shaded&view=chart (visited on 04/09/2023).

Acknowledgments

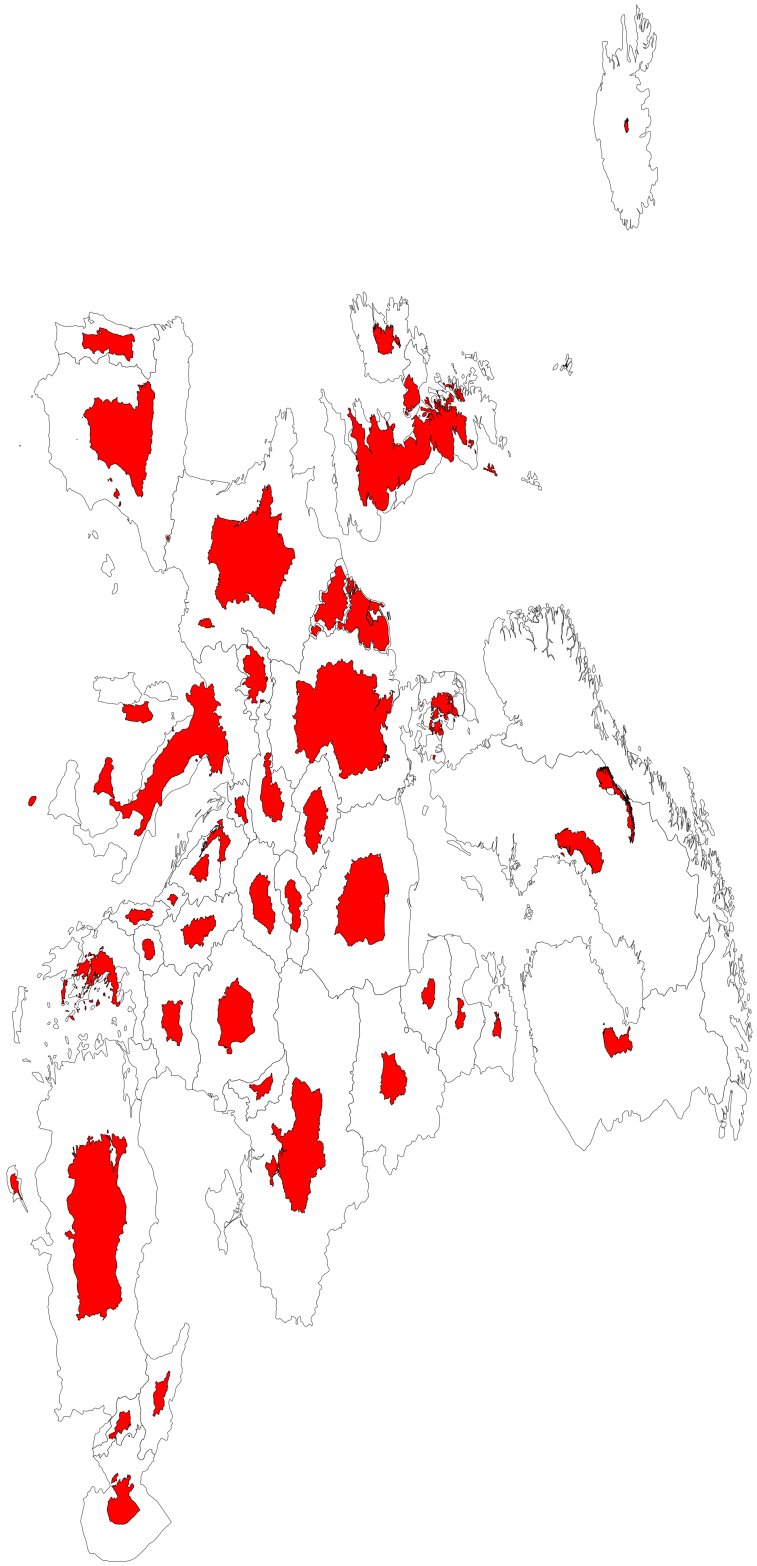
Appendix



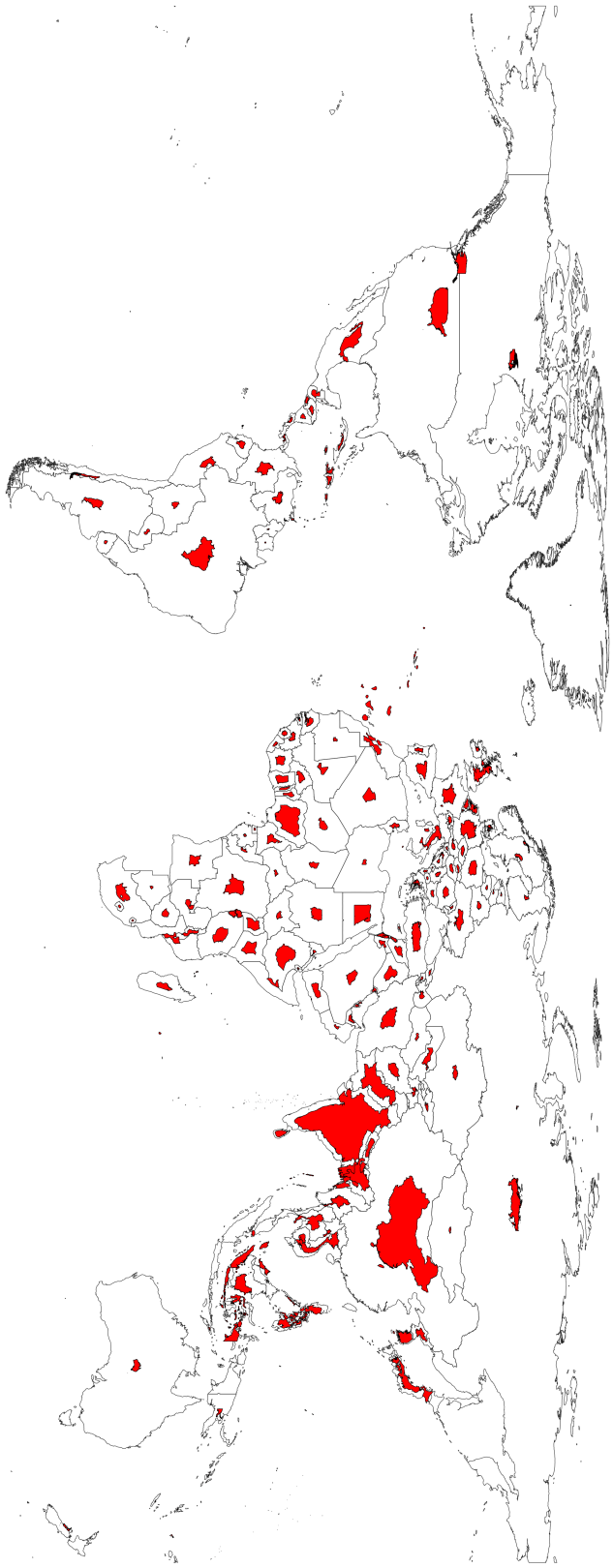
Full Resolution example of the Non-Contiguous method [29] applied to the Regions of England [28, 27]



Full Resolution example of the Non-Contiguous method [29] applied to the CUAs of the UK [26, 27]

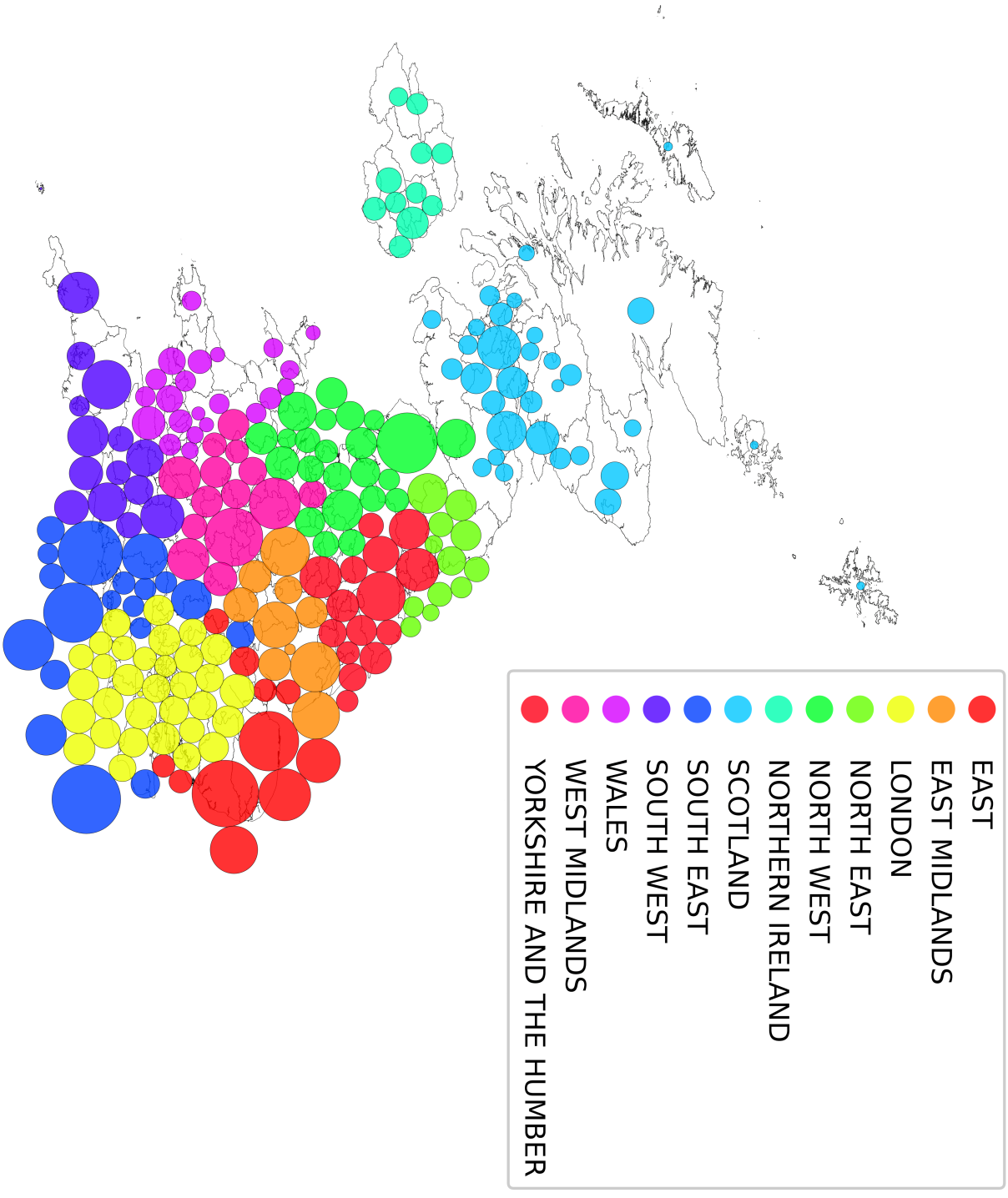


Full Resolution example of the Non-Contiguous method [29] applied to the Countries of Europe [38, 9]

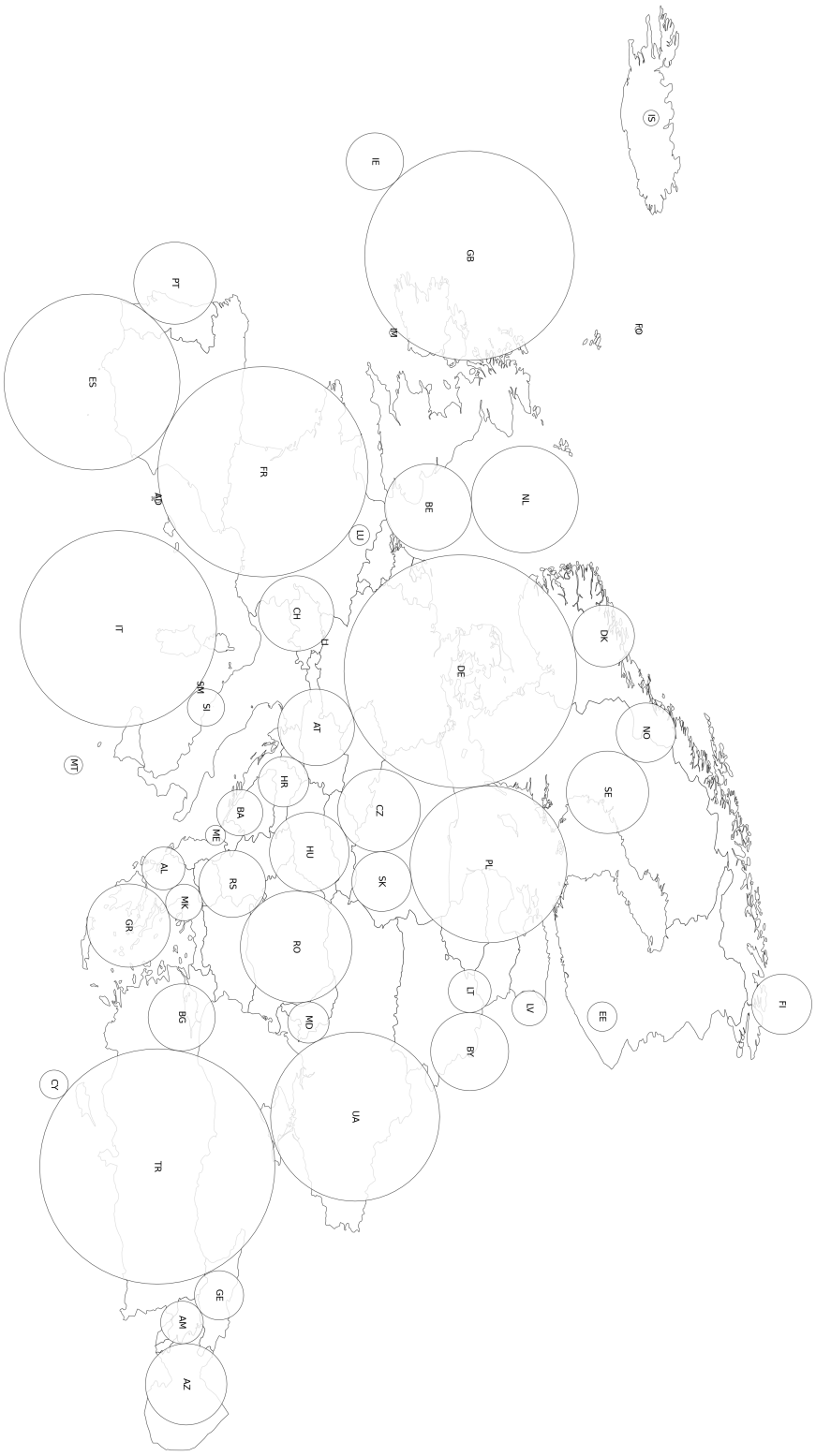


Full Resolution example of the Non-Contiguous method [29] applied to the Countries of the World [38, 9]

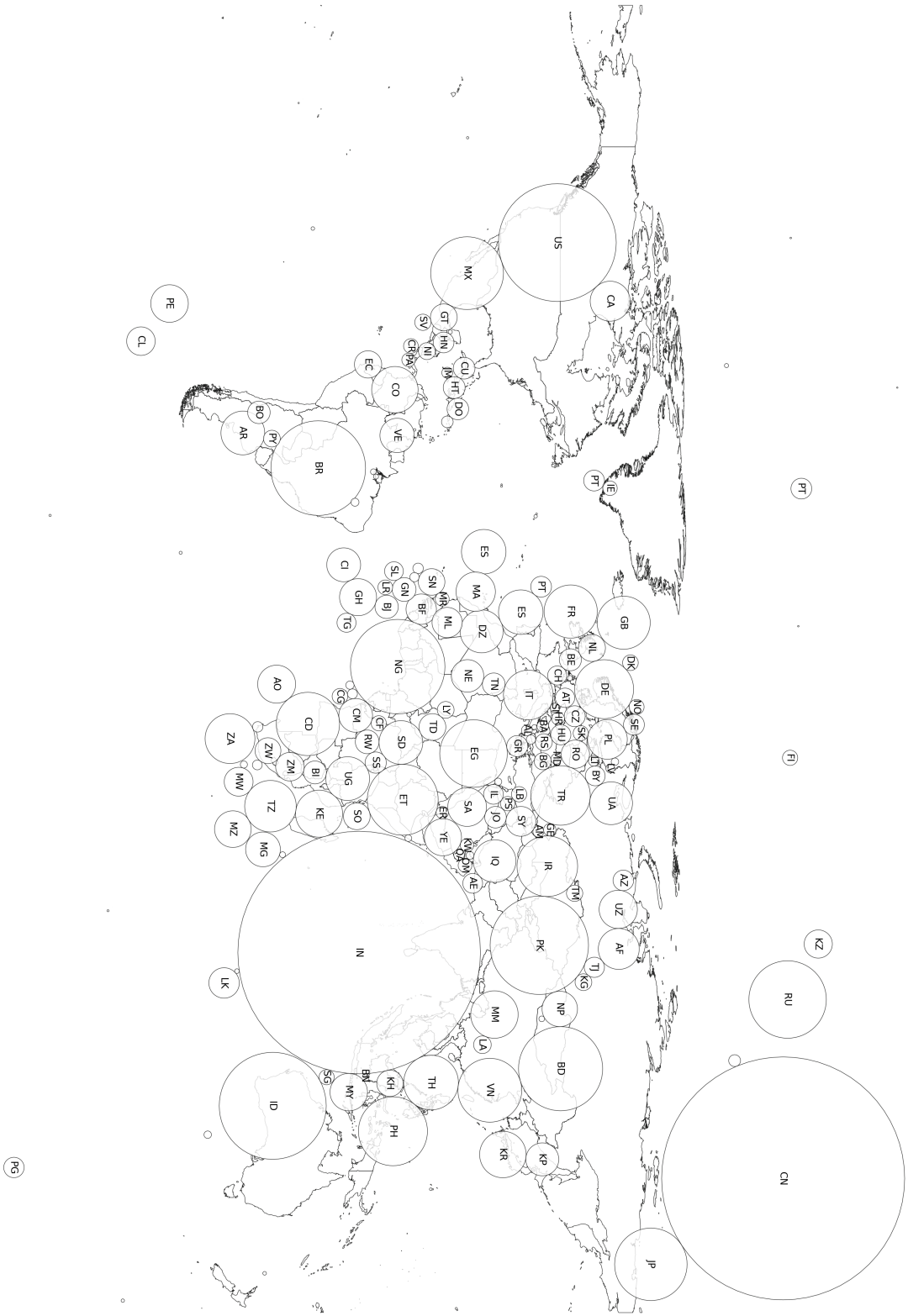
Full Resolution example of the Dorling method [7] applied to the Regions of England [28, 27]



Full Resolution example of the Dorling method [7] applied to the CUAs of England [28, 27]



Full Resolution example of the Dorling method [7] applied to the Countries of Europe [38, 9]



Full Resolution example of the Dorling method [7] applied to the Countries of the World [38, 9]