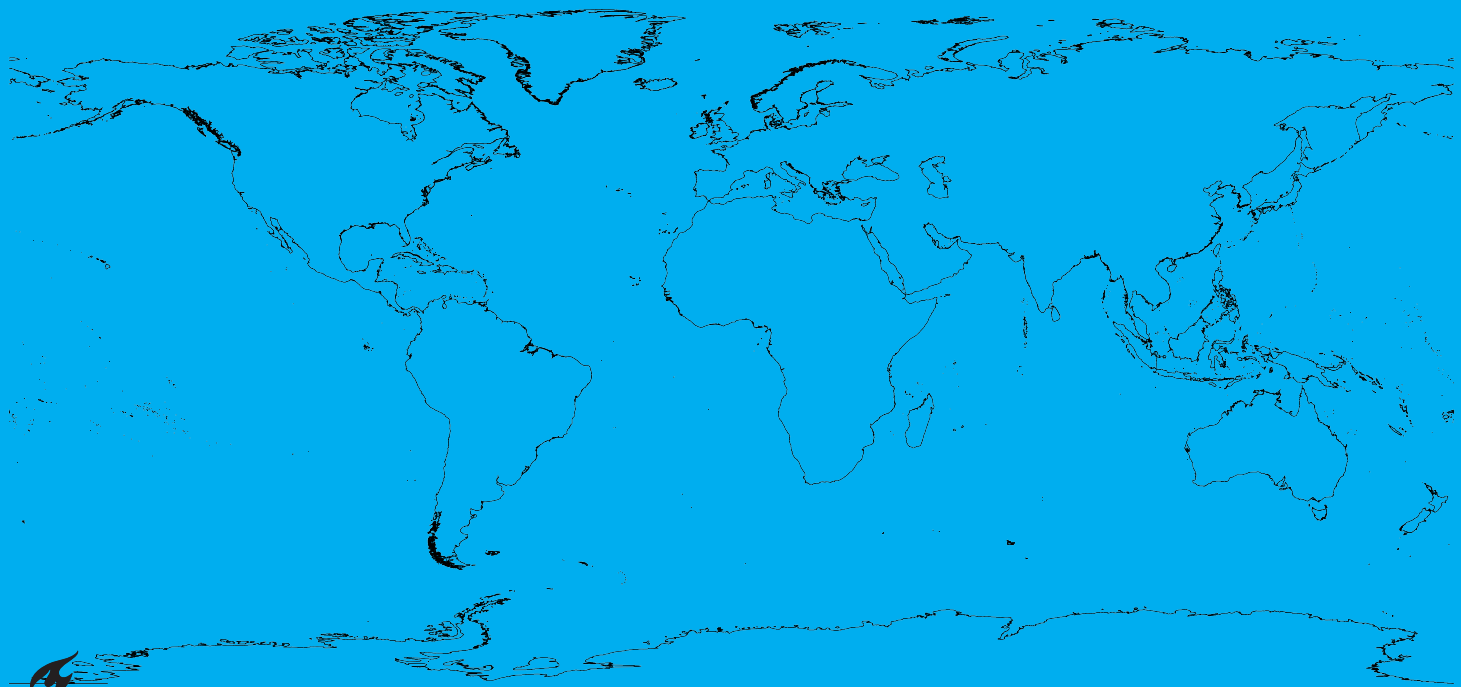
Refining Cartographic Distortion: A High-Precision Framework for Evaluating Global and Coastal Projections

Bachelor Thesis

Daan Boerlage



TU Delft

Refining Cartographic Distortion: A High-Precision Framework for Evaluating Global and Coastal Projections

Bachelor Thesis

by

Daan Boerlage

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended on Thursday July 10, 2025 at 2:00 PM.

Student number:	5871832	
Project duration:	March 13, 2025 – July 10, 2025	
Supervisors:	Dr. O. Jaïbi,	TU Delft
	Dr. P.M. Visser,	TU Delft
Thesis committee:	Dr.ir. R.F. Swarttouw,	TU Delft

This thesis is confidential and cannot be made public until July 10, 2025.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Layman's Abstract

No flat map can perfectly represent the Earth. Some shapes, sizes, or distances always end up a stretched or distorted. This study set out to improve how we measure these imperfections, using improved distortion numbers and more data, over 10 million points, to get accurate results. It builds on earlier work but replaces some of the old methods' biases to create a fairer, more precise method of scoring maps.

With this new method, the Winkel Tripel projection was identified as the overall best-performing projection for general-purpose use.

The research also explored whether the robustness of coastlines (measured with so called, fractal dimension) could help measure distortion. It turns out that this isn't a suitable method to compare maps globally, but it did lead to a new insight. For maps that are intended to accurately image coastlines as accurately as possible, like for navigation, the Azimuthal Equidistant projection was the best fit.

In short, this study gives map-makers better tools for picking the right kind of map, depending on what property they want to preserve, whether that is on the whole globe or just the coastlines.

Abstract

This research presents a comprehensive re-evaluation and improvement of map projection distortion analysis, building upon the foundational framework of Goldberg and Gott. The fundamental challenge in cartography lies in representing the Earth's spherical surface on a flat plane, called map projection, a process that inevitably introduces distortions in area, shape, or distance. This study aims to refine the quantification of these distortions by employing an improved distance metric that accurately measures projected geodesics, utilising high-precision numerical methods with over 10 million sample points to minimise error, and introducing an unbiased total distortion score that eliminates the arbitrary normalisation of previous methods. The results identify the Winkel Tripel projection as the best-performing overall map based on the improved, unbiased total distortion score.

Furthermore, this work critically investigates the viability of using the fractal dimension of coastlines as a novel distortion metric, leading to a shift in analysis towards evaluating distortions specifically along coastal geometries. However, theoretical computations using fractal dimension show that it is unsuitable as a global distortion metric due to its invariance under the bi-Lipschitz transformations that define many map projections. By establishing a normalisation approach based on coastline arc length, this study provides a robust and resolution-independent framework for tailored cartographic evaluation. The findings underscore the sensitivity of distortion scores to globe orientation and establish a foundational methodology for future property-specific distortion analyses, ensuring a more reliable and application-focused approach to selecting map projections than previously done. For the specific application of representing coastlines with minimal distortion, the Azimuthal Equidistant projection is found to be the most fitting.

Contents

List of Variables

Symbol	Description
$\mathbb{S}^2$	The 2-dimensional boundary 3-dimensional ball in $\mathbb{R}^3$.
$\mathbb{R}^n$	The n-dimensional Euclidean plane.
λ	Longitude, where $\lambda \in [-\pi, \pi)$.
ϕ	Latitude, where $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$.
R	Radius of the sphere ($R = 1$ for the unit sphere).
T	Map projection function, $T : \mathbb{S}^2 \to \mathbb{R}^2$
(x, y)	Planar coordinates from the projection: $(x, y) = T(\lambda, \phi) = (x(\lambda, \phi), y(\lambda, \phi))$.
(X, Y, Z)	Cartesian coordinates of a point on a sphere, or $\vec{r}(\lambda, \phi, R) = (X, Y, Z)$.
$x(\lambda, \phi)$	'x' part of projection function T.
$y(\lambda, \phi)$	'y' part of projection function T.
a, b	Semi-major and semi-minor axes of Tissot's indicatrices (singular values of the local transformation matrix).
T_M	Transformation matrix relating infinitesimal displacements on the sphere to the plane.
J	Jacobian matrix of the projection transformation.
$d\vec{r}$	Infinitesimal displacement vector in Cartesian coordinates.
ds	The length element on the unit sphere, thought of as the infinitesimal element of distance on the sphere.
dA	Infinitesimal area displacement.
A	Local area distortion.
I	Local isotropy (shape) distortion.
F	Local flexion distortion.
S	Local skewness distortion.
D	Local Distance distortion
B	Boundary cut distortion.
γ	A geodesic on the sphere.
$\vec{v}$	Velocity vector.
$\vec{a}$	Acceleration vector.
$\vec{a}_\perp$	Component of a perpendicular to the v.
$\vec{a}_\parallel$	Component of a parallel to v.
α	The angle of rotation to determine the flexion and skewness distortion.
H_x, H_y	Hessian matrices of the projection functions $x(\lambda, \phi)$ and $y(\lambda, \phi)$.
k	Scaling factor.
$\ell(\cdot)$	Arc length of a curve.
d	Great circle distance.
$\Delta\theta$	Central angle between two points on a sphere.
L_B	Length of the cut on the map
A_G, I_G, F_G, S_G, D_G	Global distortion scores for area, isotropy, flexion, skewness, and distance—typically RMS or mean over the globe.
T	Total distortion.

Symbol	Description
RMS	Root Mean Square
RMS^c	Root Mean Square along the coastline
$\langle\cdot\rangle$	Mathematical mean of a quantity.
$\|\cdot\|$	Length of vector in $\mathbb{R}^n$
δ	Fractal dimension of an object.
σ	Scaling factor (linear scale change).
μ	Mass-scaling factor (how "mass"—length, area, etc.—changes with scaling).
δ_{box}	Box-counting dimension (numerical estimator of fractal dimension).
N	Number of boxes of side-length ϵ needed to cover the shape.
ϵ	Side-length of the boxes used in the box-counting method.
$\hat{\epsilon}$	Subinterval length of dot-based box counting method
$\dim_H(S)$	Hausdorff dimension of a set S.
$c,,C$	Positive constants in the definition of a bi-Lipschitz transformation.
L	Total length of a coastline.
$L(\varepsilon)$	Measured length of a coastline using an instrument of size ε (illustrating Richardson's law).

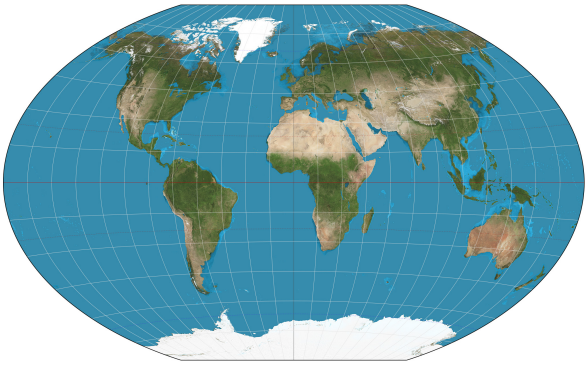
1

Introduction

The fundamental challenge of cartography: how to represent the spherical surface of the Earth on a plane. For millennia, map-makers, mathematicians, and navigators have grappled with this problem, as any attempt to unwrap a sphere onto a flat surface inevitably introduces distortions. This inherent geometric limitation, first rigorously proven by Carl Friedrich Gauss in his famous *Theorema Egregium*, establishes that no map projection can perfectly preserve all metric properties of the globe—namely area, shape, and distance—simultaneously [2]. Consequently, every map is a compromise, a careful balance of trade-offs designed to suit a specific purpose, whether for navigation, land administration, or thematic representation [17].

Ancient cartographers like Claudius Ptolemy in the 2^{nd} century AD developed some of the earliest systematic projections, see Figure 1.2, laying a foundation that would influence map-making for over a thousand years. The Age of Discovery dramatically amplified the need for accurate navigational charts, leading to the development of one of the most famous and influential projections in history: the Mercator projection of 1569, Figure 1.1a. By preserving angles locally, Mercator's map allowed sailors to plot courses as straight lines, a revolutionary tool for maritime navigation. However, this convenience came at the cost of extreme area distortion, famously exaggerating the size of landmasses near the poles, such as Greenland, see Figure 1.3. This trade-off exemplifies the core dilemma of map projections: optimising one property often requires sacrificing another. Over the centuries, hundreds of other projections have been developed, from the equal-area projections of Lambert and Gall-Peters to compromise projections like the Winkel Tripel (Figure 1.1b), which attempts to minimise all types of distortion rather than eliminating one completely.

(a) Image by Strebe, licensed under CC BY-SA 4.0 via Wikimedia Commons.



(b) Image by Strebe, released into the public domain (CC0) via Wikimedia Commons.

Figure 1.1: (a) Mercator Projection, (b) Winkel Tripel Projection

1.1. Quantifying Distortion: The Goldberg-Gott Framework

While the existence of distortion is a long-established fact, creating a systematic and objective method to quantify and compare it across different projections is a more recent endeavour. A significant modern ad-

Figure 1.2: Ptomely's map. Image by Francesco di Antonio del Chierico, licensed under CC BY-SA 3.0 via Wikimedia Commons.

vancement in this field came from David Goldberg and J. Richard Gott, who proposed a comprehensive framework for scoring map projections based on six distinct types of distortion [3]. Their work provided a robust methodology for moving beyond qualitative assessments to a rigorous, quantitative comparison of map quality. The six distortions they classified are:

1. **Area** measures how the projection stretches or shrinks the size of regions compared to their actual size on the globe. Equal-area projections eliminate this distortion entirely.

2. **Isotropy (Angle/Shape)** quantifies the distortion of shapes and angles. Conformal (or orthomorphic) projections, like the Mercator, preserve angles locally, resulting in no isotropy distortion.

3. **Flexion** measures the curvature or bending of what are originally defined as straight lines. On the globe, the shortest path between two points is called a geodesic, but this path often appears curved when projected onto a map.

4. **Skewness** relates to the change in scale along a geodesic. Even if a geodesic is projected as a straight line, the scale might not be constant along its length.

5. **Distance** captures errors in the measurement of distances between points. While some projections (Azimuthal Equidistant) can preserve true distances from a central point, no projection can preserve all distances correctly.

6. **Boundary Cut** accounts for the interruptions or cuts necessary to project the globe onto the desired map. These cuts create artificial boundaries, such as the eastern and western edges of a standard world map, separating points that are neighbours on the globe.

By measuring these six numbers across the entire surface of a map, Goldberg and Gott were able to assign a single total distortion score to any projection, providing a powerful tool for ranking and identifying the best possible map of the Earth.[4]

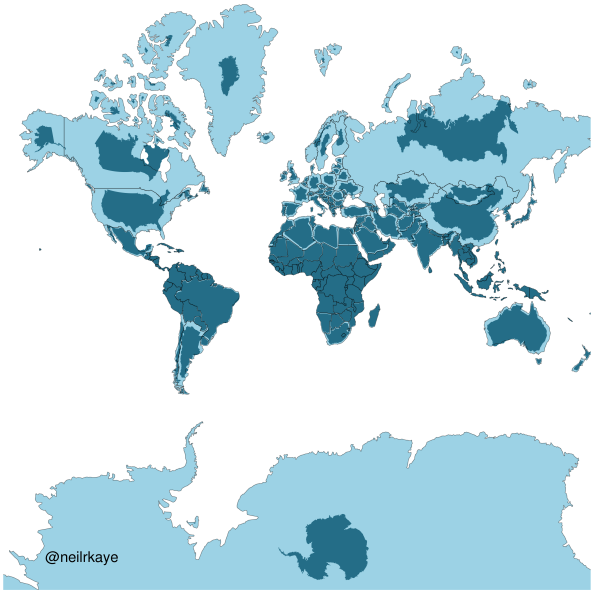


Figure 1.3: Area Distortion on the Mercator Projection

1.2. Goals and Contributions of This Study

This research builds directly upon the foundational work of Goldberg and Gott, aiming to refine their methodology, enhance the precision of their measurements, and extend the analysis to new domains. The primary goal of this study is to conduct a more accurate and unbiased evaluation of map projections, leveraging computational power and improved distortion metrics. The key contributions of this report are fourfold.

First, we employ an improved distance measure. The original definition of distance distortion can be ambiguous, particularly for points near a boundary cut. Our methodology utilises an improved calculation that correctly measures the path of the projected geodesic, providing a more faithful assessment of distance errors.

Second, we achieve a higher level of computational precision. By optimising the underlying code and sampling a vastly larger number of data points across the globe (over 10 million), we reduce numerical errors significantly, calculating distortion values with an accuracy of up to three decimal places—and even five for area and isotropy—far exceeding previous studies.

Third, we propose and implement a more objective total distortion score. The original Goldberg-Gott score was normalised to the distortions of the Equirectangular projection, an arbitrary choice that can bias the final rankings. Our approach removes this dependency, resulting in a more stable and unbiased evaluation framework.

Finally, we explore and critique the potential of a new distortion metric based on the fractal dimension of coastlines, a concept recently proposed in the literature [10]. While our theoretical analysis demonstrates that fractal dimension is invariant under most projections and thus unsuitable as a global distortion metric, this investigation leads to a crucial shift in focus: from analysing distortion across the entire globe to analysing it specifically along coastlines. Since coastlines are often the primary or sole feature of interest in many maps, evaluating distortions along these complex geometries offers a more practical and relevant measure of a map's utility for specific applications.

1.3. Overview of Chapters

This report is structured to guide the reader from the theoretical foundations of map projections to the presentation and discussion of our novel findings.

- **Chapter 2** provides a detailed mathematical definition of the six fundamental distortion measures, deriving the formulas used for their calculation and explaining the construction of a total distortion score.
- **Chapter 3** presents the results of our high-precision distortion analysis for a wide range of common

map projections. It compares the rankings generated by our unbiased scoring system with those from methods dependent on normalisation.

- **Chapter 4** introduces the concept of fractal geometry and investigates whether the fractal dimension of coastlines can serve as a seventh distortion metric. After demonstrating its theoretical limitations, the chapter pivots to justifying the analysis of the original distortion metrics specifically along coastlines.
- **Chapter 5** details the results of our coastline-specific distortion analysis, presenting findings at low, intermediate, and high resolutions to examine the impact of data granularity.
- **Chapter 6** discusses the broader implications of our findings, addressing limitations such as sensitivity to globe orientation and the Richardson effect, and identifies key opportunities for future research.
- **Chapter 7** concludes the report, summarising our primary contributions and identifying the best-performing projections for both general-purpose use and the specific task of coastline representation based on our enhanced methodology.
- **Chapter 8** serves as an appendix, containing supplementary figures, plots, and references to the computational code used in this study.

Through this comprehensive analysis, this study seeks to advance the science of cartographic evaluation, offering a more precise, reliable, and application-focused framework for choosing the best map projection for the task at hand.

2

Distortions in Map Projections

This chapter defines six distinct distortion measures. From the introduction it has become clear what defines a map projection, but a mathematical definition is essential before we can proceed to the distortions.

Definition 2.1 (Map Projection)**.** A map projection is a function that transforms points from a sphere ($\mathbb{S}^2$) onto a plane ($\mathbb{R}^2$). Formally, it is a map

$$T : \mathbb{S}^2 \to \mathbb{R}^2,$$

where $\mathbb{S}^2 \subset \mathbb{R}^3$ is the 2-dimensional sphere embedded in 3-dimensional Euclidean space.

To practically define such a transformation, we first describe points on the sphere using spherical coordinates:

- **Latitude** ϕ, where $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.
- **Longitude** λ, where $\lambda \in [-\pi, \pi)$.

These spherical coordinates are related to the 3-dimensional Cartesian coordinates (X, Y, Z) of a point on a sphere of radius R by the following equations:

$$\vec{r}(\lambda, \phi, R) = \begin{cases} X = R\cos(\phi)\cos(\lambda) \\ Y = R\cos(\phi)\sin(\lambda) \\ Z = R\sin(\phi) \end{cases} \tag{2.1}$$

where we take $R = 1$, because we will work on the unit sphere. A map projection can then be expressed as a function that maps the spherical coordinates directly to 2-dimensional planar coordinates (x, y):

$$(x, y) = T(\lambda, \phi)$$

For example, the Equirectangular projection is one of the simplest projections, defined by the linear mapping:

$$T(\lambda, \phi) = \big(x(\lambda, \phi), y(\lambda, \phi)\big) = (\lambda, \phi)$$

This projection directly uses the longitude as the x-coordinate and the latitude as the y-coordinate.

2.1. Area and Isotropy

Area distortion measures how the size of a region on the map differs from its actual size on the globe. Isotropy distortion, on the other hand, captures deviations in angles or shapes. As Gauss proved, no map projection can preserve both area and angle simultaneously [2]. However, it is possible to construct projections that preserve one of these properties individually. Such projections are known as equal-area (area-preserving) and conformal (angle- or shape-preserving) projections. These properties become more intuitive when visualised.

2.1.1. Visualising Area and Isotropy

In order to visualise the effect of different map projections onto a plane, the reference point is to project equal sized infinitesimal disks on the globe and mapping them with a function $T : \mathbb{S}^2 \to \mathbb{R}^2$ onto the plane. The area and angle of the projected circles, reveal which parts of the map are distorted the most or least. In Figure 2.1 the equal sized disks on the globe can be seen and in Figure 2.2 we see them after being mapped onto the plane. For example, in Figure 2.2a, each circle remains perfectly round, showing the absence of isotropy

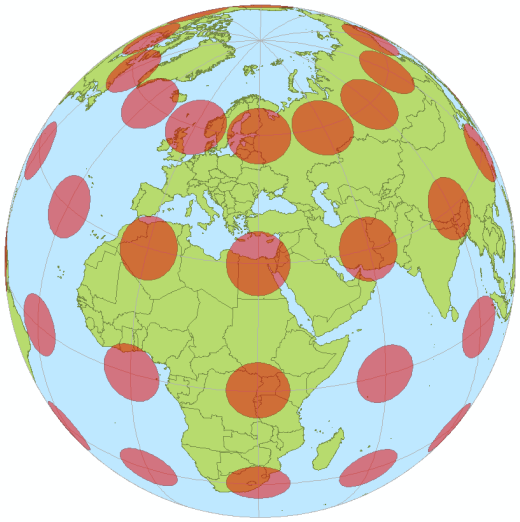
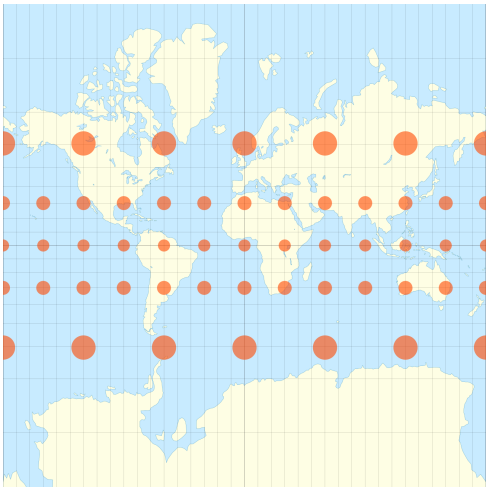
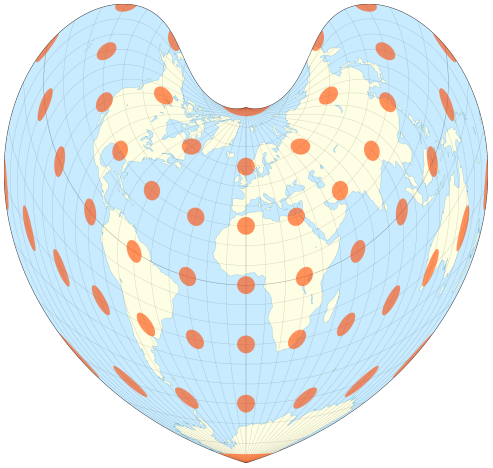


Figure 2.1: A collection of evenly sized disks on the globe. Image by Stefan Kühn, licensed under CC BY-SA 3.0 via Wikimedia Commons.

distortion. However, the disks differ in area, which indicates some distortion in the Mercator projection. A mathematically more interesting map is the Bonne projection, as seen in Figure 2.2b, which is neither conformal nor equal-area, but it tries to find some kind of balance.



(a) Image by Justin Kunimune, licensed under CC BY-SA 4.0 via Wikimedia Commons.



(b) Image by Justin Kunimune, released into the public domain (CC0) via Wikimedia Commons.

Figure 2.2: Visual comparison of distortion between the Mercator projection (a) and the Bonne projection (b).

2.1.2. Quantifying Area and Isotropy

A visual analysis can help identify which parts of the map cause the most distortion, but capturing the exact deviations requires a calculation of the distortion values. Consider a map projection $T : (\lambda, \phi) \to (x(\lambda, \phi), y(\lambda, \phi))$

with corresponding transformation matrix for infinitesimal displacement $(d\lambda, d\phi)$

$$T_M = \begin{bmatrix} \frac{\partial x}{\partial \lambda} \frac{1}{\cos(\phi)} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \lambda} \frac{1}{\cos(\phi)} & \frac{\partial y}{\partial \phi} \end{bmatrix}. \tag{2.2}$$

The singular values $a, b > 0$ of T_M measure locally the area

$$A = \ln(a \cdot b) \tag{2.3}$$

and isotropy

$$I = \ln\left(\frac{a}{b}\right). \tag{2.4}$$

We briefly derive the corresponding transformation matrix below. Let $d\vec{r}$ denote the infinitesimal displacement vector in Cartesian coordinates, with $\vec{r}$ from Equation 2.1,$R = 1$. Then,

$$d\vec{r} = \frac{\partial \vec{r}}{\partial \lambda} d\lambda + \frac{\partial \vec{r}}{\partial \phi} d\phi \tag{2.5}$$

The arc length element in spherical coordinates is equal to

$$ds^2 = \|d\vec{r}\|^2 = \cos^2\phi \, d\lambda^2 + d\phi^2, \tag{2.6}$$

so the infinitesimal displacement vector on the globe is

$$d\mathbf{s} = \begin{bmatrix} \cos\phi \, d\lambda \\ d\phi \end{bmatrix}. \tag{2.7}$$

Let $T(\lambda, \phi) = (x(\lambda, \phi), y(\lambda, \phi))$ denote the map projection. The Jacobian matrix of the projection is

$$J = \begin{bmatrix} \frac{\partial x}{\partial \lambda} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \lambda} & \frac{\partial y}{\partial \phi} \end{bmatrix}. \tag{2.8}$$

The distances on the sphere and the distances on the plane are related by the Jacobian matrix, or

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = J \begin{bmatrix} d\lambda \\ d\phi \end{bmatrix}. \tag{2.9}$$

To convert from spherical to Cartesian coordinates, we define the displacement vector

$$\begin{bmatrix} ds_\lambda \\ ds_\phi \end{bmatrix} = \begin{bmatrix} \cos\phi \cdot d\lambda \\ d\phi \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} d\lambda \\ d\phi \end{bmatrix} = \begin{bmatrix} \frac{1}{\cos\phi} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} ds_\lambda \\ ds_\phi \end{bmatrix}. \tag{2.10}$$

Tissot's Indicatrix relates how distances on the sphere change when mapped to a planer surface. So we want to find a transformation matrix that satisfies

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = T_M \begin{bmatrix} ds_\lambda \\ ds_\phi \end{bmatrix}. \tag{2.11}$$

By 2.9 and 2.10:

$$T_M = \begin{bmatrix} \frac{\partial x}{\partial \lambda} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \lambda} & \frac{\partial y}{\partial \phi} \end{bmatrix} \begin{bmatrix} \frac{1}{\cos\phi} & 0 \\ 0 & 1 \end{bmatrix}. \tag{2.12}$$

Thus, the full transformation matrix is as given in Equation 2.2.

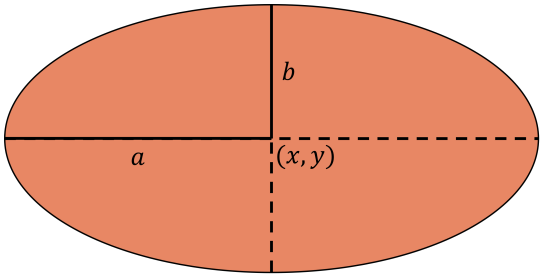


Figure 2.3: Tissot's indicatrices of an infinitesimal ellipse. A disk seen in Figure 2.2 corresponds with this ellipse.

While the use of singular value decomposition to determine a and b is a more recent method [8, 9], the original concept dates back to Tissot [18]. For this reason, a and b are commonly referred to as Tissot's indicatrices. These values correspond to the semi-axes of an infinitesimal ellipse. More precisely, a represents the semi-major axis and b the semi-minor axis, as illustrated in Figure 2.3. This ellipse corresponds to the ellipses from Figure 2.2.

Returning to the calculation, instead of following the whole process of singular value decomposition, we can use some handful expressions that simplify the calculation of the singular values [13]. From linear algebra, we have

$$a \cdot b = \sqrt{\det(T_M^\top T_M)} = \sqrt{\det(T_M)\det(T_M)} = |\det(T_M)|, \tag{2.13}$$

$$a^2 + b^2 = \text{Trace}(T_M^\top T_M), \tag{2.14}$$

Now, we can express a and b in terms of the trace and determinant. The first step is to substitute

$$b = \frac{|\det(T_M)|}{a}$$

into 2.14. An equation in terms of a follows, which we have to simplify

$$\begin{gathered} a^2 + \left(\frac{|\det(T_M)|}{a}\right)^2 = \text{Trace}(T_M^\top T_M) \iff \\ \frac{a^4 + (|\det(T_M)|)^2 - a^2 \cdot \text{Trace}(T_M^\top T_M)}{a^2} = 0 \implies \\ a^4 + (|\det(T_M)|)^2 - a^2 \cdot \text{Trace}(T_M^\top T_M) = 0. \end{gathered}$$

Let $t = a^2$,

$$\begin{gathered} t^2 + (|\det(T_M)|)^2 - t \cdot \text{Trace}(T_M^\top T_M) = 0. \implies \\ t = \frac{\text{Trace}(T_M^\top T_M) \pm \sqrt{\text{Trace}(T_M^\top T_M)^2 - |2 \cdot \det(T_M)|^2}}{2} \implies \\ a^2 = \frac{\text{Trace}(T_M^\top T_M) \pm \sqrt{\text{Trace}(T_M^\top T_M)^2 - |2 \cdot \det(T_M)|^2}}{2} \implies \\ a = \pm\sqrt{\frac{\text{Trace}(T_M^\top T_M) \pm \sqrt{\text{Trace}(T_M^\top T_M)^2 - |2 \cdot \det(T_M)|^2}}{2}}. \end{gathered}$$

Since $a > 0$, we will have

$$a = \sqrt{\frac{\text{Trace}(T_M^\top T_M) + \sqrt{\text{Trace}(T_M^\top T_M)^2 - |2 \cdot \det(T_M)|^2}}{2}} \implies$$
$$a = \frac{\sqrt{\text{Trace}(T_M^\top T_M) + \sqrt{\text{Trace}(T_M^\top T_M)^2 - |2 \cdot \det(T_M)|^2}}}{\sqrt{2}} \implies$$
$$a = \frac{\frac{1}{\sqrt{2}}\sqrt{c^2 + 2cd + d^2}}{\sqrt{2}},$$

where

$$c = \sqrt{\text{Trace}(T_M^\top T_M) + 2 \cdot |\det(T_M)|},$$
$$d = \sqrt{\text{Trace}(T_M^\top T_M) - 2 \cdot |\det(T_M)|}.$$

Now

$$a = \frac{\sqrt{c^2 + 2cd + d^2}}{\sqrt{2}\sqrt{2}} \implies$$
$$a = \frac{\sqrt{(c+d)^2}}{2} \implies$$
$$a = \frac{c+d}{2}.$$

When applying the same method for b, the following final expressions are found

$$a = \frac{\sqrt{\text{Trace}(T_M^\top T_M) + 2|\det(T_M)|} + \sqrt{\text{Trace}(T_M^\top T_M) - 2|\det(T_M)|}}{2}, \tag{2.15}$$
$$b = \frac{\sqrt{\text{Trace}(T_M^\top T_M) + 2|\det(T_M)|} - \sqrt{\text{Trace}(T_M^\top T_M) - 2|\det(T_M)|}}{2}. \tag{2.16}$$

2.1.3. Worked Example: Mercator Projection

Let us demonstrate this method using the Mercator projection from Figure 2.2a. The transformation formula $T : \mathbb{S}^2 \to \mathbb{R}^2$ is given by:

$$T(\lambda, \phi) = (\lambda, \ln(\tan\frac{\pi}{4} + \frac{\phi}{2})). \tag{2.17}$$

Then, the matrix T_M becomes:

$$T_M = \begin{bmatrix} \frac{1}{\cos\phi} & 0 \\ 0 & \frac{1}{\cos\phi} \end{bmatrix}.$$

Since a and b are determined locally, we now choose a point on the globe to evaluate, say $(0, \frac{\pi}{3})$. We find $a = 2$ and $b = 2$, which gives $A = \ln(4) \approx 1.39$ and $I = \ln(1) = 0$. In, for example, point $(0,0)$, we find both A and I to be 0. Sampling across the whole globe will give an image like Figure 2.4 for the area distortion. The isotropy heat map is not given, because $I = 0$ on the whole domain. We can do the same evaluation for the Bonne Projection, see Figure 2.5. For more analysed map projections, see Appendix A

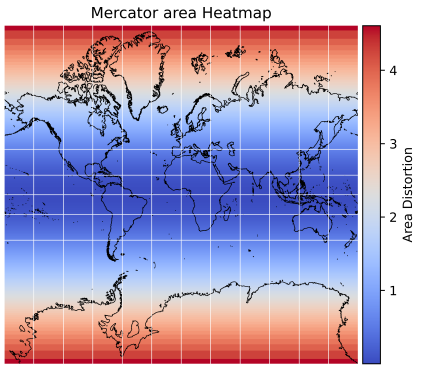
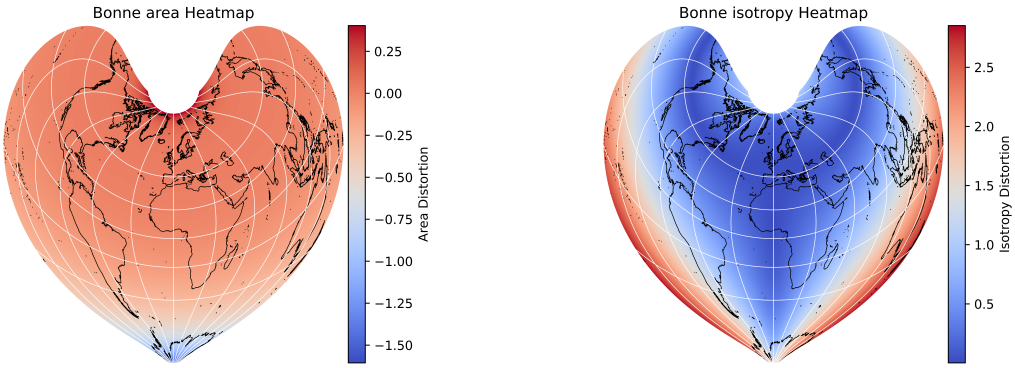


Figure 2.4: Heat map of area distortion on the Mercator projection. Red indicates high distortion in the respective region. Therefore, the most area distortion is obtained near the poles. Isotropy was left out for the Mercator projection, because it is equal to 0 across the whole map. Figures made with Appendix E code *heatmapping.py*.



(a) (b)

Figure 2.5: Heat maps of area (a) and isotropy (b) distortion on the Bonne projection. The Bonne projection shows most distortion near the boundary of its heart-shaped projected image.

2.2. Flexion and Skewness

Unlike area and isotropy, the flexion and skewness relates to the curvature of the Earth, and the embedding of the latter onto different map projections. A common notion from differential geometry is the concept of geodesic. In the context of the unit sphere embedded in $\mathbb{R}^3$ (or spheroid), the definition is given as follows.

Definition 2.2 (Geodesic)**.** A geodesic is defined as the path of shortest distance between any two points on the surface of a spheroid [6]. In general, the geodesics on the sphere are great circles: intersections of the sphere with the planes through the origin. A geodesic from point p_i to point p_j, $\gamma_{(i,j)}$, can be parametrised by:

$$\gamma_{(i,j)}(s) = \mathbf{p_i}\cos s + \mathbf{b_j}\sin s \tag{2.18}$$

where

- $\mathbf{p_i}$: the starting point in Cartesian coordinates;
- $\mathbf{b_j} = \dfrac{\mathbf{p_j} - (\mathbf{p_i}\cdot\mathbf{p_j})\mathbf{p_i}}{\left\|\mathbf{p_j} - (\mathbf{p_i}\cdot\mathbf{p_j})\mathbf{p_i}\right\|}$: the unit vector orthogonal to $\mathbf{p_i}$ pointing in the direction of $\mathbf{p_j}$;
- $s \in [0,\theta]$, with $\theta = \arccos(\mathbf{p_i}\cdot\mathbf{p_j})$: the spherical distance (central angle) between the two points $\mathbf{p_i}$ and $\mathbf{p_j}$. The parameter s represents the arc length along the geodesic from $\mathbf{p_i}$ to $\mathbf{p_j}$.

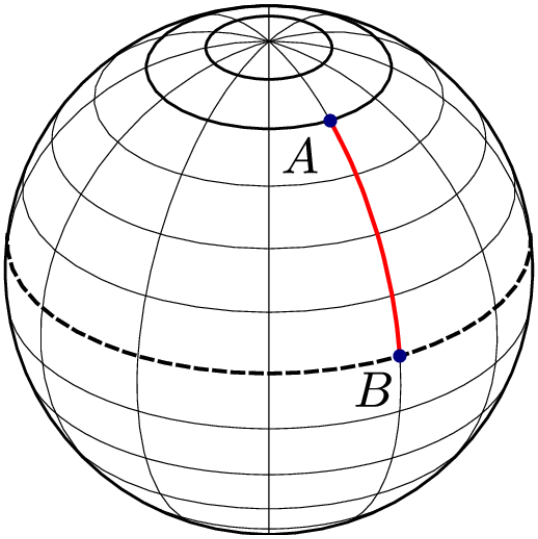


Figure 2.6: : A visual representation of a geodesic on a sphere, with the red line representing the geodesic from point A to point B [14].

The flexion (F) distortion measures the change in direction of a geodesic mapped from the globe onto the plane and the skewness (S) measures the change of velocity across this geodesic. The best way to visualise this, is to imagine yourself walking on the globe in a perfect straight line with constant speed. Now, if you would walk the same path on the map and it would still be a straight line, then the flexion in that direction is equal to zero. If you would walk the path with the same speed, then there is no skewness distortion.

What may have become clear from this visualisation is that both the velocity and the acceleration vectors are required to compute the flexion and skewness. Goldberg and Gott were the first to derive expressions for these quantities. However, this research employs novel and simplified methods [7, 13]. We begin by introducing the mathematical definitions of flexion and skewness, followed by an explanation of how each component is determined. Flexion is defined as

$$F = \frac{1}{2\pi}\int_0^{2\pi} \frac{\|\vec{a}_\perp\|}{\|\vec{v}\|}\, d\alpha \tag{2.19}$$

and skewness as

$$S = \frac{1}{2\pi}\int_0^{2\pi} \frac{\|\vec{a}_\parallel\|}{\|\vec{v}\|}\, d\alpha, \tag{2.20}$$

The terms are defined as follows:

- $\vec{v}$: the velocity vector tangent to the geodesic;
- $\vec{a}_\parallel = \dfrac{\vec{a}\cdot\vec{v}}{\vec{v}\cdot\vec{v}}$;
- $\vec{a}_\perp = \vec{a} - \vec{a}_\parallel$;
- α: the angle representing the direction of the geodesic.

Notice how the direction of acceleration reflects the difference between flexion and skewness. This direction is determined by the geodesic, but we want to calculate the distortions locally. Therefore, we integrate over an angle α in the equation, representing the direction of the geodesic. Now, $\vec{a}_\perp$ is the acceleration orthogonal to $\vec{v}$ and $\vec{a}_\parallel$ the acceleration parallel to $\vec{v}$. Flexion and skewness can also be expressed directly in terms of the velocity $\vec{v}$, its perpendicular direction $\vec{v}_\perp$, and the acceleration $\vec{a}$, as follows

$$F = \frac{1}{2\pi}\int_0^{2\pi} \frac{|\vec{a}\cdot\vec{v}_\perp|}{\|\vec{v}\|^2}\, d\alpha, \tag{2.21}$$

$$S = \frac{1}{2\pi}\int_0^{2\pi} \frac{|\vec{a}\cdot\vec{v}|}{\|\vec{v}\|^2}\, d\alpha. \tag{2.22}$$

Finally, we express the velocity $\vec{v}$ and acceleration $\vec{a}$ vectors as

$$\vec{v} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = Ju, \tag{2.23}$$

$$\vec{a} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = Jw + \begin{bmatrix} u^\top H_x u \\ u^\top H_y u \end{bmatrix}, \tag{2.24}$$

where

$$u = \begin{bmatrix} \cos\alpha \\ \cos\phi \\ \sin\alpha \end{bmatrix}, J = \begin{bmatrix} \frac{\partial x}{\partial \lambda} & \frac{\partial x}{\partial \phi} \\ \frac{\partial y}{\partial \lambda} & \frac{\partial y}{\partial \phi} \end{bmatrix}, H_x = \begin{bmatrix} \frac{\partial^2 x}{\partial \lambda^2} & \frac{\partial^2 x}{\partial \lambda \partial \phi} \\ \frac{\partial^2 x}{\partial \phi \partial \lambda} & \frac{\partial^2 x}{\partial \phi^2} \end{bmatrix}, H_y = \begin{bmatrix} \frac{\partial^2 y}{\partial \lambda^2} & \frac{\partial^2 y}{\partial \lambda \partial \phi} \\ \frac{\partial^2 y}{\partial \phi \partial \lambda} & \frac{\partial^2 y}{\partial \phi^2} \end{bmatrix}, w = \begin{bmatrix} \frac{\tan\phi \sin 2\alpha}{\cos\phi} \\ -\tan\phi \cos^2\alpha \end{bmatrix}.$$

Here, J is the Jacobian matrix of the map projection, and H_x and H_y are the Hessian matrices of x and y, respectively. The vectors u and w represent the direction and directional change of geodesic motion in geographic coordinates. Their forms are adopted from earlier derivations in [7, 13].

2.2.1. Worked Example: Mercator Projection

For consistency, we will work with the Mercator projection again. We start by computing the Jacobian and Hessian matrices.

$$J = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\cos\phi} \end{bmatrix}, H_x = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, H_y = \begin{bmatrix} 0 & 0 \\ 0 & \frac{\tan\phi}{\cos\phi} \end{bmatrix}$$

This allows for $\vec{v}$, $\vec{v}_\perp$ and $\vec{a}$.

$$\vec{v} = Ju = \begin{bmatrix} \frac{\cos\alpha}{\cos\phi} \\ \frac{\sin\alpha}{\cos\phi} \end{bmatrix}, \vec{v}_\perp = \begin{bmatrix} \frac{\sin\alpha}{\cos\phi} \\ -\frac{\cos\alpha}{\cos\phi} \end{bmatrix},$$

$$\vec{a} = Jw + \begin{bmatrix} u^\top H_x u \\ u^\top H_y u \end{bmatrix} = \begin{bmatrix} \frac{\tan\phi \sin^2\alpha}{\cos\phi} \\ -\frac{\tan\phi \cos^2\alpha}{\cos\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\tan\phi \sin^2\alpha}{\cos\phi} \end{bmatrix} = \begin{bmatrix} \frac{\tan\phi \sin^2\alpha}{\cos\phi} \\ -\frac{\tan\phi \cos 2\alpha}{\cos\phi} \end{bmatrix}.$$

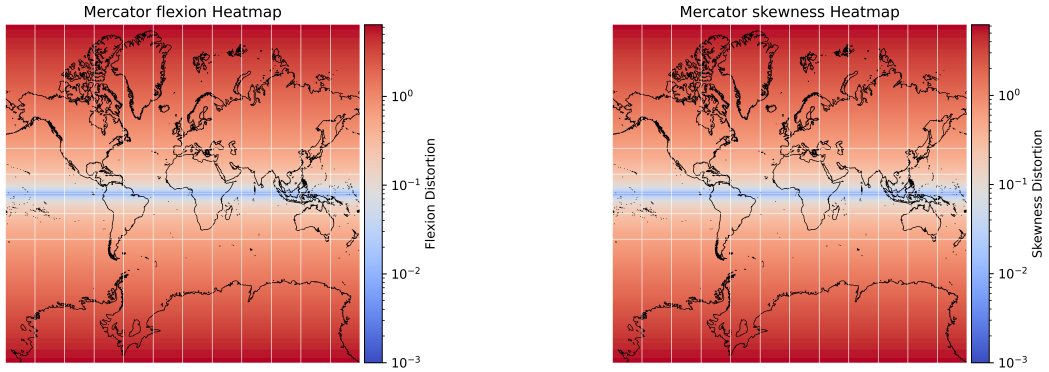
Having all the terms, we can substitute them into the definitions of flexion and skewness, resulting in:

$$\begin{aligned} F &= \frac{1}{2\pi} \int_0^{2\pi} \frac{|\vec{a} \cdot \vec{v}_\perp|}{\|\vec{v}\|^2} \, d\alpha \\ &= \frac{1}{2\pi} \int_0^{2\pi} \frac{|\cos\alpha \tan\phi| / \cos^2\phi}{1/\cos^2\phi} \, d\alpha \\ &= \frac{1}{2\pi} \int_0^{2\pi} |\cos\alpha \tan\phi| d\alpha \\ &= \frac{1}{2\pi} \cdot |\tan\phi| \int_0^{2\pi} |\cos\alpha| d\alpha \\ &= \frac{1}{2\pi} \cdot 4 \cdot |\tan\phi| \\ &= \frac{2}{\pi} \cdot |\tan\phi|. \end{aligned}$$

and

$$
\begin{aligned}
S &= \frac{1}{2\pi}\int_0^{2\pi} \frac{|\vec{a}\cdot\vec{v}|}{\|\vec{v}\|^2}\,d\alpha \\
&= \frac{1}{2\pi}\int_0^{2\pi} \frac{|\sin\alpha\tan\phi|/\cos^2\phi}{1/\cos^2\phi}\,d\alpha \\
&= \frac{1}{2\pi}\int_0^{2\pi} |\sin\alpha\tan\phi|\,d\alpha \\
&= \frac{1}{2\pi}\cdot|\tan\phi|\int_0^{2\pi} |\sin\alpha|\,d\alpha \\
&= \frac{1}{2\pi}\cdot 4\cdot|\tan\phi| \\
&= \frac{2}{\pi}\cdot|\tan\phi|.
\end{aligned}
$$

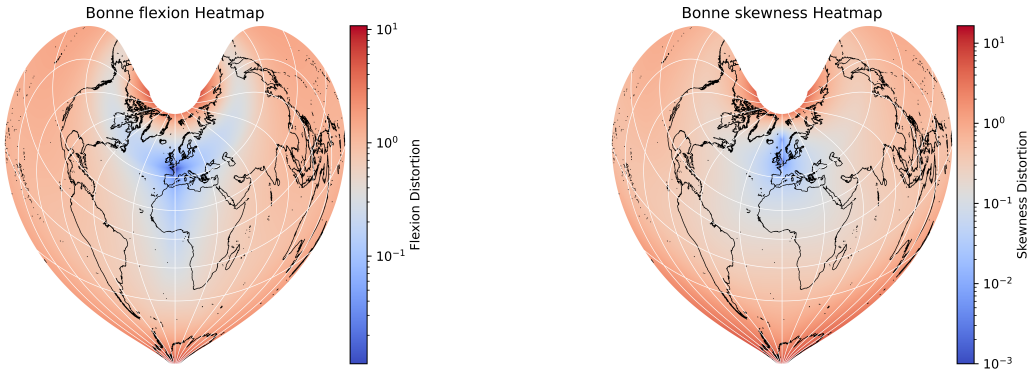
Observe how in the case of the Mercator projection, flexion and skewness are both equal and independent of the longitude (λ). By generating the same heat maps as in Section 2.1.3, the overall distortion patterns across the map become apparent, see Figure 2.7. For more analysed map projections, see Appendix A A visual



(a) (b)

Figure 2.7: Heat maps of flexion (a) and skewness (b) distortion on the Mercator projection. Figures were made with Appendix E code *heatmapping.py*.

comparison was also conducted for the Bonne projection, as shown in Figure 2.8. The heatmaps are very similar to the heatmaps of area and isotropy. That is, the distortion of the Mercator projection is symmetric with respect to the x-axis and both maps indicate the most distortion near the poles.



(a) (b)

Figure 2.8: Heat maps of flexion (a) and skewness (b) distortion on the Bonne projection.

2.3. Distance

A natural consequence of defining geodesics is using their original lengths on the sphere, and their length after projection, as a measure of distortion. To compute this distance distortion number, we define the great circle distance first as follows.

Definition 2.3 (great circle distance)**.** The great circle distance d is the length of the geodesic connecting two points (1 and 2) on the surface of a sphere. By the spherical coordinates:

$$\textbf{Point 1: } p_1 = (\cos\phi_1 \cos\lambda_1, \cos\phi_1 \sin\lambda_1, \sin\phi_1),$$

$$\textbf{Point 2: } p_2 = (\cos\phi_2 \cos\lambda_2, \cos\phi_2 \sin\lambda_2, \sin\phi_2)$$

The distance along the surface of the sphere between p_1 and p_2 is given by the central angle $\Delta\theta$ between them. This angle can be computed using the inner product of the corresponding unit vectors on the sphere:

$$\cos(\Delta\theta) = p_1 \cdot p_2 \tag{2.25}$$

$$\begin{aligned}\cos(\Delta\theta) &= \cos\phi_1 \cos\lambda_1 \cos\phi_2 \cos\lambda_2 + \\ &\quad \cos\phi_1 \sin\lambda_1 \cos\phi_2 \sin\lambda_2 + \\ &\quad \sin\phi_1 \sin\phi_2 \\ \cos\Delta\theta &= \sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 (\cos\lambda_1 \cos\lambda_2 + \sin\lambda_1 \sin\lambda_2) \\ \Delta\theta &= \arccos(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos\Delta\lambda). \end{aligned} \tag{2.26}$$

For general spheres of radius R, $d = R\Delta\theta$, but since we consider the unit sphere, $R = 1$, so the great circle distance (d) is $d = \Delta\theta$.

Originally Goldberg and Gott defined the distance function as

$$\ln\left(\frac{d_{ij,\text{map}}}{d_{ij,\text{globe}}}\right).$$

Here, $d_{ij,\text{map}}$ is the distance between points i and j projected onto the plane and $d_{ij,\text{globe}}$ is the great circle distance between points i and j. However, this definition does not take into account two crucial features: (i) the Earth is a sphere, so we can travel two ways (in the direction of the shortest path or the π radians in the other direction) and (ii) the geodesic between any two points on the surface of the globe is not necessarily a straight line on the map. For an illustration of these problems, see Figures 2.9 and 2.10. For more analysed map projections, see Appendix B

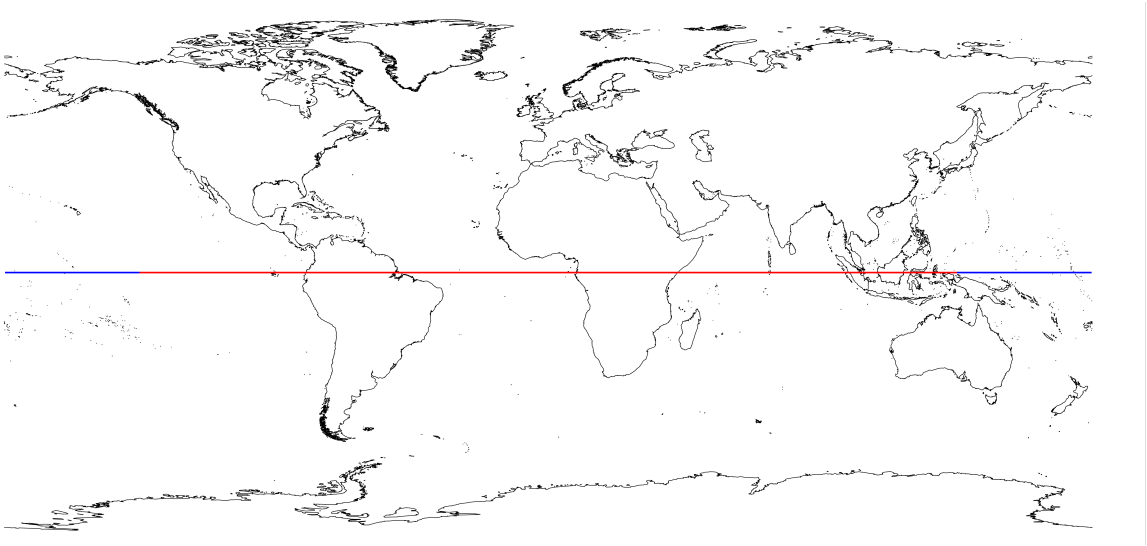


Figure 2.9: The Equirectangular projection with the distance measure defined by Goldberg and Gott (red) and what it should be (blue). Figure made with Appendix E code *distortions.py*.

To cover these two problems, we need to project the shortest path between the two points on the globe (the geodesic) onto the plane and measure the length of this mapped path. Basically, we need to measure the blue line instead of the red line from Figures 2.9 and 2.10. Mathematically this can be expressed as

$$D = \ln\left(\frac{\ell(T(\gamma_{i,j}))}{\ell(\gamma_{i,j})}\right), \tag{2.27}$$

where:

- $\gamma_{(i,j)}$: the geodesic between two points i and j on the unit sphere;
- $T(\gamma_{(i,j)})$: the projected image of the geodesic on the plane under the map projection T;
- $\ell(\cdot)$: arc length (i.e., the length of a curve).

It remains to be proven whether the improved distance measure is invariant under uniform scaling of the map. This property is desirable, as a lack of scale invariance would unjustly favour smaller maps over larger ones in distortion assessments.

Proof. We start by scaling both the map projection transformation functions by a constant factor $k > 0$.

$$\begin{cases} x(\lambda,\phi) \to \hat{x}(\lambda,\phi) = k \cdot x(\lambda,\phi) \\ y(\lambda,\phi) \to \hat{y}(\lambda,\phi) = k \cdot y(\lambda,\phi) \end{cases}$$

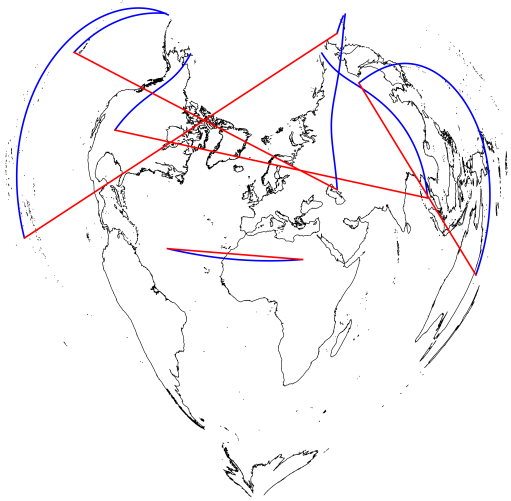
Take a geodesic $\gamma_{(i,j)}$ on the unit sphere from point (λ_i,ϕ_i) to point (λ_j,ϕ_j) By the definition of great-circle distance (Definition 2.3), the length of the scaled geodesic is

$$\ell(\gamma_{(i,j)}) = \Delta\theta = \arccos\left(\sin\phi_i \sin\phi_j + \cos\phi_i \cos\phi_j \cos\Delta\lambda\right).$$

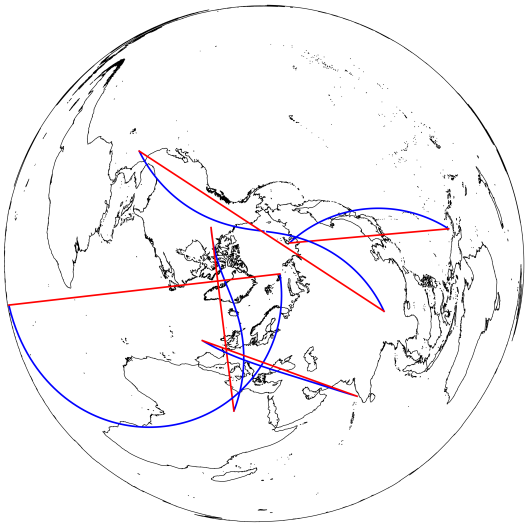
The length of the projected image of the geodesic on the plane under the map projection $T(\lambda,\phi) = (x(\lambda,\phi), y(\lambda,\phi))$ can be computed using the arc length formula. Let $\gamma_{(i,j)}(s) = (\lambda(s),\phi(s))$ parametrise the geodesic by arc length s on the sphere. Then the length of the projected geodesic is

$$\ell(T(\gamma_{(i,j)})) = \int_{s_1}^{s_2} \sqrt{\left(\frac{d}{ds}x(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}y(\lambda(s),\phi(s))\right)^2}\, ds.$$

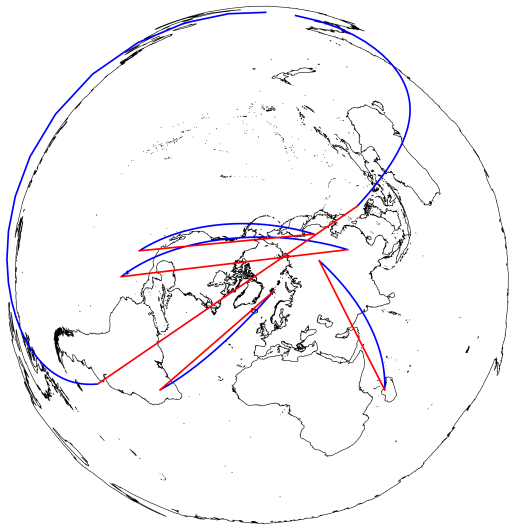
(a)



(b)



(c)



(d)

Figure 2.10: Different projections: Mercator (a), Bonne (b), Wiechel (c) and Azimuthal Euidistant (North Polar Aspect) (d), projecting the distance measure defined by Goldberg and Gott (red) and what it should be (blue). Five random paired points were sampled and their respective distances were plotted by Appendix E code *distortions.py*.

Subsequently, we take $\hat{T}(\lambda,\phi) = (\hat{x}(\lambda,\phi), \hat{y}(\lambda,\phi))$ to project the geodesic and compute its length:

$$\begin{aligned}
\ell(\hat{T}(\gamma_{(i,j)})) &= \int_{s_1}^{s_2} \sqrt{\left(\frac{d}{ds}\hat{x}(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}\hat{y}(\lambda(s),\phi(s))\right)^2}\, ds \\
&= \int_{s_1}^{s_2} \sqrt{\left(\frac{d}{ds}k\cdot x(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}k\cdot y(\lambda(s),\phi(s))\right)^2}\, ds \\
&= \int_{s_1}^{s_2} \sqrt{k^2\cdot\left(\frac{d}{ds}x(\lambda(s),\phi(s))\right)^2 + k^2\cdot\left(\frac{d}{ds}y(\lambda(s),\phi(s))\right)^2}\, ds \\
&= \int_{s_1}^{s_2} \sqrt{k^2\cdot\left[\left(\frac{d}{ds}x(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}y(\lambda(s),\phi(s))\right)^2\right]}\, ds \\
&= k\cdot\int_{s_1}^{s_2} \sqrt{\left(\frac{d}{ds}x(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}y(\lambda(s),\phi(s))\right)^2}\, ds \\
&= k\cdot\ell(T(\gamma_{(i,j)})).
\end{aligned}$$

Finally, we use the definition of the improved distance measure (Equation 2.27) to determine whether it is invariant under uniform scaling.

$$\hat{D} = \ln\left(\frac{\ell(\hat{T}(\gamma_{(i,j)}))}{\ell(\gamma_{(i,j)})}\right) = \ln\left(\frac{k\cdot\ell(T(\gamma_{(i,j)}))}{\ell(\gamma_{(i,j)})}\right) = \ln k + \ln\left(\frac{\ell(T(\gamma_{(i,j)}))}{\ell(\gamma_{(i,j)})}\right) = \ln k + D.$$

Hence, the measure is dependent on scaling, which is not desired, as it would unfairly penalise larger maps. A solution to this issue will be presented in Section 2.5. □

2.3.1. Worked Example: Distortion in the Equirectangular Projection

Let us show how this distance measure works in practice, but this time on the Equirectangular projection projection (the map from Figure 2.9). The transformation formula $T : \mathbb{S}^2 \to \mathbb{R}^2$ is given by

$$T(\lambda,\phi) = (x(\lambda,\phi), y(\lambda,\phi)) = (\lambda,\phi). \tag{2.28}$$

We first calculate the length of the projected geodesic ($\ell(T(\gamma_{i,j}))$). For that, a parametrisation $\gamma_{(i,j)}(s) = (\lambda(s),\phi(s))$ will be derived. Recall Definition 2.2, the parametrisation of a geodesic on the unit sphere from point $\mathbf{p_i}$ to point $\mathbf{p_j}$ in Cartesian coordinates is given by

$$\gamma_{(i,j)}(s) = \mathrm{p_i}\cos s + \mathrm{b}_j \sin s. \tag{2.29}$$

With

$$\mathrm{p_i} = (X,Y,Z) = (\cos(\phi_i)\cos(\lambda_i), \cos(\phi_i)\sin(\lambda_i), \sin(\phi_i)) \text{ and } \mathrm{p_j} = (\cos(\phi_j)\cos(\lambda_j), \cos(\phi_j)\sin(\lambda_j), \sin(\phi_j)).$$

Now, we choose two points to evaluate, for example: the north pole ($(\lambda_i,\phi_i) = (0, \frac{\pi}{2})$)

$$\mathrm{p_i} = (0,0,1)$$

and the point at $\frac{\pi}{4}$ latitude on the prime meridian ($(\lambda_j,\phi_j) = (0, \frac{\pi}{4})$)

$$\mathrm{p_j} = \left(\frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2}\right).$$

The angle between them is

$$\theta = \arccos(\mathrm{p_i}\cdot\mathrm{p_j}) = \arccos\left(\frac{\sqrt{2}}{2}\right) = \frac{\pi}{4}.$$

The vector $\mathbf{b}_j$ is

$$\mathrm{b}_j = \frac{\mathrm{p_j} - (\mathrm{p_i}\cdot\mathrm{p_j})\mathrm{p_i}}{\|\mathrm{p_j} - (\mathrm{p_i}\cdot\mathrm{p_j})\mathrm{p_i}\|} = \frac{\left(\frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2}\right) - \frac{\sqrt{2}}{2}(0,0,1)}{\left\|\left(\frac{\sqrt{2}}{2}, 0, 0\right)\right\|} = (1,0,0).$$

Therefore, the parametrisation is

$$\gamma_{(i,j)}(s) = \mathrm{p_i}\cos s + \mathrm{b}_j \sin s = (0,0,1)\cos s + (1,0,0)\sin s = (\sin s, 0, \cos s),$$

with

$$s \in [0, \pi/4].$$

Because our definition of $\ell(T(\gamma_{(i,j)}))$ requires the parametrisation to be in spherical coordinates, we need to convert $\gamma(s)$ back to spherical coordinates. From $\gamma(s)$, we identify

$$X(s) = \sin s, \quad Y(s) = 0, \quad Z(s) = \cos s.$$

Thus,

$$\phi(s) = \arcsin(z(s)) = \arcsin(\cos s),$$

and since $y(s) = 0$ and $x(s) > 0$ for $s \in (0, \pi/4]$,

$$\lambda(s) = 0.$$

Hence the spherical coordinate parametrisation is

$$\gamma_{(i,j)}(s) = (\lambda(s), \phi(s)) = (0, \arcsin(\cos s)), \quad s \in [0, \pi/4].$$

Now, we can compute its length when mapped by the Equirectangular projection:

$$\begin{aligned}
\ell(T(\gamma_{(i,j)})) &= \int_{s_1}^{s_2} \sqrt{\left(\frac{d}{ds}x(\lambda(s),\phi(s))\right)^2 + \left(\frac{d}{ds}y(\lambda(s),\phi(s))\right)^2}\, ds \\
&= \int_0^{\frac{\pi}{4}} \sqrt{\left(\frac{d}{ds}\lambda(s)\right)^2 + \left(\frac{d}{ds}\phi(s)\right)^2}\, ds \\
&= \int_0^{\frac{\pi}{4}} \sqrt{\left(\frac{d}{ds}\arcsin(\cos s))\right)^2}\, ds \\
&= \int_0^{\frac{\pi}{4}} \left|\frac{d}{ds}\arcsin(\cos s))\right| ds \\
&= \left|\int_0^{\frac{\pi}{4}} \frac{d}{ds}\arcsin(\cos s))\, ds\right| \\
&= \left|[\arcsin(\cos s))]_0^{\frac{\pi}{4}}\right| \\
&= \frac{\pi}{4}.
\end{aligned}$$

Next, we compute the great circle distance from p_i to p_j:

$$\ell(\gamma_{(i,j)}) = \arccos(\sin\phi_i \sin\phi_j + \cos\phi_i \cos\phi_j \cos\Delta\lambda) = \frac{\pi}{4}.$$

Finally,

$$D = \ln\left(\frac{\ell(T(\gamma_{(i,j)}))}{\ell(\gamma_{(i,j)})}\right) = \ln\left(\frac{\frac{\pi}{4}}{\frac{\pi}{4}}\right) = 0.$$

The distortion in this example is zero, because we took two points along the prime meridian. For almost any other pair of points, the distortion is not zero.

2.4. Boundary Cut

The boundary cut distortion measures the part of the boundary that needed to be cut in order to create the map and is given by:

$$B = \frac{\text{Length of the cut on the map}}{4\pi}. \tag{2.30}$$

The length of the cut on the map for most mappings are straightforward, because they belong to one of the following categories.

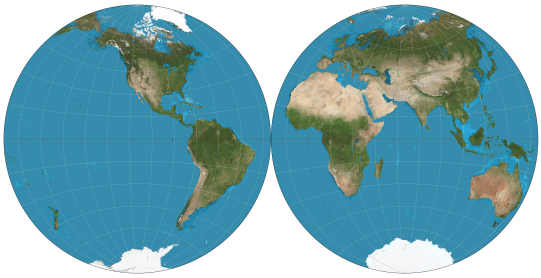


Figure 2.11: Nicolosi Globular Projection. An example of a projection split into two hemispheres. The boundary cut of such a projection equals $B = 0.5$. Image by Strebe, licensed under CC BY-SA 4.0 via Wikimedia Commons.

1. Non-interrupted projections, like the Azimuthal Equidistant from Figure 2.10d. Their boundary cut is as the name suggests $B = 0$.
2. Projections with a cut of length π on the globe, like the Mercator projection from Figure 2.2a or the Equirectangular projection from Figure 2.9. These are the most common map projections and have boundary cut $B = \frac{1}{4}$.
3. Projections with a cut of length 2π on the globe. These are the map projections that are split into two hemispheres, like the Nicolosi globular projection from Figure 2.11 and the boundary cut is $B = 1/2$.

However, there are a few maps, like the Boggs eumorphic, Fuller, Dymaxion, Waterman butterfly and HEALPix projection, that have multiple boundary cuts and are harder to determine.

2.5. From Local to Global Distortion

So far, distortions have only been defined locally. To compute a global distortion measure, one typically averages or takes the root mean square (RMS) of multiple local measurements. For the most precise global evaluation, this requires integrating the distortion function over the entire globe. However, due to the complexity of most map projection formulas, such integrals are generally evaluated numerically. Goldberg and Gott approached this by sampling random coordinates and computing either the mean or RMS of the distortion values, depending on the metric. This method is, in principle, equivalent to performing a numerical integration. The RMS is defined as follows.

Definition 2.4 (Root Mean Square (RMS))**.** The *Root Mean Square* is a measure of the magnitude of a varying quantity, defined as follows in different contexts:

- **Finite set of values:** For a discrete set of n values $x_1, x_2, \ldots, x_n$, the RMS is

$$\mathrm{RMS}(x_1, x_2, \ldots, x_n) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} = \sqrt{\langle x^2 \rangle}. \tag{2.31}$$

  with $\langle x^2 \rangle$ depicting the mathematical mean of x^2.

- **Continuous function over an interval:** For a continuous function $f(x)$ on the interval $[a, b]$, the RMS is

$$\mathrm{RMS}[f] = \sqrt{\frac{1}{b-a}\int_a^b f(x)^2\,dx}. \tag{2.32}$$

- **Function over the globe:** For a distortion measure $G(\lambda, \phi)$ (except for distance and boundary cut) defined over longitude $\lambda \in [-\pi, \pi)$ and latitude $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, the RMS over the sphere is

$$\mathrm{RMS}[G] = \sqrt{\frac{1}{4\pi}\int_{-\pi}^{\pi}\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} G(\lambda,\phi)^2 \cos\phi\,d\phi\,d\lambda}. \tag{2.33}$$

  Here, the factor $\cos(\phi)$ is the Jacobian determinant and accounts for the varying surface area at different latitudes in spherical coordinates.

In summary, the global distortion metrics become:

$$A_G = \text{RMS}\left(A_i - \langle A_i \rangle\right) = \text{RMS}(\ln(a_i b_i) - \langle \ln(a_i b_i) \rangle), \tag{2.34}$$

$$I_G = \text{RMS}\left(I_i\right) = \text{RMS}\left(\ln\left(\frac{a_i}{b_i}\right)\right), \tag{2.35}$$

$$F_G = \langle |F_i| \rangle = \left\langle \int_0^{2\pi} \frac{\|\vec{a}_{i\perp}|}{\|\vec{v}_i\|} d\alpha \right\rangle, \tag{2.36}$$

$$S_G = \langle |S_i| \rangle = \left\langle \int_0^{2\pi} \frac{|\vec{a}_{i\|} \cdot \hat{v}_i|}{\|\vec{v}_i\|} d\alpha \right\rangle, \tag{2.37}$$

$$D_G = \text{RMS}\left(D_{i,j} - \langle D_{i,j} \rangle\right) = \text{RMS}\left(\ln\left(\frac{\ell(T(\gamma_{i,j}))}{\ell(\gamma_{i,j})}\right) - \left\langle \ln\left(\frac{\ell(T(\gamma_{i,j}))}{\ell(\gamma_{i,j})}\right) \right\rangle\right), \tag{2.38}$$

$$B = \frac{L_B}{4\pi}. \tag{2.39}$$

Here $\langle \cdot \rangle$ denotes the mathematical mean. In these definitions, i denotes the position where each distortion is locally computed. In addition, the distance metric contains an index (i, j), which indicates the geodesic from point p_i to p_j. Notice how the global area and distance distortions include the extra terms $-\langle A_i \rangle$ and $-\langle D_{i,j} \rangle$. This was added to make the global measure invariant under scaling. The other distortions already satisfy this requirement [13].

2.6. Calculating a Total Distortion Score

To combine these distortions into one value, Goldberg and Gott weighted every distortion with normalisation constants. In the article the distortion values of the Equirectangular projection were chosen as weights. This gives the following equation:

$$\text{Total Distortion} = \left(\frac{A}{N_A}\right)^2 + \left(\frac{I}{N_I}\right)^2 + \left(\frac{F}{N_F}\right)^2 + \left(\frac{S}{N_S}\right)^2 + \left(\frac{D}{N_D}\right)^2 + \left(\frac{B}{N_B}\right)^2 \tag{2.40}$$

Here, N_A is the area distortion of the Equirectangular projection and similarly for the other normalisation constants. The choice to normalise with this projection seems rather obscure. It may therefore not be a reliable method to combine the distortions and ultimately pointing out the optimum map. In fact, one will see in Section 3.3, Table 3.2 how different mappings turn out to be preferable if we change the normalising constants. Consequently, just basing the optimum map on Equation 2.40 is not sufficient. We propose a better, unbiased, measure

$$\text{Total Distortion} = A^2 + I^2 + F^2 + S^2 + D^2 + B^2. \tag{2.41}$$

This will be used to determine the total distortion in the next chapters.

3

Map Analysis

In this chapter, the distortions defined in Section 2.5 are calculated for a range of existing map projections. The aim is to improve upon the measurements previously carried out by Goldberg and Gott, using the improved distance measure and evaluating more coordinates (λ_i, ϕ_i). These refinements may ultimately lead to the new optimal map projection.

3.1. Data Points

Most global distortions of map projections are difficult to determine analytically. As a result, the global distortion values were computed numerically, which requires sampling data points (λ, ϕ) over the sphere. The longitude λ is sampled uniformly over the interval $[-\pi, \pi)$. However, latitude ϕ can not be sampled uniformly over $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Recall the expression for the differential area element in spherical coordinates:

$$dA = dX \cdot dY = \cos\phi \cdot d\lambda \cdot d\phi.$$

This shows that sampling uniformly in ϕ would result in a higher density of points near the poles on the map. To achieve a uniform distribution of points, one must account for the $\cos\phi$ term.

Using a slightly informal notation,

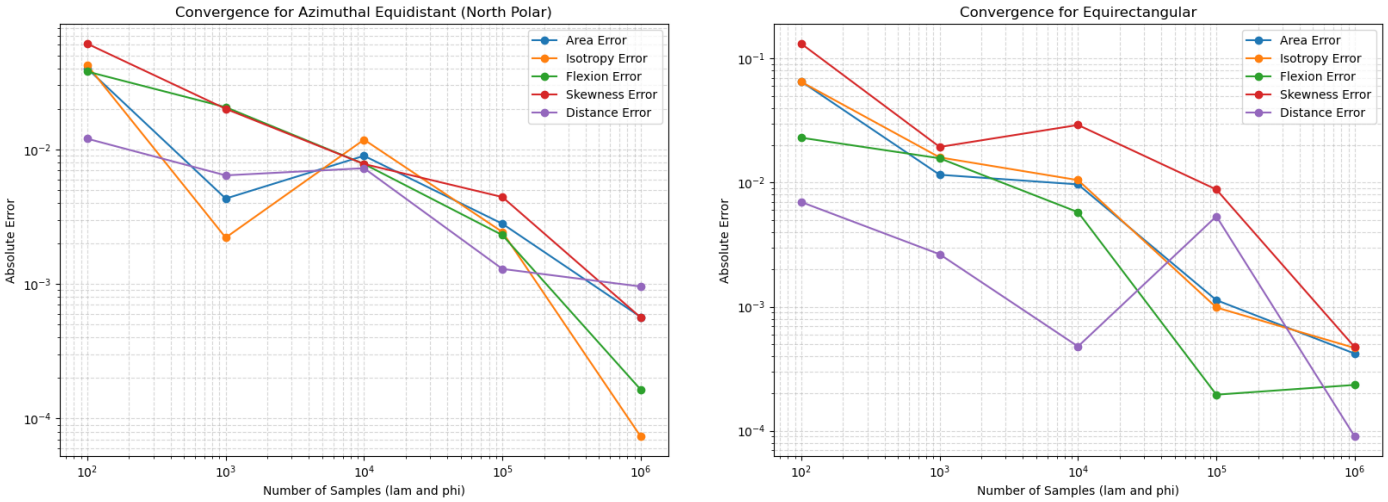
$$dA = \cos\phi \cdot d\phi \cdot d\lambda = d(\sin\phi) \cdot d\lambda.$$

This suggests a simple sampling strategy: instead of sampling ϕ directly, sample a uniform variable $u \in [-1, 1]$ and set $\phi = \arcsin(u)$. This ensures that the sampled points are uniformly distributed over the surface of the globe.

3.2. Error Analysis

When using numerical methods, it is important to be aware of potential deviations from the true solution. To address this, an error analysis was conducted prior to calculating the distortions.

In general, increasing the number of samples improves accuracy but also demands more computational power. Goldberg and Gott evaluated 30,000 samples, which was sufficient to reduce the error to two signifant digits. By optimising the code and switching from Python to the faster programming language C, the calculation time of 10,000,000 samples was brought to less than a minute. Thereby, creating the possibility of reducing the error even further to three significant digits. As shown in Figure 3.1a, using approximately 10^6 data points was sufficient to achieve the desired accuracy. Some projections required more data points, while others required fewer. For the error analysis of the remaining projections, see Appendix C.



(a) (b)

Figure 3.1: Error Analysis of the Area, Isotropy, Flexion, Skewness and Distance Distortions for the Azimuthal Equidistant (North Polar Aspect) projection (a) and Equirectangular Projection (b). On the x-axis the number of samples are presented, and on the y-axis the error is depicted (both logarithmically scaled). Figures made with Appendix E code *plot_error.py*.

Of all the different distortions we researched, flexion and skewness took the longest to compute, because every coordinate need to be evaluated in multiple directions/angles α. Likewise, the distance measure is computationally intensive, because between each coordinate pair $(\lambda_i, \phi_i), (\lambda_j, \phi_j)$ we need to interpolate along the geodesic. Now, if we leave the interpolation out and only look at the area and isotropy distortions, the following error can be attained:

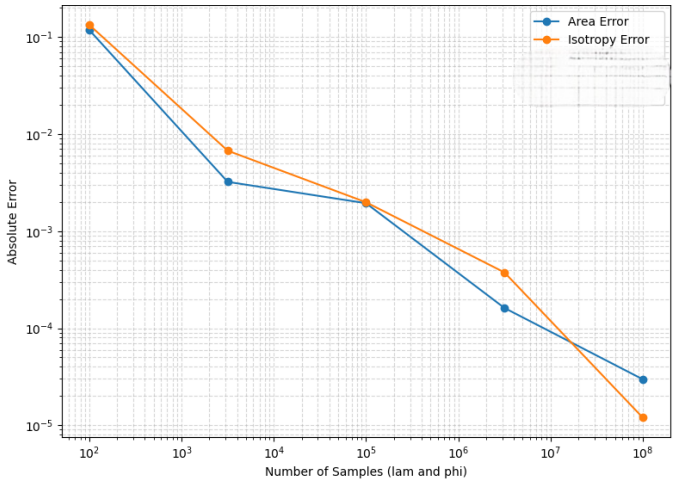


Figure 3.2: Error Analysis of only Area and Isotropy Distortion for the Equirectangular Projection. On the x-axis the number of samples are presented, and on the y-axis the error is depicted (both logarithmically scaled). Figure made with Appendix E code *plot_error.py*.

Our computations of the area and isotropy can be done up to the fifth significant digit. However, this analysis was only performed for illustration of the potential accuracy. The code is available for those who are interested in calculating distortions with this precision.

3.3. Distortion Results

With the precision of our distortion calculations now established, the following results emerge, where T represents the total distortion value defined in Equation 2.41, calculated by Appendix E code *c_distortions.py*.

Projection	A_G	I_G	F_G	S_G	D_G	B	T
Mercator	0.843	0	0.636	0.634	0.673	0.25	2.033
Equirectangular	0.421	0.521	0.636	0.631	0.686	0.25	1.784
Lambert Cylindrical Equal-Area	0	1.037	0.635	0.81	0.691	0.25	2.675
Behrmann	0	0.903	0.696	0.768	0.646	0.25	2.370
Gall-Peters	0	0.847	0.77	0.737	0.587	0.25	2.261
Sinusoidal	0	0.943	0.856	0.708	0.560	0.25	2.499
Hammer	0	0.823	0.829	0.481	0.529	0.25	1.939
Aitoff	0.129	0.702	0.788	0.421	0.547	0.25	1.669
Winkel Tripel	0.23	0.494	0.741	0.375	0.559	0.25	1.362
Stereographic (North Polar)	2	0	1	1	1.110	0	7.233
Azimuthal Equidistant (North Polar)	0.596	0.866	1	0.583	0.488	0	2.683
Lambert Azimuthal Equal-Area (North Polar)	0	1.414	1	0.644	0.427	0	3.597
Miller Cylindrical	0.624	0.261	0.638	0.632	0.667	0.25	1.772
Cassini	0.42	0.52	0.636	0.631	0.671	0.25	1.762
Central Cylindrical	1.264	0.521	0.676	1.171	0.944	0.25	4.651
Collignon	0	1.469	0.695	0.979	0.706	0.25	4.160
Lambert Conic	1.562	0	0.814	0.807	0.904	0.25	4.633
Bonne	0.252	1.246	0.997	0.792	0.467	0.25	3.518
Wiechel	0	1.533	1.231	0.751	0.451	0	4.633
Winkel II	0.29	0.549	0.724	0.513	0.608	0.25	1.605
Equidistant Conic	0.595	0.607	0.829	0.589	0.585	0.25	2,161
Spilhaus Stereographic	2	0	1	1	1.113	0	7.239

Table 3.1: Distortion values for multiple projections, with A_G, I_G, F_G, S_G, D_G and B the global distortion symbols for area, isotropy, flexion, skewness, distance and boundary cut respectively

3.4. Effect of Normalisation on Distortion Rankings

As already mentioned in Section 2.6, different mappings come out on top if the normalising constants are varied in Equation 2.40. The following table demonstrates this.

Projection	**Normalised to:**								
	Equi.	Aitoff	Winkel	Miller	Cassini	Central	Bonne	Winkel II	Equi. Conic
Mercator	7.983	48.139	19.476	5.843	8.045	3.131	15.315	12.975	5.964
Equirectangular	6.000	16.671	10.532	8.486	6.053	3.814	7.162	7.564	5.192
Lambert Cylindrical Equal-Area	8.623	9.131	12.333	20.492	8.683	6.858	5.334	9.123	8.974
Behrmann	7.571	8.158	10.751	16.574	7.622	5.962	4.866	8.000	7.532
Gall-Peters	7.207	7.629	9.984	15.123	7.250	5.723	4.506	7.509	6.911
Sinusoidal	8.013	7.861	10.544	17.813	8.055	6.597	4.546	8.101	7.715
Hammer	6.372	5.724	7.568	13.840	6.408	5.482	3.781	6.196	5.538
Aitoff	5.526	6.000	6.681	10.918	5.562	4.649	3.858	5.500	4.567
Winkel Tripel	4.574	7.398	6.000	7.122	4.609	3.587	4.202	4.868	3.475
Stereographic	30.176	251.745	88.486	18.002	30.399	6.804	71.241	56.607	30.444
Azimuthal Equidistant	8.600	27.193	14.788	15.764	8.643	5.689	8.717	10.556	8.165
Lambert Azimuthal Equal-Area	11.268	8.618	13.547	33.256	11.314	10.062	3.793	10.612	12.588
Miller Cylindrical	6.405	28.935	13.644	6.000	6.458	3.176	10.263	9.355	5.043
Cassini	5.949	16.552	10.448	8.424	6.000	3.787	7.057	7.497	5.112
Central Cylindrical	17.485	109.014	45.746	15.646	17.616	6.000	33.067	29.393	17.270
Collignon	13.611	13.229	19.129	37.383	13.688	11.265	6.687	14.071	15.785
Lambert Conic	19.778	155.090	55.570	12.359	19.921	5.368	44.871	35.961	16.719
Bonne	11.575	13.836	15.530	28.456	11.619	9.637	6.000	11.776	11.872
Wiechel	14.254	11.072	17.051	40.091	14.306	12.614	4.871	13.382	16.818
Winkel II	5.328	10.230	7.831	8.417	5.369	3.917	5.160	6.000	4.390
Equidistant Conic	7.653	27.219	14.011	10.647	7.700	4.721	9.623	9.986	6.000
Spilhaus Stereographic	30.187	251.764	88.503	18.015	30.412	6.810	71.267	56.622	30.477

Table 3.2: Total distortion values normalised to different projections using Equation 2.40 to depict its bias ness.

4

Fractals And Coastline Distortions

This chapter explores the potential of introducing the fractal dimension of coastlines as a seventh distortion measure. In addition, it examines the motivation for assessing distortions specifically along coastlines of the continents on Earth, as well as the anticipated impact of fractal dimensionality on these distortion metrics. To provide the necessary foundation, the concept of fractal dimension will first be introduced.

4.1. Fractal Dimension Basics

A line is one-dimensional, a square is two-dimensional, and a cube is three-dimensional. These are the most familiar examples when we think about dimensions. However, fractal geometry shows that not all shapes have whole-number dimensions. This idea stems from how dimension is defined, particularly in terms of how an object scales. Consider the following examples to illustrate this concept.

1. A line (with length 1) can be broken up into two smaller lines. Each of which is a perfect copy of the original, scaled down by a half.

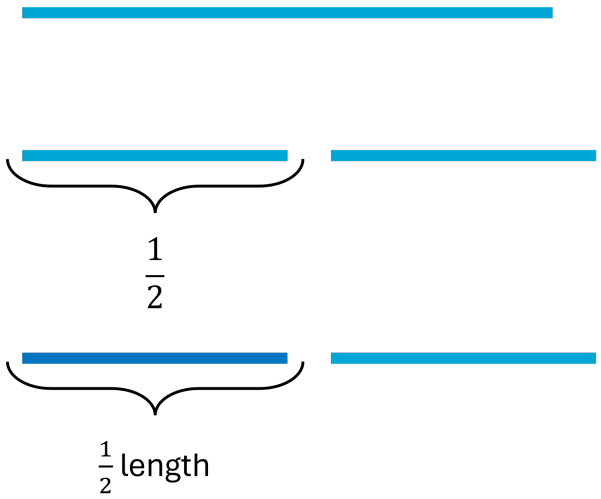


Figure 4.1: Scaling Factor $\frac{1}{2}$ and Mass Scaling Factor $\frac{1}{2}$ for a Line.

2. A square (with width 1) can be broken up into four smaller squares. Each of which is a perfect copy of the original, scaled down by a half.

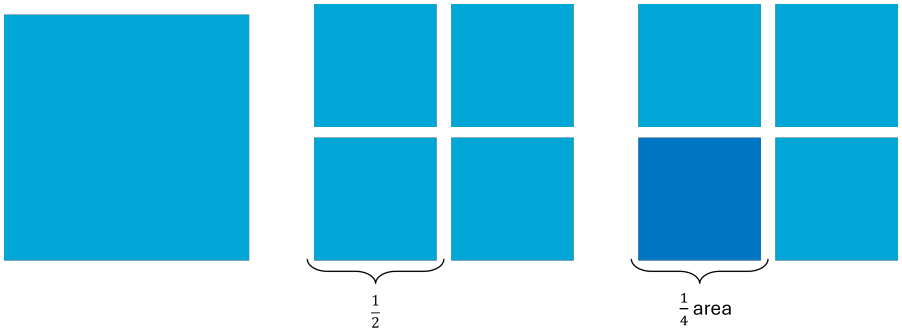


Figure 4.2: Scaling Factor $\frac{1}{2}$ and Mass Scaling Factor $\frac{1}{4}$ for a Square.

3. A cube (with width 1) can be broken up into eight smaller cubes. Each of which is a perfect copy of the original, scaled down by a half.

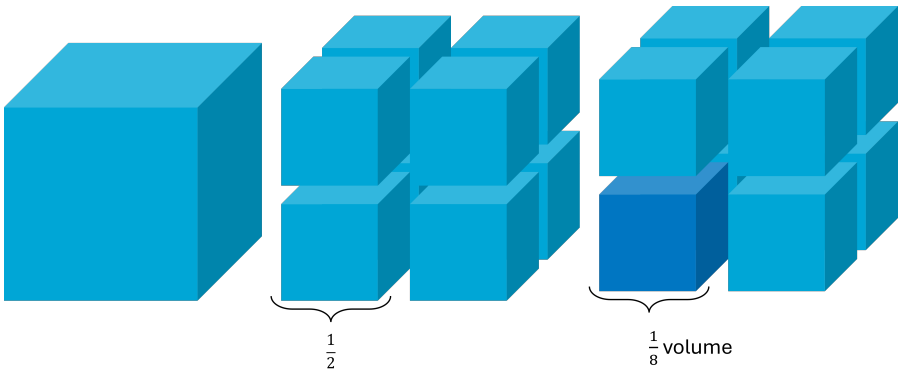


Figure 4.3: Scaling Factor $\frac{1}{2}$ and Mass Scaling Factor $\frac{1}{8}$ for a Cube.

These three shapes can be split into smaller copies of themselves and are therefore what is called self-similar. Each copy was scaled down by a half, but if we measure the length, area and volume: the smaller line becomes half the length of the original line; the smaller square one quarter the area of the original square and the smaller cube one eighth the volume of the original cube.

To generalise the concept of scaling, we will refer to quantities such as length, area, and volume collectively as mass or measure. The fractal dimension characterises how this mass changes as a shape is scaled. In the previous examples, we considered self-similar shapes, as they provide a clear and intuitive illustration of how the measure scales with size. The following relations between the scaling factor $\frac{1}{2}$ and the mass scaling factor could therefore easily be attained.

1. **Line:** mass scaling factor = $\frac{1}{2}$.
2. **Square:** mass scaling factor = $\frac{1}{4} = (\frac{1}{2})^2$.
3. **Cube:** mass scaling factor = $\frac{1}{8} = (\frac{1}{2})^3$

These expressions show that the mass scaling factor can be written as a power of the scaling factor $\frac{1}{2}$, where the exponent corresponds to the dimension of the shape. In general, for self-similar shapes, their dimension δ satisfies

$$\text{mass scaling factor} = (\text{scaling factor})^{\delta}, \tag{4.1}$$

which results in the following expression for δ

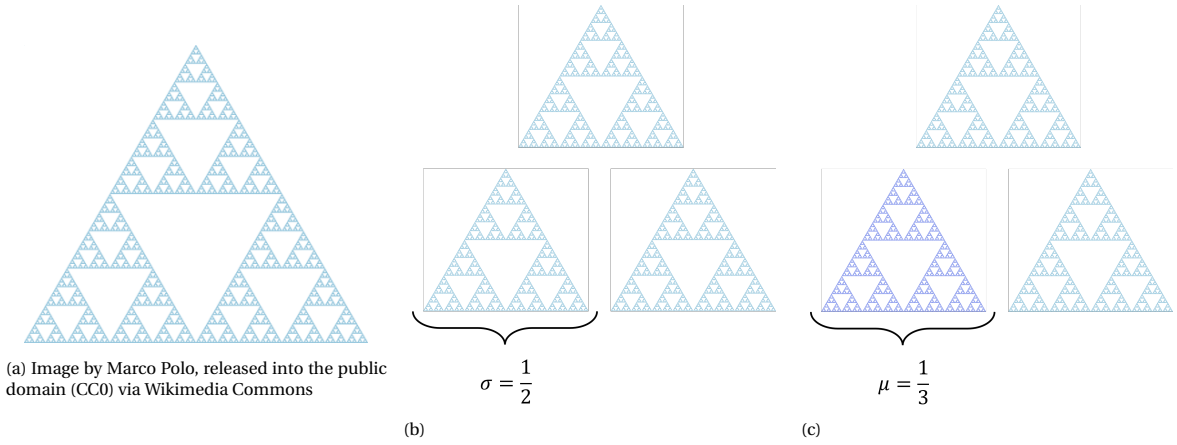
$$\delta = \log_{\sigma}(\mu) = \frac{\ln(\mu)}{\ln(\sigma)}, \tag{4.2}$$

where μ is the mass scaling factor and σ the scaling factor.

Now, let us consider another self-similar shape, the *Sierpiński triangle*, see Figure 4.4a. Unlike the previous examples, we will see that the Sierpiński triangle has a non-integer dimension and is therefore a fractal. The Sierpiński triangle can be broken up into three smaller triangles. Each of which is a perfect copy of the original, scaled down by a half. Its mass is scaled down by a third. Therefore, the fractal dimension is equal to the number δ that satisfies $\frac{1}{3} = (\frac{1}{2})^{\delta}$. Using expression 4.2,

$$\delta = \frac{\ln \frac{1}{3}}{\ln \frac{1}{2}} = \frac{\ln 1 - \ln 3}{\ln 1 - \ln 2} = \frac{\ln 3}{\ln 2} \approx 1.585.$$

So, in fractal geometry, the Sierpiński triangle is a 1.585-dimensional shape. For an illustration, see Figure 4.4.



(a) Image by Marco Polo, released into the public domain (CC0) via Wikimedia Commons

(b)

(c)

Figure 4.4: Illustration of the Scaling Factor and Mass Scaling Factor of the Sierpiński Triangle. The Sierpiński triangle (a), scaling factor $\sigma = \frac{1}{2}$ (b) and mass scaling factor $\mu = \frac{1}{3}$

Let us examine one more shape, the *von Koch curve*, which has a different scaling factor than a half. The von Koch curve can be broken up into four smaller curves. Each of which is a perfect copy of the original, scaled down by a third. Its mass is scaled down by a fourth. Therefore, the fractal dimension is equal to the number δ that satisfies $\frac{1}{4} = (\frac{1}{3})^{\delta}$. Using expression 4.2

$$\delta = \frac{\ln 1/4}{\ln 1/3} \approx 1.262.$$

So, in fractal geometry, the von Koch curve is a 1.262-dimensional shape. For an illustration, see Figure 4.5.

The shapes discussed so far are great for an intuitive understanding of fractal dimension, as their scaling and mass scaling factors are straightforward to determine. Howeve, we want to extend this to more complex or irregular shapes. To analyse such cases, we require a more general method for calculating the fractal dimension. The Hausdorff measure is widely regarded as the most formal definition of fractal dimension, but more challenging to compute [5]. The box-counting method is more straightforward and is a method that estimates the dimension numerically.

4.1.1. Box-counting Method

Since the box-counting method is the fundamental used method for the approach used in Section 4.2, a short explanation will be given. To illustrate the limitations of our earlier method, consider the example of a circle. We know that scaling a circle by a factor of two increases its area by a factor of four, because of the properties of a disk. However, unlike self-similar shapes, it is not possible to construct the larger circle by assembling four exact copies of the original, see Figure 4.6. As a result, we can not directly verify whether the mass scaling factor is indeed four, and the previous approach to determine dimension breaks down in this case.

We will now approach this problem using the box-counting method. This involves overlaying a grid of boxes on the plane and counting how many boxes intersect the circle, as in Figure 4.7. Next, we scale the system by a factor of two and count again how many boxes are required to cover the circle. The ratio of the number of boxes at different scales approximates the mass scaling factor, which we again use in Equation 4.2.

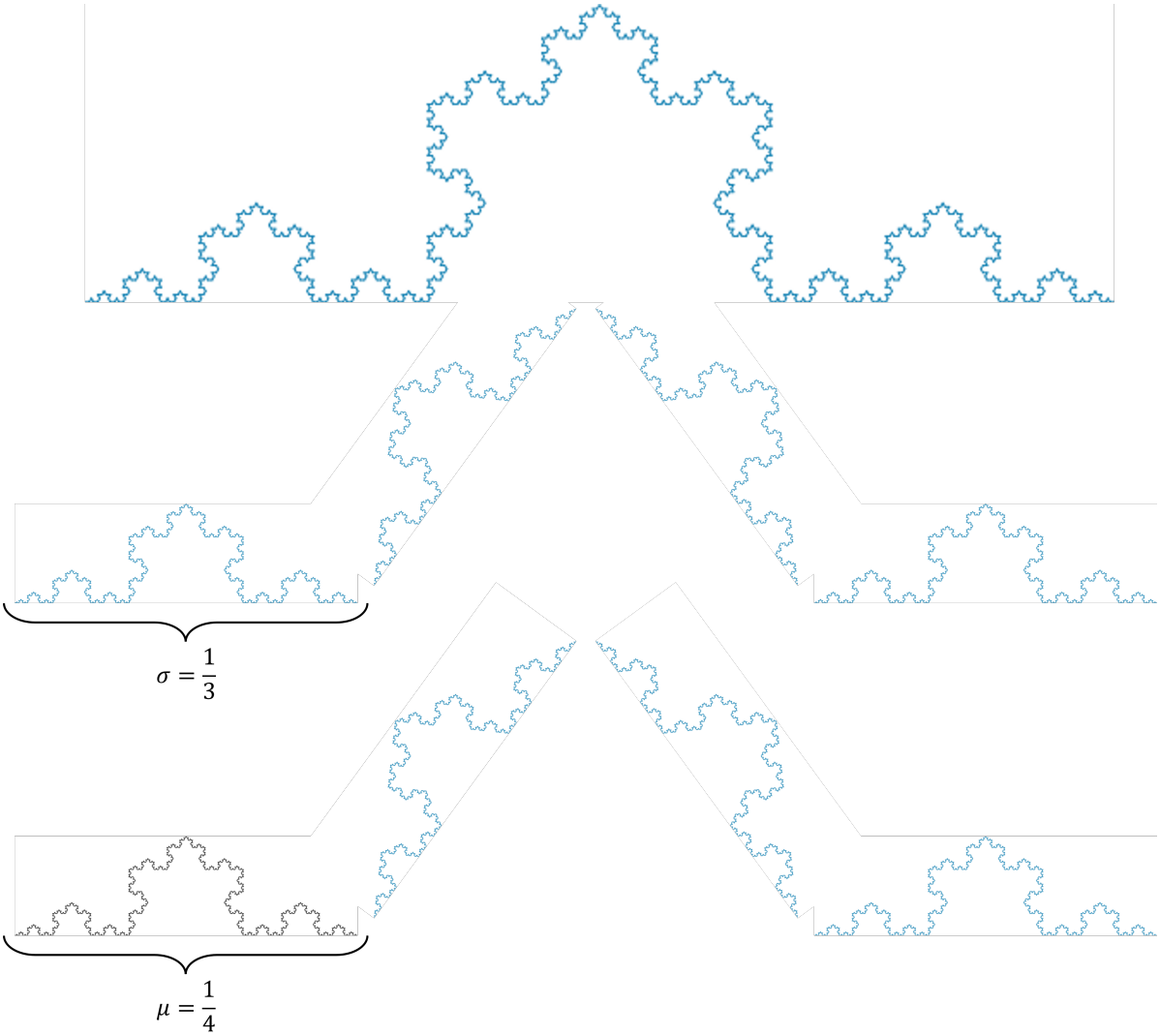


Figure 4.5: Illustration of the Scaling Factor $\sigma = \frac{1}{3}$ and Mass Scaling Factor $\mu = \frac{1}{4}$ of the Von Koch Curve

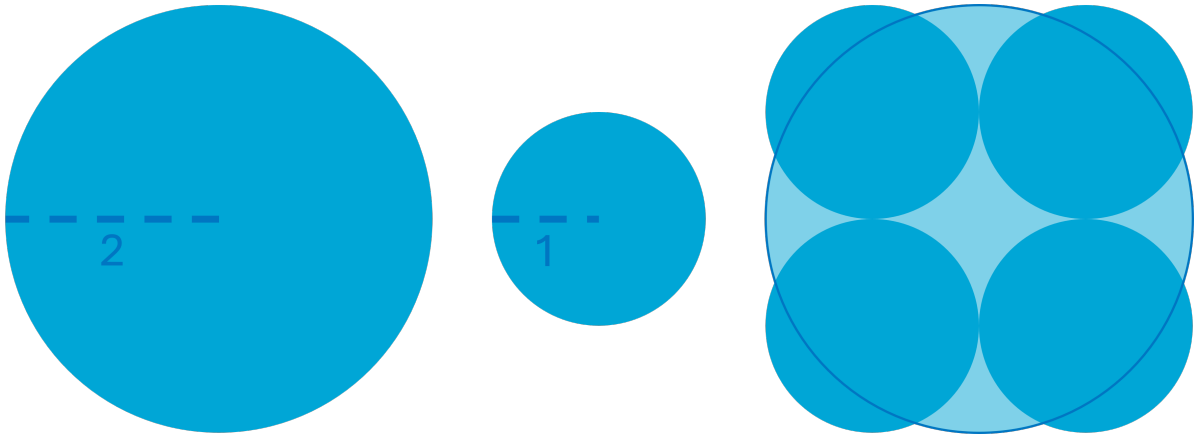


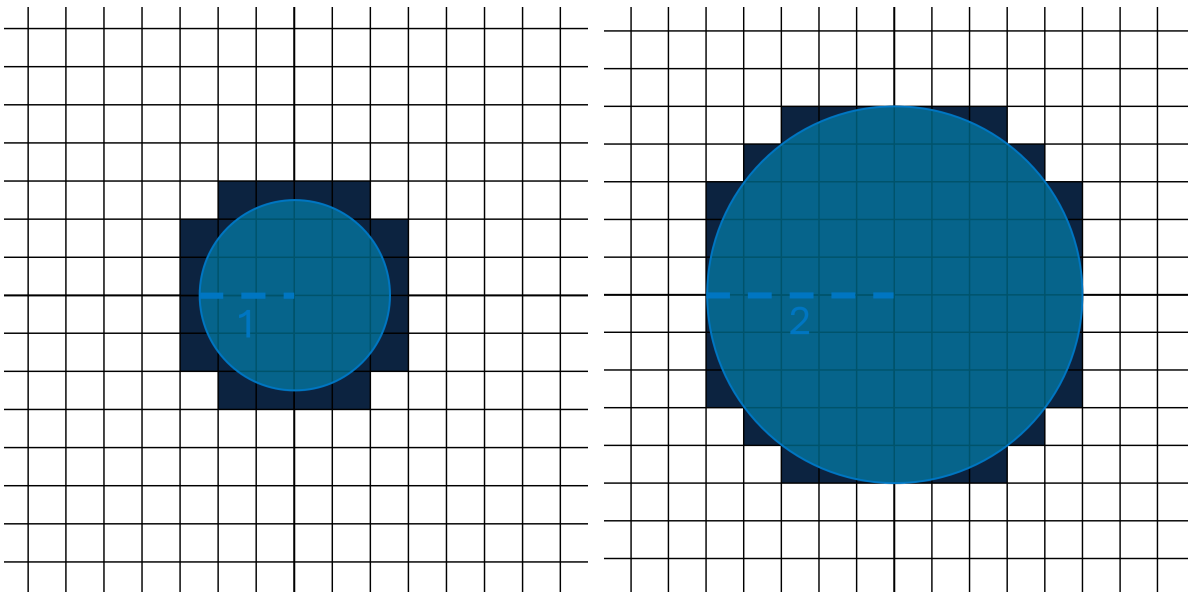
Figure 4.6: Illustration of fitting four scaled down Circles into the Original Circle, Showing the Limitations of Method 4.2.

In Figure 4.7. after one iteration, we estimate the dimension of a circle, $\delta = 1.292$, which is not close to actual dimension $\delta = 2$ of a circle, but as the size of the shape increases (more iterations of the box-counting method), this value converges to the true fractal dimension. Alternatively, and more practically, one can fix the shape and instead scale down the box size, which is the standard approach in box-counting. In this case, the theoretical box-counting dimension is defined as:

$$\delta_{box} = \lim_{\epsilon \to 0} \frac{\ln N}{\ln 1/\epsilon}, \tag{4.3}$$

where N denotes the boxes needed to cover the shape and ϵ is the side-length of the boxes [19].

In practice, we can not make the box size infinitely small. Instead, we compute the number of boxes N required to cover the shape at several finite scales ϵ, and plot $\ln(1/\epsilon)$ against $\ln N$. A linear regression is then performed on this log-log plot, and the slope of the resulting line provides an estimate of the box-counting dimension.



(a) (b)

Figure 4.7: Illustration of Box-Counting Method. In Figure (a) 32 boxes cover the circle with radius 1. Then, the circle is scaled by a factor 2, resulting in a circle with radius 2, figure (b). Now, 88 boxes cover the circle, resulting in a Mass Scaling Factor $\mu = \frac{88}{36} = 2.44$. Completing the calculation of the fractal dimension, we find $\delta = \frac{\ln 2.44}{\ln 2} = 1.29$.

4.2. Can Fractal Dimension Serve as a New Distortion Metric?

So far, self-similar shapes with integer and non-integer dimensions (fractals) and non-self-similar shapes are covered. The French mathematician Mandelbrot described coastlines as not perfect self-similarity exhibiting shapes, but he found non-integer dimension in coastlines [11, 12]. Therefore it is not an exact fractal, but a real-world fractal: zooming in reveals more detail, without a clear resolution limit. When evaluating real-world fractals we can consider the fractal dimension as a measure of robustness in the shape or figure.

Consequently, one may expect this robustness to differ for distinct map projections. Although counter-intuitive for the shapes discussed in Section 4.1, if this holds for coastlines, we construct a new distortion measure. Recently, the fractal dimensions of different stretches of coastline before and after projection using the divide-and-conquer algorithm (implementation of box-counting) and image processing were computed [10], see Table 4.1. However, these results are not sufficient to conclude if there is indeed a difference in fractal

Region	**Fractal Dimension**	
	Mercator	Equirectangular
Northern Norway: High Latitude, Horizontal Coastline	1.5361	1.4748
Southwestern Greenland: Medium-High Latitude, Vertical Coastline	1.5832	1.5917
Northeastern Greenland: High Latitude, Vertical Coastline	1.4847	1.6140
Northern Tip of the Antarctic Peninsula: Medium-High Latitude, Mixed Coastline	1.5362	1.5240
Southwestern Antarctic Peninsula: High Latitude, Mixed Coastline	1.3558	1.4055

Table 4.1: Fractal dimensions of Different Stretches of Coastline after projected by the Mercator and Equirectangular Projections [10].

dimension along the entire coastline between the Mercator and Equirectangular projection. Furthermore, the accuracy of the implemented box-counting method depends on the resolution of the image. Properties such as the plotted line-width of the coastline also influence the outcome. So, before introducing the fractal dimension of coastlines as new distortion measure, we need to verify the results for the complete coastline.

4.2.1. Improved Method: Dot-Based Box Counting

As mentioned in Section 4.1 we will use a method similar to the box-counting method. It is an effective approach to compute the box-counting dimension based on the mathematical definition and intervals [19]. In particular, this method improves accuracy by eliminating any dependence on pixel resolution or image quality. Instead, it uses *dots*. Dots are described by the creators of the algorithm as individual points that are mathematically generated to represent a fractal object. Unlike pixels, which are tied to image resolution and screen quality, dots are derived from precise mathematical definitions such as explicit equations (e.g., $y = x$, $y = x^2$). The density and accuracy of the representation improve as the spacing between dots becomes smaller. With this method, the fractal is displayed using dots rather than pixels. As a result, traditional detection boxes based on pixel grids or matrix structures are no longer applicable. Instead, a mathematical criterion is employed to determine which sets of points from the fractal fall within each detection box. The whole domain of the fractal will be split uniformly into subintervals with side length $\hat{\epsilon}$. In order to judge whether at least one point falls in the detected interval, an inequality that serves as the inclusion criterion for each box is formalised

$$m\hat{\epsilon} \leq x_i < (m+1)\hat{\epsilon}, \quad n\hat{\epsilon} \leq y_i < (n+1)\hat{\epsilon}, \quad m,n \in \mathbb{Z}, \quad m,n \in \mathbb{Z}, \quad 0 \leq m,n \leq \left(\frac{\iota}{\hat{\epsilon}} - 1\right), \tag{4.4}$$

where

- x_i, y_i: coordinates of a point on the fractal;
- $\hat{\epsilon}$: side length of each detection box;
- m, n: integer indices of the detection box in the horizontal and vertical directions, respectively;
- ι: side length of the entire domain in which the fractal is contained (e.g., $\iota = 1024$);
- $\left(\frac{\iota}{\hat{\epsilon}} - 1\right)$: maximum number of boxes (per axis) that fit in the domain.

Having established the methodology, the next step is to assess whether the results presented in Table 4.1 also hold for the entire coastline of all continents. To this end, one of the most accurate coastline datasets

Projection	Fractal Dimension	Standard Error of Regression (SE)	Confidence Interval (±1 SE)
Mercator	1.27616	0.01148	[1.2647, 1.2876]
Equirectangular	1.25638	0.00600	[1.2504, 1.2624]
Bonne	1.24646	0.00577	[1.2407, 1.2522]
Sinusoidal	1.25391	0.00583	[1.2481, 1.2597]
Wiechel	1.23020	0.00737	[1.2228, 1.2376]

Table 4.2: Fractal Dimension of the Entire Coastline for Various Mappings using the Improved Box-Counting, with Interval Size $\hat{\epsilon} \in [2^{-4}, 2^{-5}, \ldots, 2^{-14}]$ and $\iota = 1$. Calculated by Appendix E code *FractalDimension.py*.

available was used, and various map projections were applied to determine the (x_i, y_i) coordinates. The resulting estimated fractal dimensions are shown in Table 4.2. While the confidence intervals suggest that the fractal dimensions vary between projections, these intervals are based solely on the standard error of the regression fit and do not account for potential numerical errors in the box-counting process. Consequently, although there are differences between projections, the relatively small variations and the uncertainty involved make it difficult to draw definitive conclusions about their statistical significance.

4.2.2. Verification with Hausdorff Dimension

To get a clearer and more reliable understanding, we now turn to a mathematical definition of fractal dimension that doesn't rely on approximations or sampling. This brings us to the Hausdorff dimension. The formal definition is given below [1, 16].

Definition 4.1 (Hausdorff dimension)**.** Let X be a metric space. If $S \subset X$ and $\delta \in [0, \infty)$, then we let

$$H_\eta^\delta(S) = \inf\left\{ \sum_{i=1}^{\infty} (\operatorname{diam}(U_i))^\delta : S \subseteq \bigcup_{i=1}^{\infty} U_i,\ \operatorname{diam}(U_i) < \eta \right\}. \tag{4.5}$$

Here, $\operatorname{diam}(U_i) = \sup\{\|x - y\| : x, y \in U_i\}$. Moreover, the infimum is taken over all countable covers U of S. The Hausdorff δ-dimensional outer measure is defined as

$$\mathscr{H}^\delta(S) = \lim_{\eta \to 0} H_\eta^\delta(S). \tag{4.6}$$

Finally, the *Hausdorff dimension* is given by

$$\dim_H(S) = \inf\left\{ \delta \geq 0 : \mathscr{H}^\delta(S) = 0 \right\}. \tag{4.7}$$

Because this measure will only be used as a tool to determine if the results from Table 4.1 and Table 4.2 are accurate, its full derivation will not be included. In particular, we want to show whether the Hausdorff dimension is preserved under the different map projections. This approach provides a practical check on the validity of the results in Table 4.1 and Table 4.2, without requiring a full measurement of the Hausdorff measure itself. The goal is to prove if $\dim_H(S_E) = \dim_H(S_M)$, where $S_E \subset \mathbb{R}^2$ is the image of the coastline in Equirectangular projection, and $S_M \subset \mathbb{R}^2$ the image of the coastline under the Mercator projection.

Proof. Let $S_E \subset \mathbb{R}^2$ be the image of the coastline in Equirectangular projection, and let $S_M \subset \mathbb{R}^2$ be the image of the same coastline under the Mercator projection. Since $\mathbb{R}^2$ is a metric space and S_E and S_M inherit the metric, they are metric spaces themselves. As mentioned, the goal is to show that $\dim_H(S_E) = \dim_H(S_M)$.

To do this, it suffices to show that the map $F : S_E \to S_M$ that converts Equirectangular coordinates into Mercator coordinates is bi-Lipschitz on the relevant domain.

We define the transformation $F = T_M \circ T_E^{-1}$:

$$F(x, y) = \left(x, \ln \tan\left(\frac{\pi}{4} + \frac{y}{2} \right) \right),$$

where T_M and T_E are the transformation formulas of the Mercator and Equirectangular projections, respectively.

Claim: We now claim that F is bi-Lipschitz on any compact subset $D \subset S_E \subset T_E^{-1}([-\pi, \pi) \times (-\frac{\pi}{2}, \frac{\pi}{2}))$, to avoid singularities near the poles.

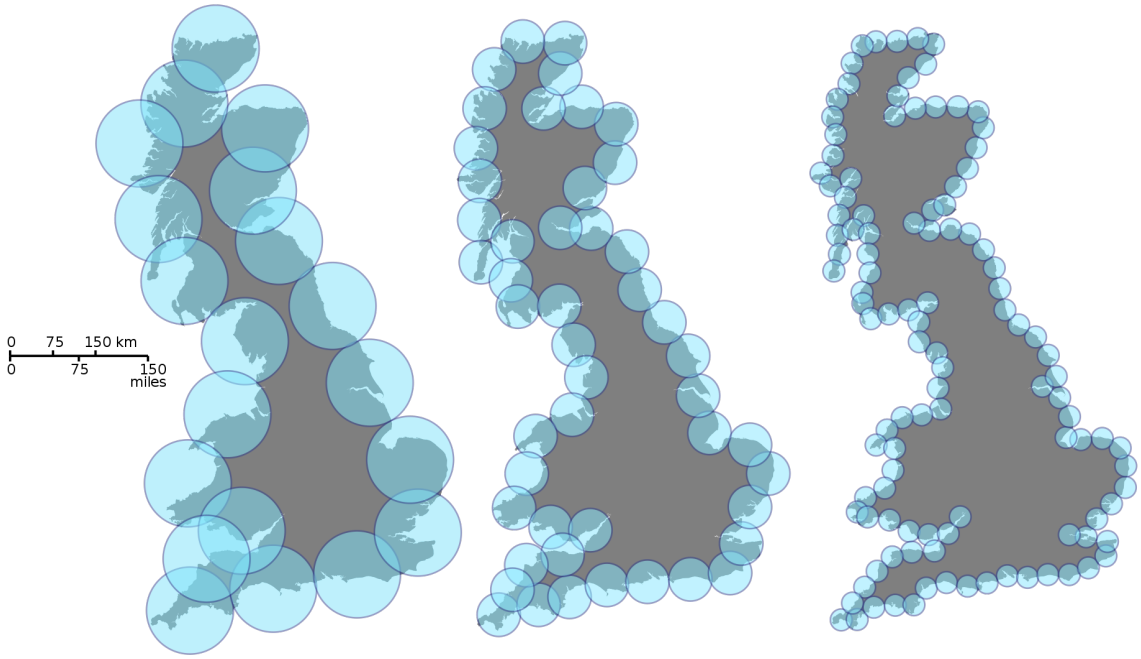


Figure 4.8: Illustration of the Hausdroff Definition by Covering the Coast of Great Britain with Sets U_i. Image by Prokofiev, licensed under CC BY-SA 4.0 via Wikimedia Commons.

Since S_E is not defined near the poles, $D \subseteq S_E$. By bi-Lipschitz, there exist constants $c, C > 0$ such that for all $\hat{x}_1, \hat{x}_2 = (x_1, y_z), (x_2, y_2) \in D$:

$$c\,\|\hat{x}_1 - \hat{x}_2\| \leq \|F(x_1) - F(x_2\| \leq C\,\|\hat{x}_1 - \hat{x}_2\|.$$

Now, suppose $\mathscr{H}^\delta(S_E) = 0$. Then for any $\eta_1 > 0$, there is a cover $\bigcup_{i=1}^{\infty} U_i$ of S_E with $\operatorname{diam}(U_i) < \eta$ such that:

$$\sum_{i=1}^{\infty} \operatorname{diam}(U_i)^\delta < \eta_1.$$

Since F is bi-Lipschitz on $D \subseteq S_E$, we have:

$$\operatorname{diam}(F(U_i)) \leq C \cdot \operatorname{diam}(U_i).$$

So,

$$\sum_{i=1}^{\infty} \operatorname{diam}(F(U_i))^\delta \leq C^\delta \cdot \sum_{i=1}^{\infty} \operatorname{diam}(U_i)^\delta < C^\delta \cdot \eta_1.$$

Since $\delta \in [0, \infty)$, $\mathscr{H}^\delta(F(S_E)) = 0$. The reverse inequality follows by using the inverse map F^{-1}, which is Lipschitz (similar to bi-Lipschitz, but only the upper bound holds), so:

$$\mathscr{H}^\delta(S) = 0 \iff \mathscr{H}^\delta(F(S_E)) = 0 \implies \dim_H(S_E) = \dim_H(F(S_E)) = \dim_H(S_M).$$

Proof of claim: We now formalise the earlier claim that the Mercator transformation

$$F(x, y) = \left(x,\, \ln\tan\left(\frac{\pi}{4} + \frac{y}{2}\right)\right)$$

is bi-Lipschitz on compact subsets of the Equirectangular domain that are bounded away from the poles.

Let $f(y) = \ln\tan\left(\frac{\pi}{4} + \frac{y}{2}\right)$. The function f is strictly increasing on $(-\frac{\pi}{2}, \frac{\pi}{2})$ because it is the composition of increasing functions. Therefore, $F(x, y) = (x, f(y))$ is injective on any domain $D = [a, b] \times [c, d] \subset [-\pi, \pi) \times (-\frac{\pi}{2}, \frac{\pi}{2})$.

Step 1: Lipschitz continuity of F. Since F is separable in x and y, we analyse the Lipschitz property of each component.

Clearly, $x \mapsto x$ is Lipschitz with constant 1. We now analyse $f(y)$. We compute its derivative:

$$f'(y) = \frac{1}{\cos y}$$

On a *compact* interval $[c,d] \subset (-\frac{\pi}{2}, \frac{\pi}{2})$, this derivative is strictly positive and bounded: there exist constants $0 < m \leq f'(y) \leq M < \infty$. By the Mean Value Theorem, this implies f is Lipschitz continuous with constant M.

Therefore, for all $(x_1, y_1), (x_2, y_2) \in D$,

$$\|F(x_1, y_1) - F(x_2, y_2)\| \leq \sqrt{1 + M^2} \cdot \|(x_1, y_1) - (x_2, y_2)\|,$$

so F is Lipschitz on D.

Step 2: Lipschitz continuity of F^{-1}. Since f is strictly increasing and differentiable with derivative bounded below by $m > 0$ on the domain, it is invertible, and its inverse function f^{-1} is differentiable as well. The derivative of the inverse satisfies:

$$\left(f^{-1}\right)'(y) = \frac{1}{f'\left(f^{-1}(y)\right)} \leq \frac{1}{m}.$$

Therefore, by the Mean Value Theorem, f^{-1} is Lipschitz continuous with Lipschitz constant $\frac{1}{m}$, and

$$F^{-1}(x, y) = \left(x, f^{-1}(y)\right)$$

is Lipschitz on $F(D)$ with constant $\sqrt{1 + (1/m)^2}$. □

The above proof shows that the Hausdorff (fractal) dimension is invariant under bi-Lipschitz transformations. Since projections such as the Mercator and Equirectangular maps are bi-Lipschitz (excluding singularities at the poles), the fractal dimension of the coastline should theoretically remain unchanged—both locally and globally—under these mappings. This theoretical invariance, however, contradicts the differences observed in Table 4.1, suggesting that the differences in measured fractal dimensions are likely artifacts of sampling or numerical methods rather than true geometric distortions. Consequently, introducing fractal dimension as a seventh distortion measure appears unjustified, as it does not reflect true geometric distortion introduced by the projection.

4.3. Measuring Distortions Along Coastlines

Although having established that fractal dimension can not meaningfully capture distortion, the coastline itself remains a valuable object of analysis. In many cartographic contexts, such as thematic maps focused on maritime navigation, coastal ecology, or geopolitical boundaries, the map content is essentially limited to the coastline. In such cases, the distortion metrics computed over entire landmasses or oceans as in Chapter 3, become less relevant. Instead, it is more appropriate to evaluate these standard distortions specifically along the coastline itself. By restricting our attention to the coastline geometry, we obtain distortion measures that more directly reflect the primary or only feature of interest.

For this purpose, we will reuse the distortion definitions introduced in Chapter 2, but omit the distance distortion, as it measures on the surface rather than the coastline. Since, we are evaluating coastlines now, this measure along the surface is not meaningful any more. Furthermore, the definition of the boundary cut requires modification, as the length of the cut on the globe does not correspond directly to the cut length along the coastline. A more appropriate approach might be to count the number of locations where the coastline is intersected by the cut. However, this measure is also unsuitable, since for most cylindrical projections the count would be three, a relatively large value that would disproportionately dominate the overall distortion metrics. Additionally, small shifts in the coastline can cause large fluctuations in this value. For these reasons, the boundary cut distortion was omitted from the coastline analysis. All other distortions are still meaningful along the coastline geometry and will be evaluated.

4.3.1. Richardson Effect

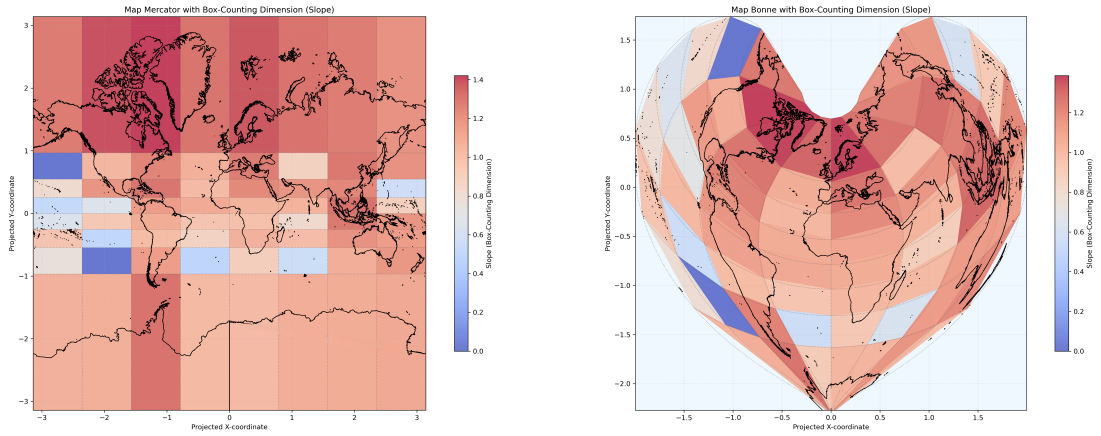
To properly interpret the distortion values, we will first examine how these distortions are expected to behave in theory. This provides a mathematical and geometric basis for the results, helping to distinguish genuine projection effects from artifacts caused by resolution or sampling.

First described by Richardson [15], the Richardson effect describes the observation that the measured length of an irregular boundary, such as a coastline, increases as the scale of measurement becomes finer [11]. More formally, if $L(\varepsilon)$ denotes the measured length of the coastline when using measuring steps of size ε, then

$$L(\varepsilon) \sim \varepsilon^{1-\delta},$$

where δ is the fractal dimension of the coastline. For idealised smooth curves, $\delta = 1$, and the length is independent of scale. For coastlines and other fractal-like structures, however, we found $\delta > 1$, so the measured length increases as $\varepsilon \to 0$. This relation implies that distortion metrics computed along coastlines can vary significantly depending on the resolution of the dataset used: higher-resolution datasets reveal more geometric detail and thus exhibit greater sensitivity to distortions. As a result, some pieces of coastline in a dense dataset, might disproportionately dominate the overall distortion metrics.

To further complement this analysis, heatmaps are generated, depicting localised fractal dimensions along the coastline. These visualisations help identify which geographic regions have the greatest geometric irregularity and contribute the most to the overall distortion measure. From Figure 4.9, it is evident that the Nordic regions, particularly the Canadian Arctic, exhibit the highest local fractal dimensions, indicating geometric complexity. Additionally, the coastline of Antarctica also displays significant irregularity. Some regions register fractal dimensions below one; these values are likely artifacts resulting from sparse data or the absence of coastline in those areas. A similar analysis can be performed at different region sizes, as shown



(a) (b)

Figure 4.9: Local determined fractal dimension of the Mercator (a) and Bonne (b) projections in 64 regions. Figures made with Appendix E code *Fractal_Dimension_Plot.py*

in Figure 4.10. This reveals that the southern tip of South America exhibits a highly fractal coastline, and Indonesia also contributes significantly due to its dense, intricate island structures. For more analysed map projections, see Appendix D.

4.3.2. Normalising to the Arc Length

We have shown that by the Richardson effect, highly fractal pieces of coastline can disproportionally influence the total distortion. Yet we still prefer high resolution for its greater accuracy. To balance these needs, we need to normalise each local distortion. In Section 2.5 we defined the global distortion as either an average or a root-mean-square of local distortions. This method will still be applied to find the distortion along the entire coastline, but each local measurement will be weighted to its corresponding arc length. The arc length is computed using the midpoint rule, as the points in the coastline datasets are irregularly spaced and are not uniformly divided along the curve. Consequently, the distortion along the entire coastline is given by the following revised formulation of the root mean square.

Definition 4.2 (Root Mean Square Along the Coastline (RMS^c) with Midpoint Measure)**.** The *Root Mean Square* (RMS) of a distortion D, defined along a coastline with length L, is computed using a weighted average that reflects the geometric distribution of the coastline points. Given discrete samples $\{x_i\}_{i=1}^n$ ordered along the coastline and corresponding distortion values $\{D_i\}_{i=1}^n$, the RMS with a midpoint-based weight is

defined as:

$$\begin{aligned} \mathrm{RMS}^c(D_1,\ldots,D_n) &= \sqrt{\frac{1}{L}\sum_{i=1}^{n} D_i^2 \cdot (\text{midpoint distance})} \\ &= \sqrt{\frac{1}{L}\left[\sum_{i=2}^{n-1} D_i^2 \cdot \frac{\|x_{i-1}-x_i\|+\|x_i-x_{i+1}\|}{2} + D_1^2\cdot\|x_1-x_2\| + D_n^2\cdot\|x_{n-1}-x_n\|\right]}. \end{aligned} \tag{4.8}$$

Here, the segment lengths act as weights to approximate integration over arc length. The endpoints D_1 and D_n are treated separately because they lack neighbours, unlike the interior points.

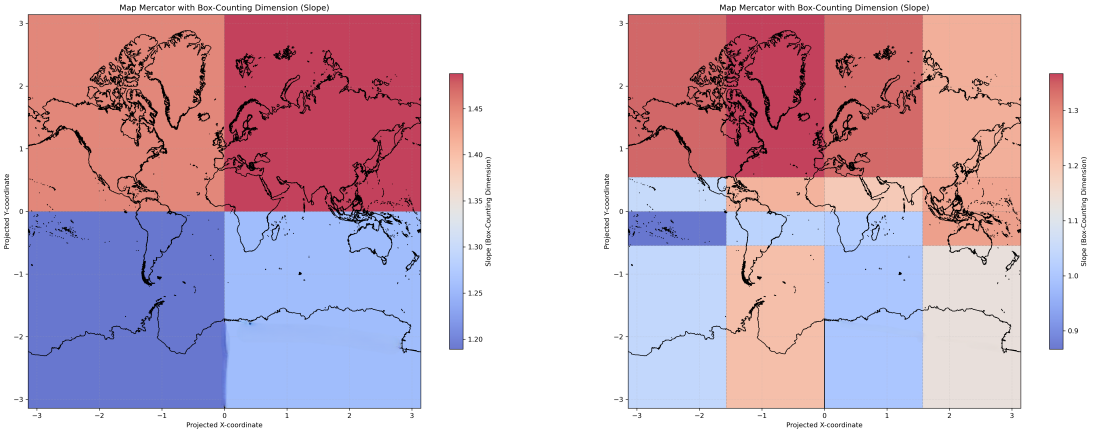
Although the expression may look like it could be written more generally, this form is necessary to account for the structure of the coastline data that will be used. It guarantees that the global distortion is weighted by the actual physical distance along the coastline, as opposed to a simple unweighted average over sampled points.

Finally, the coastline data at different three levels of resolution: low, intermediate and high, will be employed. This allows us to examine how strong the resolution influences the distortion metrics along coastlines. The data was downloaded from the Global Self-consistent, Hierarchical, High-resolution Geography Database (same as was used for the fractal dimension computations). See Table 4.3 for an overview of the data. One can check that the Richardson relation holds within the error marge of the fractal dimension.

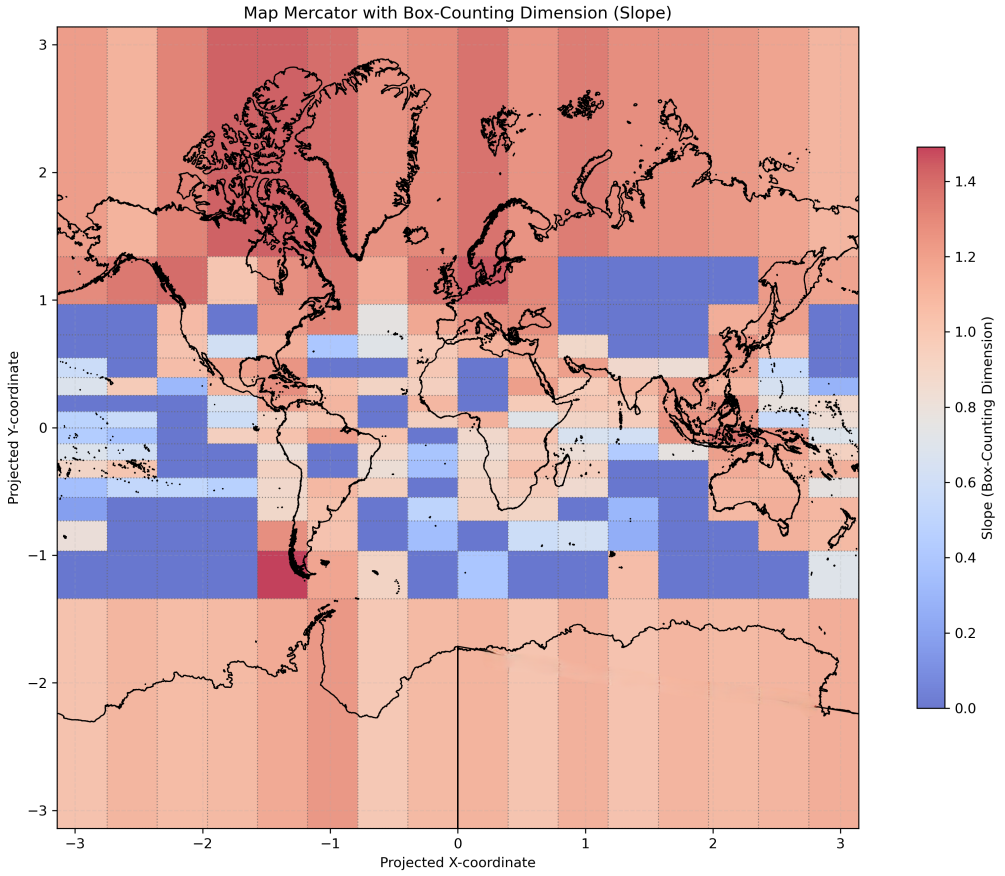
Resolution	**Number of Points**	**Length (km)**	**Step-size ε (m)**	**Estimated Fractal Dimension (±1 SE)**
Low	314,514	1,194,015	3796	[1.220, 1.252]
Intermediate	1,514,70	1,500,390	990	[1.229, 1.254]
High	9,516,273	1,666,095	72	[1.250, 1.262]

Table 4.3: Coastline-Data Overview.

While we have established that fractal dimension itself does not qualify as a standalone distortion metric, its spatial patterns inform our understanding of projection behavior along coastlines. In the next chapter, we build on this foundation by presenting the distortion values computed for coastlines at varying levels of resolution.



(a) (b)



(c)

Figure 4.10: Local determined Fractal Dimension of the Mercator Projection in Different Region Sizes: 4 Regions (a), 16 Regions (b), 256 Regions (c).

5

Coastline Distortion Results

5.1. Low Resolution

As mentioned, the data used in this study was obtained from the Global Self-consistent, Hierarchical, High-resolution Geography Database. It is important to note that the dataset referred to as low resolution in our analysis corresponds to what GSHHG labels as intermediate precision. This naming may be misleading, as it does not represent the intermediate resolution in our study. That is, because GSHHG's highest-resolution data is labeled as full precision and not as high precision. The following distortion results were computed by Appendix E code *coastline_distortions.py*.

Projection	Area	Isotropy	Flexion	Skewness	Total
Azimuthal Equidistant (North Polar)	0.471	0.590	0.622	0.257	1.023
Cassini	0.394	0.487	0.608	0.568	1.084
Equidistant Conic	0.471	0.494	0.695	0.456	1.156
Winkel Tripel	0.250	0.509	0.943	0.418	1.385
Lambert Azimuthal Equal-Area (North Polar)	0	0.954	0.644	0.311	1.421
Aitoff	0.106	0.743	0.890	0.483	1.588
Winkel II	0.311	0.599	0.950	0.605	1.723
Lambert Conic	1.107	0	0.511	0.506	1.742
Bonne	0.176	0.959	0.768	0.494	1.785
Hammer	0.	0.869	0.927	0.551	1.918
Miller Cylindrical	0.709	0.316	0.895	0.832	2.095
Equirectangular	0.469	0.672	0.874	0.899	2.243
Wiechel	0	1.070	0.970	0.418	2.261
Mercator	0.939	0	0.867	0.859	2.371
Sinusoidal	0	0.972	0.983	0.808	2.563
Stereographic (North Polar)	1.541	0	0.567	0.563	3.012
Gall-Peters	0	0.977	1.042	1.010	3.060
Collignon	0	1.253	0.888	0.946	3.253
Behrmann	0	1.156	0.909	1.116	3.408
Lambert Cylindrical Equal-Area	0	1.344	0.812	1.206	3.919
Spilhaus Stereographic	1.579	0	1.104	1.103	4.930
Central Cylindrical	1.408	0.672	0.896	1.441	5.316

Table 5.1: Distortion Results of the Low Resolution Data normalised with respect to the Arc Length using Definition 4.2.

5.2. Intermediate Resolution

The intermediate resolution dataset corresponds to what GSHHG labels as high precision. Using this dataset, the following distortion results were computed.

Projection	Area	Isotropy	Flexion	Skewness	Total
Azimuthal Equidistant (North Polar)	0.464	0.580	0.613	0.249	0.991
Cassini	0.395	0.488	0.607	0.564	1.081
Equidistant Conic	0.464	0.490	0.682	0.448	1.121
Winkel Tripel	0.247	0.502	0.934	0.408	1.351
Lambert Azimuthal Equal-Area (North Polar)	0	0.938	0.636	0.304	1.376
Aitoff	0.106	0.735	0.878	0.472	1.546
Winkel II	0.306	0.592	0.940	0.591	1.678
Lambert Conic	1.092	0	0.499	0.494	1.686
Bonne	0.174	0.940	0.752	0.481	1.711
Hammer	0	0.860	0.916	0.539	1.868
Miller Cylindrical	0.699	0.311	0.889	0.821	2.050
Equirectangular	0.463	0.665	0.871	0.883	2.194
Wiechel	0	1.055	0.963	0.410	2.208
Mercator	0.925	0	0.859	0.852	2.320
Sinusoidal	0	0.963	0.968	0.792	2.491
Stereographic (North Polar)	1.520	0	0.558	0.553	2.927
Gall-Peters	0	0.962	1.044	0.990	2.995
Collignon	0	1.239	0.876	0.928	3.163
Behrmann	0	1.141	0.912	1.097	3.335
Lambert Cylindrical Equal-Area	0	1.329	0.815	1.187	3.840
Spilhaus Stereographic	1.571	0	1.106	1.105	4.913
Central Cylindrical	1.388	0.665	0.884	1.422	5.172

Table 5.2: Distortion Results of the Intermediate Resolution Data normalised with respect to the Arc Length using Definition 4.2.

5.3. High Resolution

The high resolution dataset corresponds to what GSHHG labels as full precision. Using this dataset, the following distortion results were computed.

Projection	Area	Isotropy	Flexion	Skewness	Total
Azimuthal Equidistant (North Polar)	0.474	0.591	0.620	0.258	1.025
Cassini	0.393	0.485	0.604	0.557	1.065
Equidistant Conic	0.474	0.498	0.683	0.454	1.145
Winkel Tripel	0.244	0.499	0.932	0.403	1.340
Lambert Azimuthal Equal-Area (North Polar)	0	0.956	0.641	0.313	1.423
Aitoff	0.105	0.732	0.875	0.467	1.532
Winkel II	0.303	0.590	0.939	0.585	1.662
Bonne	0.178	0.941	0.750	0.485	1.715
Lambert Conic	1.121	0	0.504	0.500	1.762
Hammer	0	0.856	0.912	0.534	1.850
Miller Cylindrical	0.693	0.309	0.889	0.818	2.035
Equirectangular	0.458	0.662	0.872	0.878	2.181
Wiechel	0	1.071	0.969	0.421	2.263
Mercator	0.917	0	0.858	0.851	2.300
Sinusoidal	0	0.959	0.963	0.785	2.464
Gall-Peters	0	0.954	1.049	0.983	2.976
Stereographic (North Polar)	1.554	0	0.567	0.562	3.051
Collignon	0	1.245	0.870	0.928	3.167
Behrmann	0	1.134	0.917	1.090	3.316
Lambert Cylindrical Equal-Area	0	1.324	0.819	1.181	3.821
Spilhaus Stereographic	1.573	0	1.104	1.103	4.910
Central Cylindrical	1.375	0.662	0.880	1.416	5.108

Table 5.3: Distortion Results of the high Resolution Data normalised with respect to the Arc Length using Definition 4.2.

6

Discussion

6.1. Distance Measures

This study introduces several improvements and critical assessments of existing distortion metrics used in evaluating map projections. Specifically, an improved distance measure was employed, which took points near the boundary cut into account and the fact that the projected geodesic does not necessarily become a straight line on the map. Map projections such as the gnomonic, orthographic, or Nicolosi globular projection (Figure 2.7), especially those divided into hemispheres, could potentially achieve interresting scores in distance evaluations. Though not analyzed within this research, future studies could greatly benefit from a deeper exploration of these projections, which might offer significant insights into optimal projection choices.

6.2. Revisiting Normalisation Methods

The original total distortion metric, which normalised results based on the Equirectangular projection, appeared arbitrary and biased. To address this, the current research introduced a more unbiased total distortion measure, independent of such normalisation. Unlike the previous approach (Table 3.2), which varied significantly based on the chosen reference map, the newly proposed measure consistently evaluates distortion, providing stable rankings and thus enhancing reliability.

6.3. Computational Precision and Accuracy

The enhanced computational methods significantly increased accuracy and reduced numerical errors, achieving distortion measurements precise up to three decimal places. Specifically, area and isotropy measures reached accuracies up to five decimal places. Although such extreme precision was illustrated rather than systematically applied across all projections, future research could explore whether employing such high levels of accuracy might identify different optimal projections or reveal previously undetected subtle variations in projection quality.

6.4. Fractal Dimension as a Distortion Metric

A refined dot-based box-counting method for fractal dimension calculation was implemented, significantly reducing, but not entirely eliminating, limitations inherent in previous pixel-based methods. Although fractal dimension itself was ultimately found insufficient as a distortion measure in itself due to its invariance under bi-Lipschitz transformations (excluding singularities near poles), it remains valuable for localised analyses of coastal geometry, providing insights into coastal complexity.

6.5. Theoretical Insights and Limitations

The preservation of fractal dimension under map projections was theoretically proven, highlighting previous inconsistencies reported in Table 4.1 as artifacts arising from numerical methods rather than true geometrical distortion. However, this theoretical invariance is contingent upon avoiding singularities, specifically at the poles. Future researchers should be cautious about this limitation when employing coastline data, especially if data points near polar regions are included, as it might yield misleading conclusions about fractal

dimensions. However, in this study, we successfully circumvented issues at the poles because our data did not include points defined exactly at these singularities, allowing for robust theoretical analyses and reliable results.

6.6. Importance of Proper Normalisation

Normalisation with respect to coastline arc-length was executed to ensure accurate comparison across projections. However, due to time constraints, normalisation was uniformly based on the Equirectangular projection rather than individually tailored. Future work should aim to normalise each projection individually based on its own coastline, which could lead to more precise and meaningful comparative analyses.

6.7. Sensitivity to Globe Orientation

The high performance of the Cassini projection underscores the problem of orientation sensitivity clearly; despite essentially being identical to the Equirectangular projection (involving a rotation of the globe before applying the same transformation) it scores remarkably well, whereas the Equirectangular projection itself does not. This observation indicates potential improvements if other projections were similarly rotated or adjusted before projection. Thus, the current optimal projection (Azimuthal Equidistant) might be surpassed by other configurations not yet explored, highlighting the importance of considering geographic orientation in future research.

Figure 6.1: Cassini Projection. Image by Strebe, licensed under CC BY-SA 4.0 via Wikimedia Commons.

6.8. Opportunities for Further Research

Future research could significantly extend this work by analysing additional resolution levels from databases such as GSHHG. Moreover, this research emphasises a fundamental conceptual shift: the importance of evaluating map projections based explicitly on the geographic properties they represent. For instance, a map designed solely to depict Dutch states should primarily evaluate distortion effects within the Netherlands, rendering distortions of states in Belgium irrelevant. Similarly, a map emphasising political boundaries should specifically assess distortions along those boundaries rather than irrelevant geographic features. This perspective introduces a generalised and adaptable method that can evaluate any map projection based on its intended purpose and specific content. Such an approach allows cartographers and researchers to conduct property-specific distortion analyses, tailoring evaluations precisely to the needs of the intended audience or application.

Furthermore, projections with multiple boundary cuts represents an important avenue for future exploration. Projections such as the Boggs eumorphic, Fuller, Dymaxion, Waterman butterfly, and HEALPix pos-

sess complex boundary cuts, making them difficult to evaluate using traditional distortion measures. During this research, attempts were made to analyse the HEALPix projection using numerical distortion estimation. These initial results showed great potential and even outperformed the Winkel Tripel projection. However, due to limitations in reliability and reproducibility, these findings were not included in the final results. Overcoming these computational challenges could significantly broaden the practical applicability and effectiveness of distortion analysis, and improve maps.

7

Conclusion

The findings of this research significantly improved distortion analysis of map projections previously introduced by Goldberg and Gott, refining it into a more reliable and unbiased measure. By implementing this improved methodology, the Winkel Tripel projection was identified as the overall best-performing projection for general-purpose use.

A central part of this study involved a critical examination of the potential for the fractal dimension of coastlines to serve as a new distortion metric. The research concludes through theoretical analysis that fractal dimension is unsuitable as a global distortion metric since it is invariance under the bi-Lipschitz transformations that define many map projections. However, the investigation affirmed its utility for localised geometric analysis.

This inquiry led to a shift in focus: evaluating distortions specifically along coastal features. The study introduces a novel normalisation approach that uses coastline arc length, establishing a robust framework for evaluating projections tailored to coastal representation. This coastline-specific analysis, conducted at multiple resolutions, revealed that the Azimuthal Equidistant projection is the most effective for this particular application.

Furthermore, the results underscore the sensitivity of distortion scores to the orientation of the globe, indicating that performance can be optimised through rotational adjustments. Ultimately, this study provides a foundational methodology for future, property-specific distortion analyses, offering a more precise and application-focused approach to selecting map projections.

Bibliography

[1] Jimmy Briggs and Tim Tyree. Hausdorff measure, 2016.

[2] Carl Friedrich Gauss. Allgemeine flächentheorie. *Commentationes Societatis Regiae Scientiarum Gottingensis Recentiores*, 6:99–146, 1827. Also in: Gauss, Werke, Vol. 4, pp. 217–258, Königliche Gesellschaft der Wissenschaften, Göttingen, 1903.

[3] David M Goldberg and J Richard Gott III. Flexion and skewness in map projections of the earth. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 42(4):297–318, 2007.

[4] J Richard Gott III, David M Goldberg, and Robert J Vanderbei. Flat maps that improve on the winkel tripel. *arXiv preprint arXiv:2102.08176*, 2021.

[5] Felix Hausdorff. Dimension und äußeres maß. *Mathematische Annalen*, 79(1):157–179, 1918.

[6] Melita Kennedy, Steve Kopp, et al. *Understanding map projections*, volume 8. Esri Redlands, CA, 2000.

[7] Krisztián Kerkovits. Calculation and visualization of flexion and skewness. *Kartogr. I Geoinf*, 17:32–45, 2018.

[8] Paul H. Laskowski. Map distortions and singular value decomposition. In *Technical Papers, ASPRS-ACSM Annual Convention*, volume 4, pages 42–51, Baltimore, March 1987. March 29–April 3, 1987.

[9] Piotr H Laskowski. The traditional and modern look at tissot's indicatrix. *The American Cartographer*, 16(2):123–133, 1989.

[10] Franklin Lee. Computational quantification of map projection distortion by fractal dimension of coastlines. *Journal of Applied Mathematics and Physics*, 12(5):1890–1903, 2024.

[11] Benoit Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *science*, 156(3775):636–638, 1967.

[12] Benoit B Mandelbrot. The fractal geometry of nature/revised and enlarged edition. *New York*, 1983.

[13] M.C.Bukman. Map projections. `https://resolver.tudelft.nl/uuid:faada51d-b08c-4c48-85d8-92d305f6d7bc`, 2024.

[14] Vikash Mittal. Geometric phase and its applications: topological phases, quantum walks and non-inertial quantum systems. *arXiv preprint arXiv:2209.04810*, 2022.

[15] Lewis F Richardson. The problem of contiguity: an appendix to statistics of deadly quarrels. *General systems yearbook*, 6:139–187, 1961.

[16] Dierk Schleicher. Hausdorff dimension, its properties, and its surprises. *The American Mathematical Monthly*, 114(6):509–528, 2007.

[17] John P. Snyder. *Map Projections: A Working Manual*, volume 1395 of *U.S. Geological Survey Professional Paper*. U.S. Government Printing Office, Washington, D.C., 1987. ISBN 9780608042494. URL `https://pubs.usgs.gov/pp/1395/report.pdf`.

[18] Nicolas Auguste Tissot. Mémoire sur la représentation des surfaces et les projections des cartes géographiques. *Mémoires présentés par divers savants à l'Académie des Sciences de l'Institut National de France, Série 2*, 19:1–105, 1881.

[19] Jiaxin Wu, Xin Jin, Shuo Mi, and Jinbo Tang. An effective method to compute the box-counting dimension based on the mathematical definition and intervals. *Results in Engineering*, 6:100106, 2020.

A

Heatmaps

A.1. Aitoff

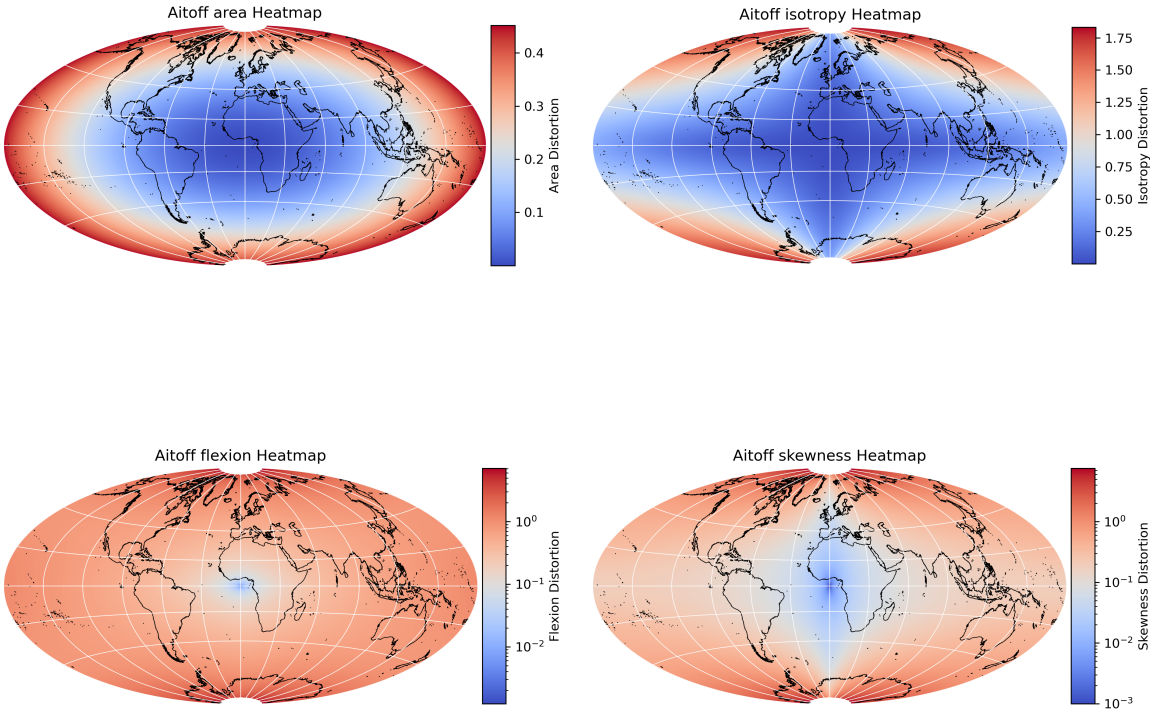


Figure A.1: Aitoff projection – Area, Isotropy, Flexion, Skewness

A.2. Azimuthal Equidistant (North Polar)

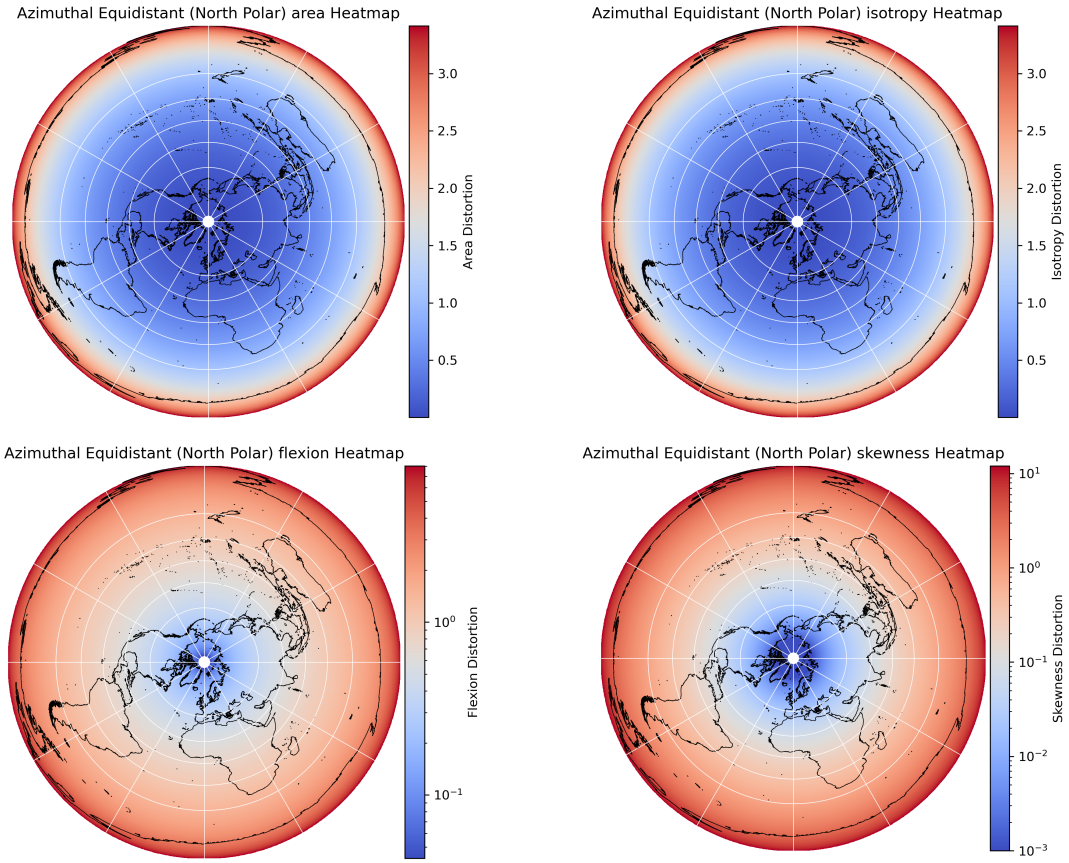


Figure A.2: Azimuthal Equidistant (North Polar) projection – Area, Isotropy, Flexion, Skewness

A.3. Behrmann

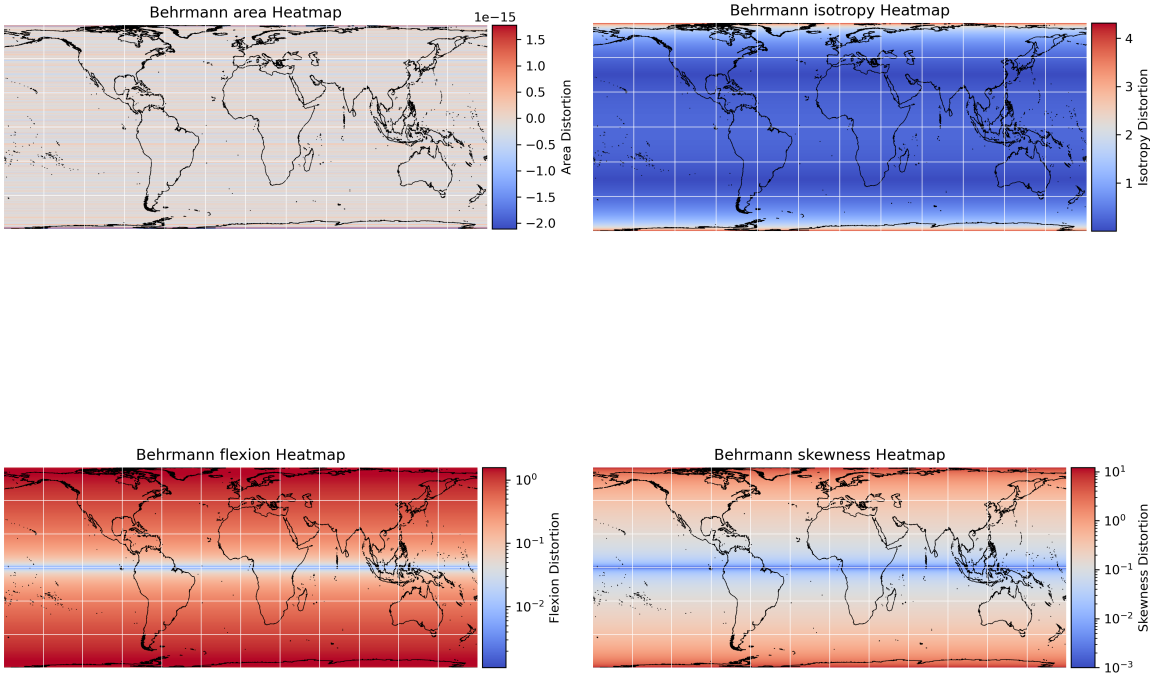


Figure A.3: Behrmann projection – Area, Isotropy, Flexion, Skewness

A.4. Cassini

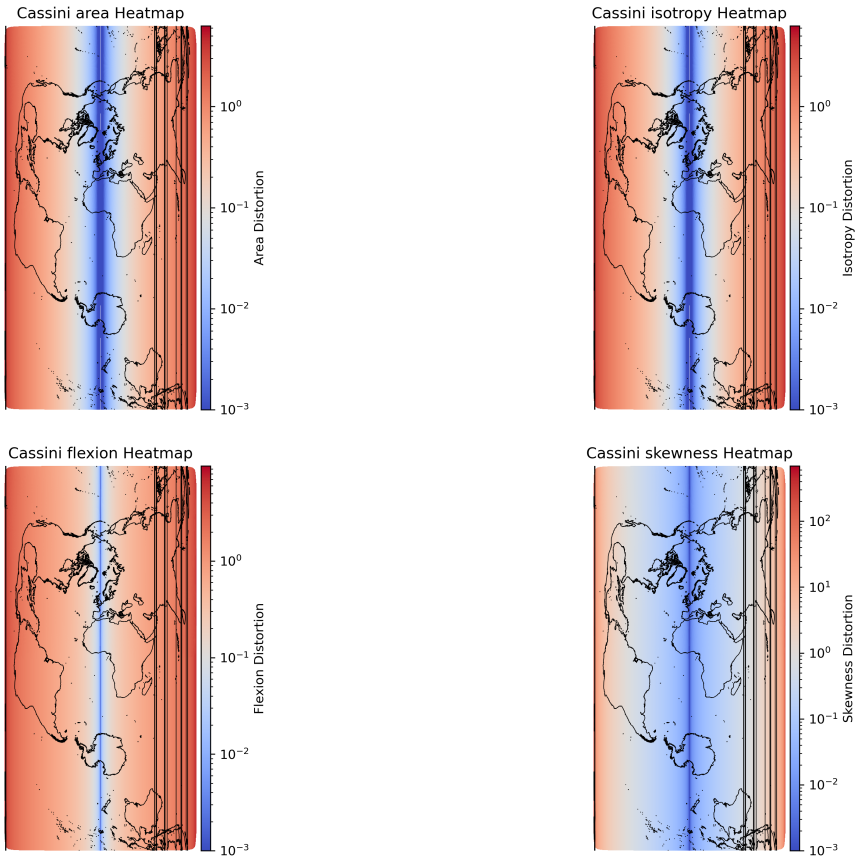


Figure A.4: Cassini projection – Area, Isotropy, Flexion, Skewness

A.5. Central Cylindrical

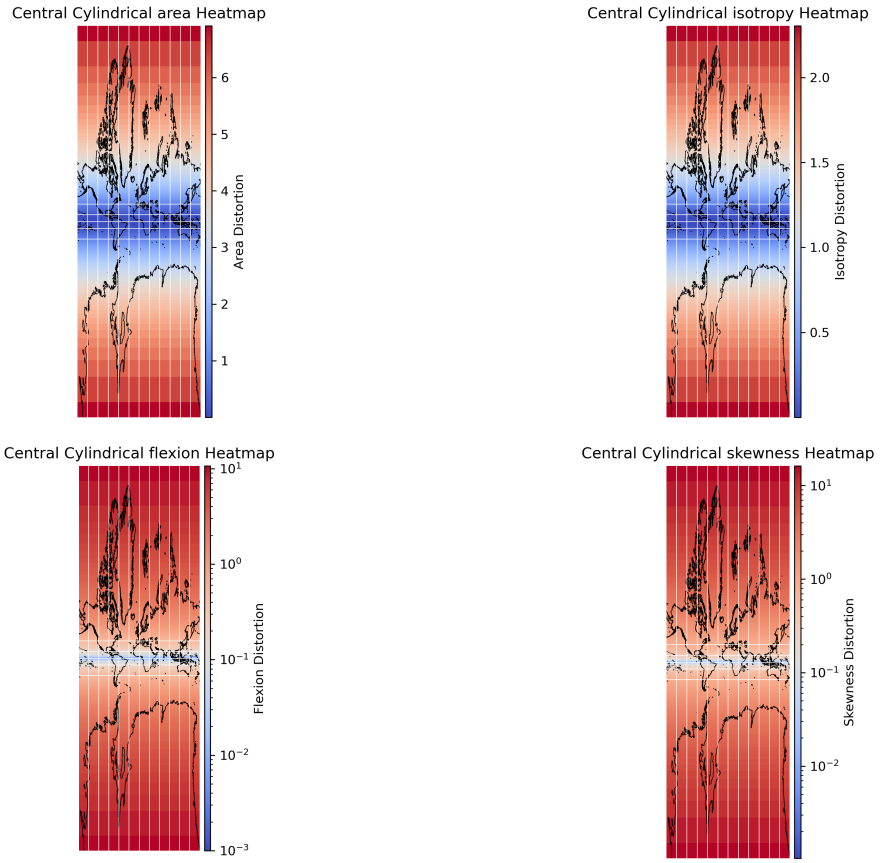


Figure A.5: Central Cylindrical projection – Area, Isotropy, Flexion, Skewness

A.6. Collignon

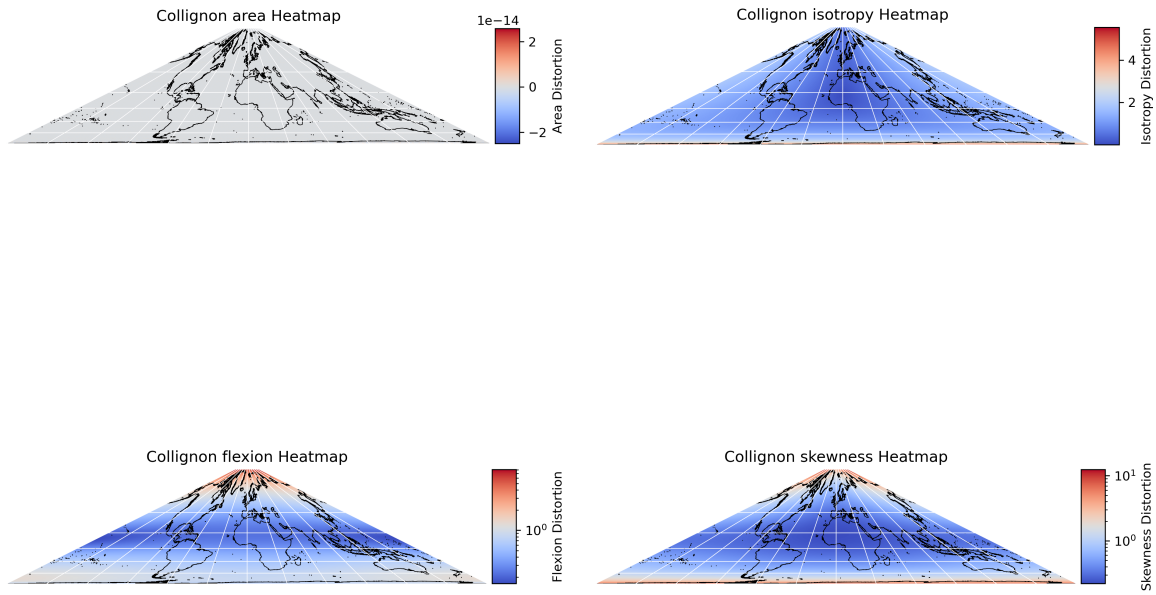


Figure A.6: Collignon projection – Area, Isotropy, Flexion, Skewness

A.7. Equidistant Conic

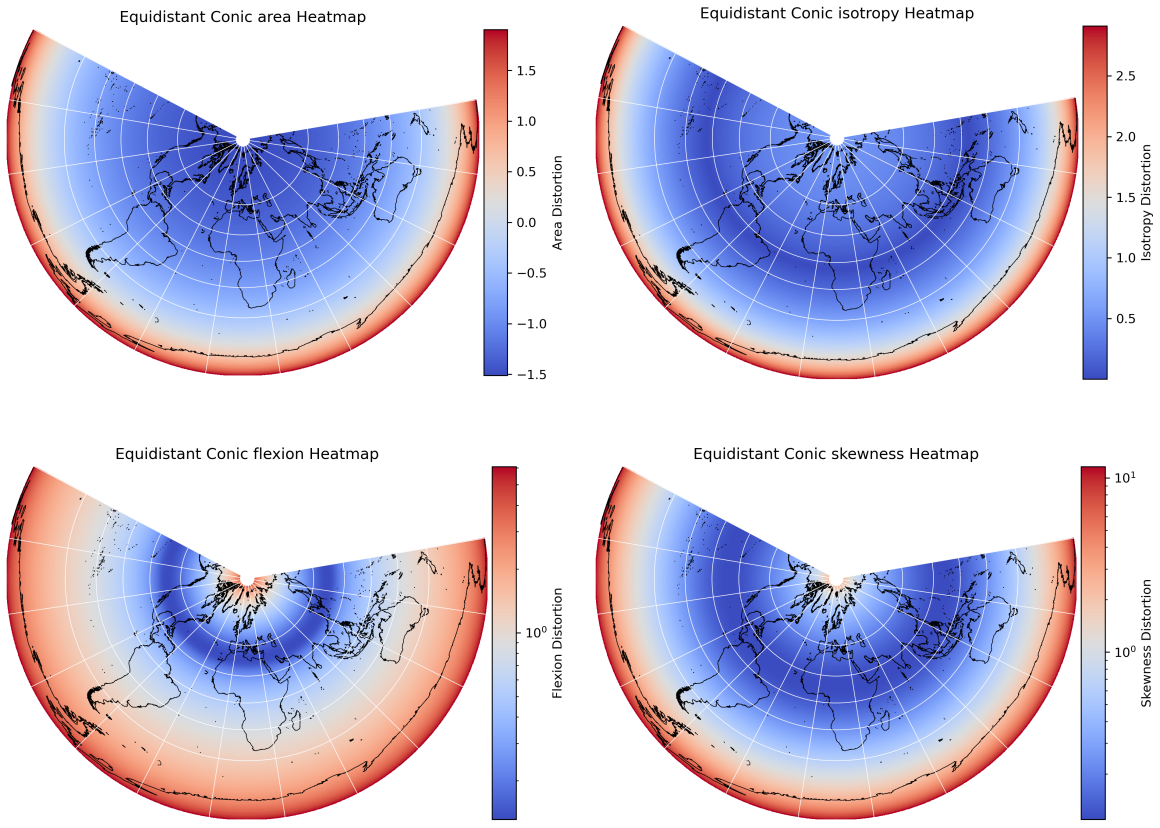


Figure A.7: Equidistant Conic projection – Area, Isotropy, Flexion, Skewness

A.8. Equirectangular

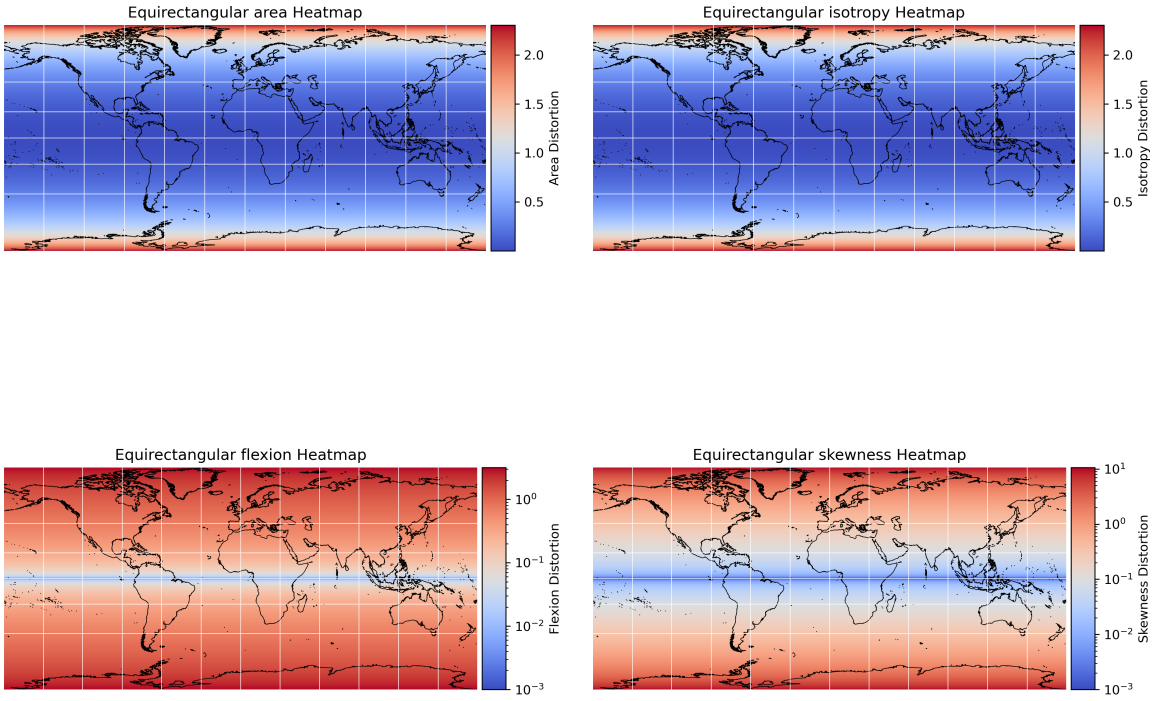


Figure A.8: Equirectangular projection – Area, Isotropy, Flexion, Skewness

A.9. Gall-Peters

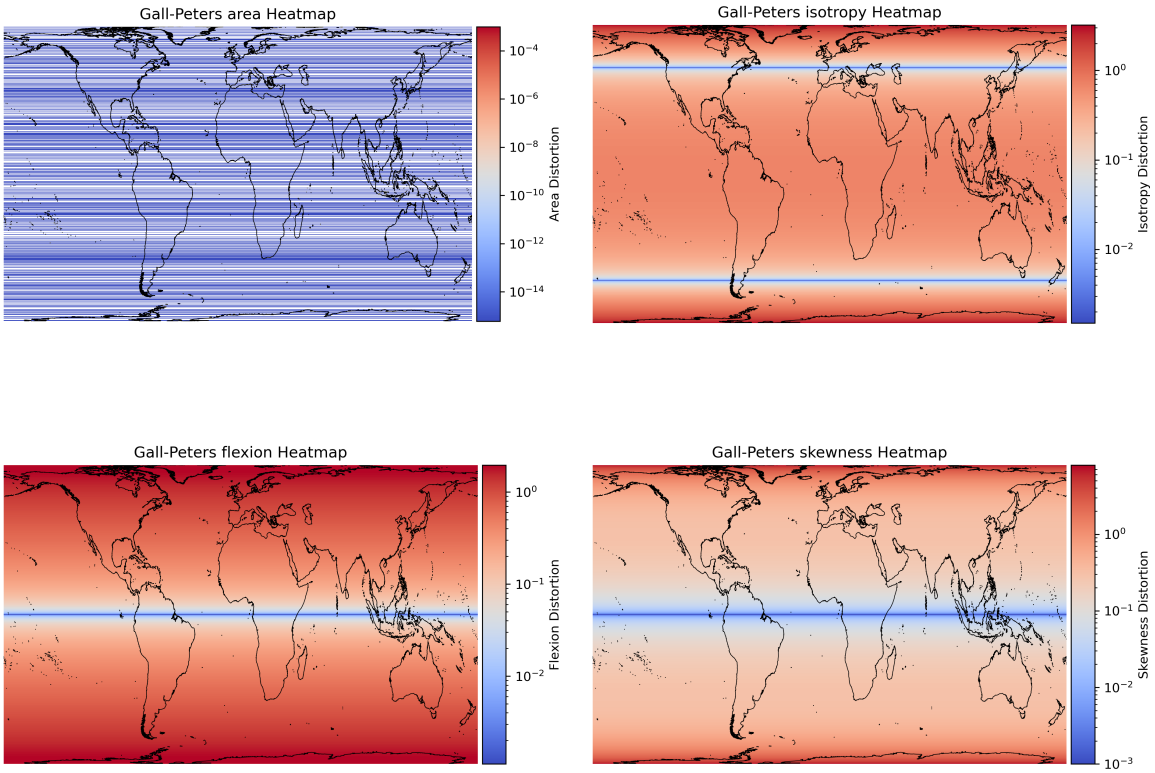


Figure A.9: Gall-Peters projection – Area, Isotropy, Flexion, Skewness

A.10. Hammer

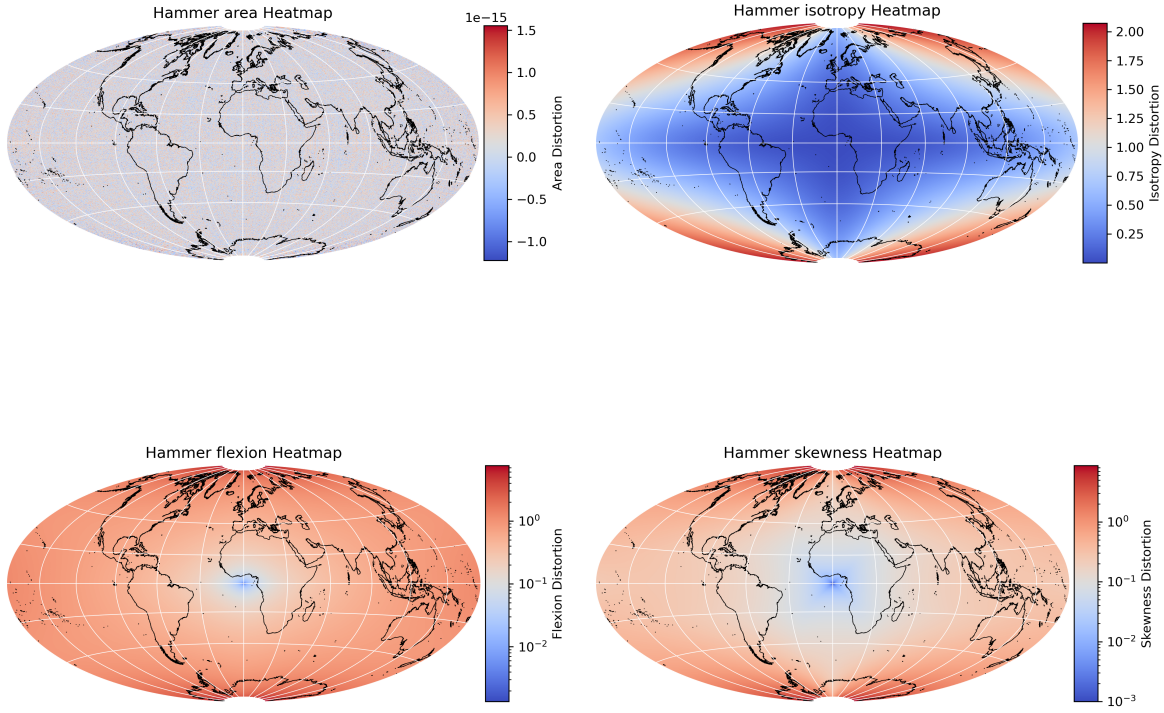


Figure A.10: Hammer projection – Area, Isotropy, Flexion, Skewness

A.11. Lambert Azimuthal Equal-Area (North Polar)

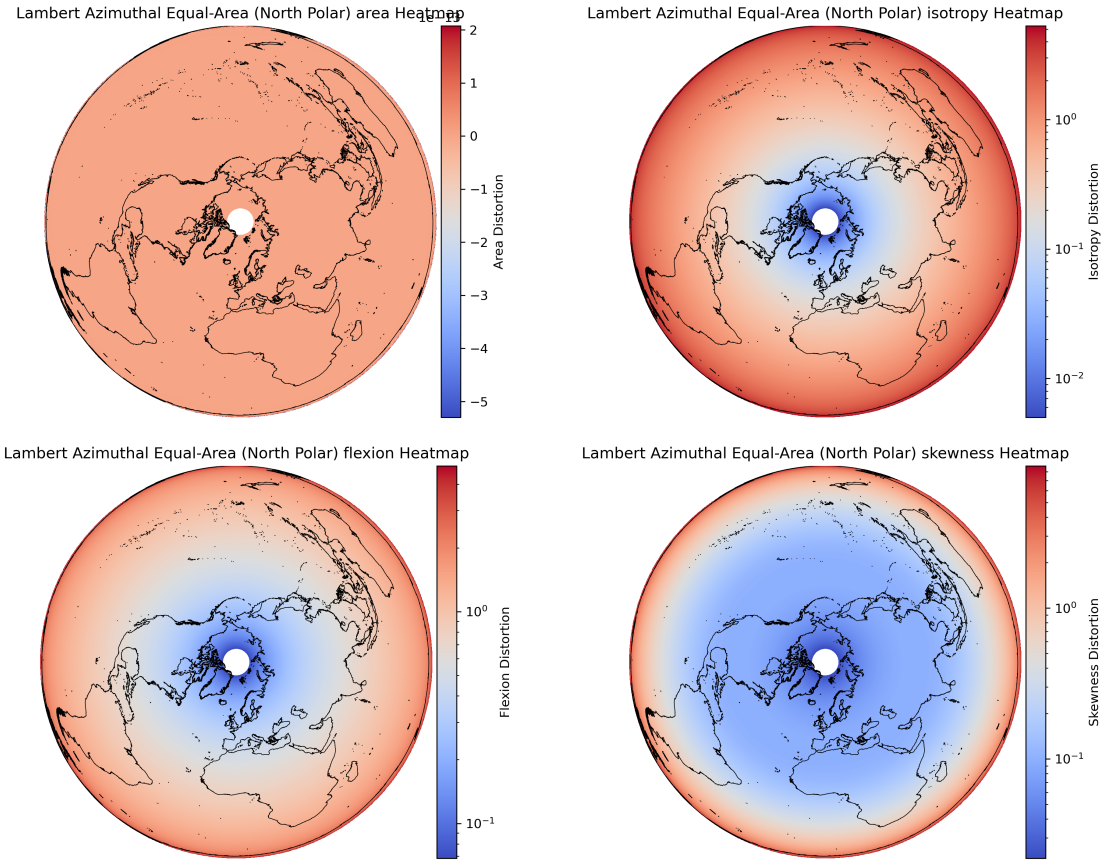


Figure A.11: Lambert Azimuthal Equal-Area (North Polar) projection – Area, Isotropy, Flexion, Skewness

A.12. Lambert Conic

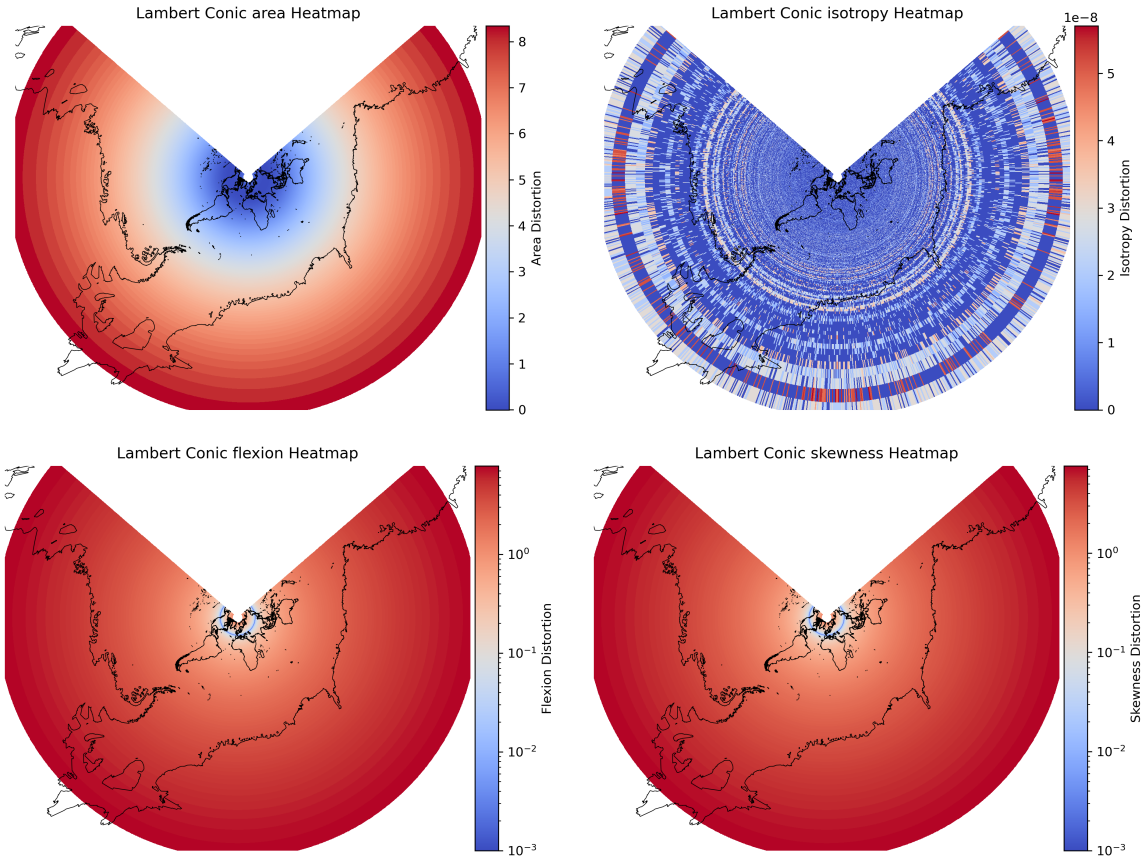


Figure A.12: Lambert Conic projection – Area, Isotropy, Flexion, Skewness

A.13. Lambert Cylindrical Equal-Area

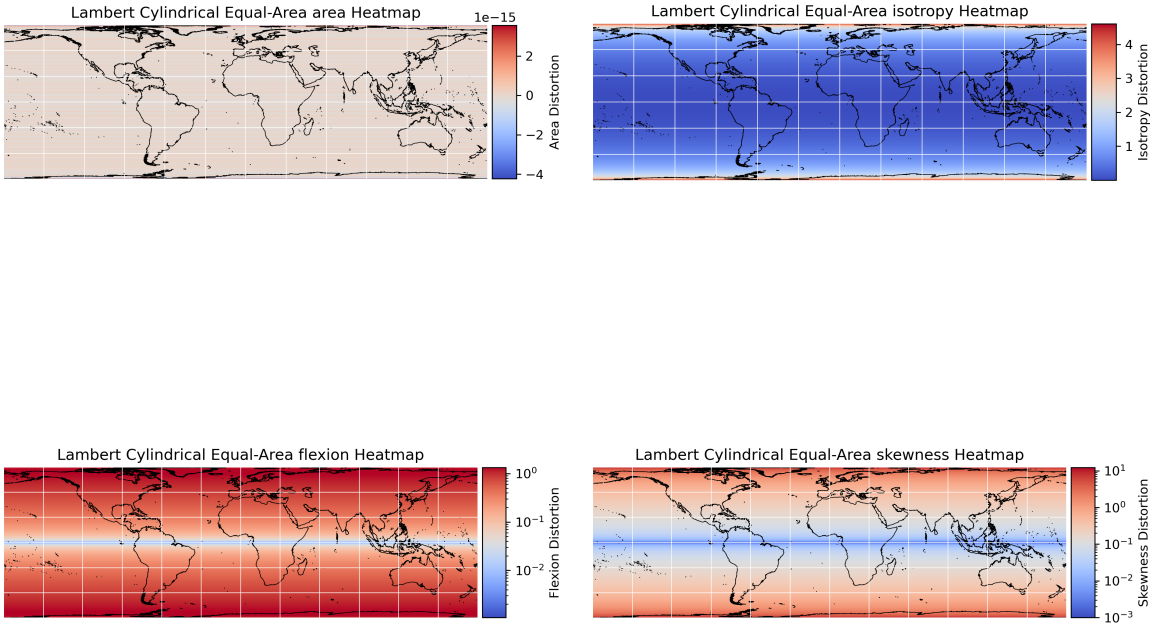


Figure A.13: Lambert Cylindrical Equal-Area projection – Area, Isotropy, Flexion, Skewness

A.14. Miller Cylindrical

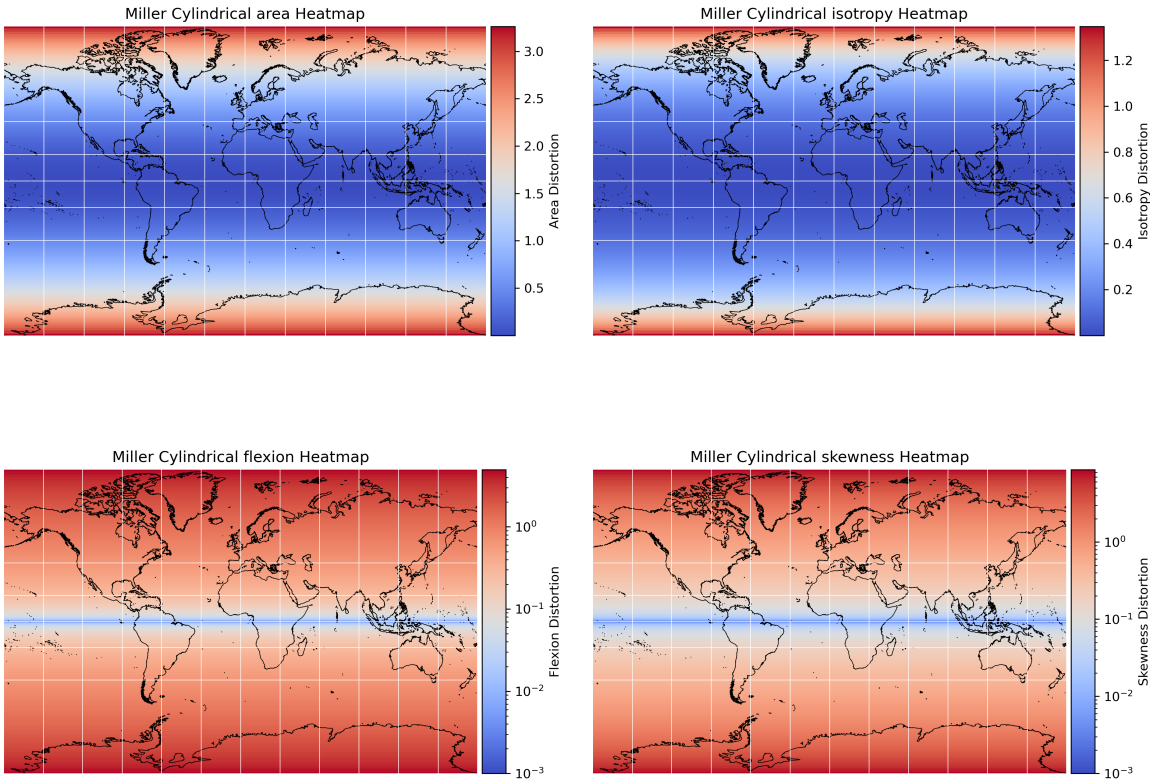


Figure A.14: Miller Cylindrical projection – Area, Isotropy, Flexion, Skewness

A.15. Sinusoidal

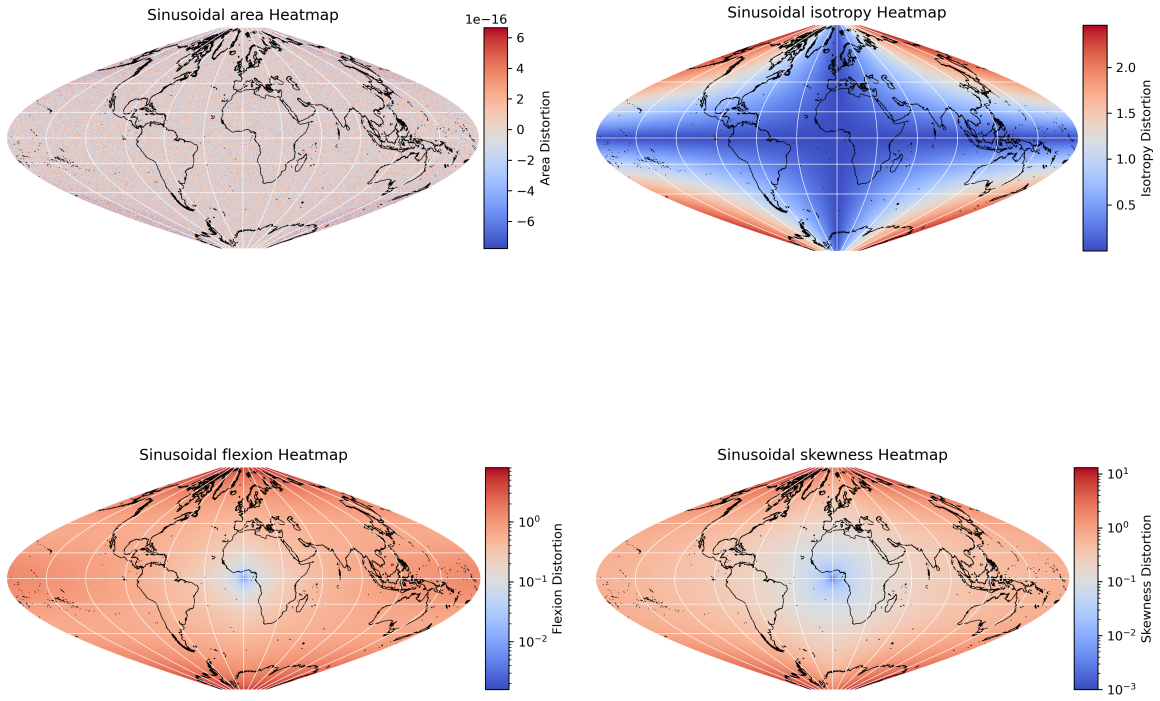


Figure A.15: Sinusoidal projection – Area, Isotropy, Flexion, Skewness

A.16. Stereographic (North Polar)

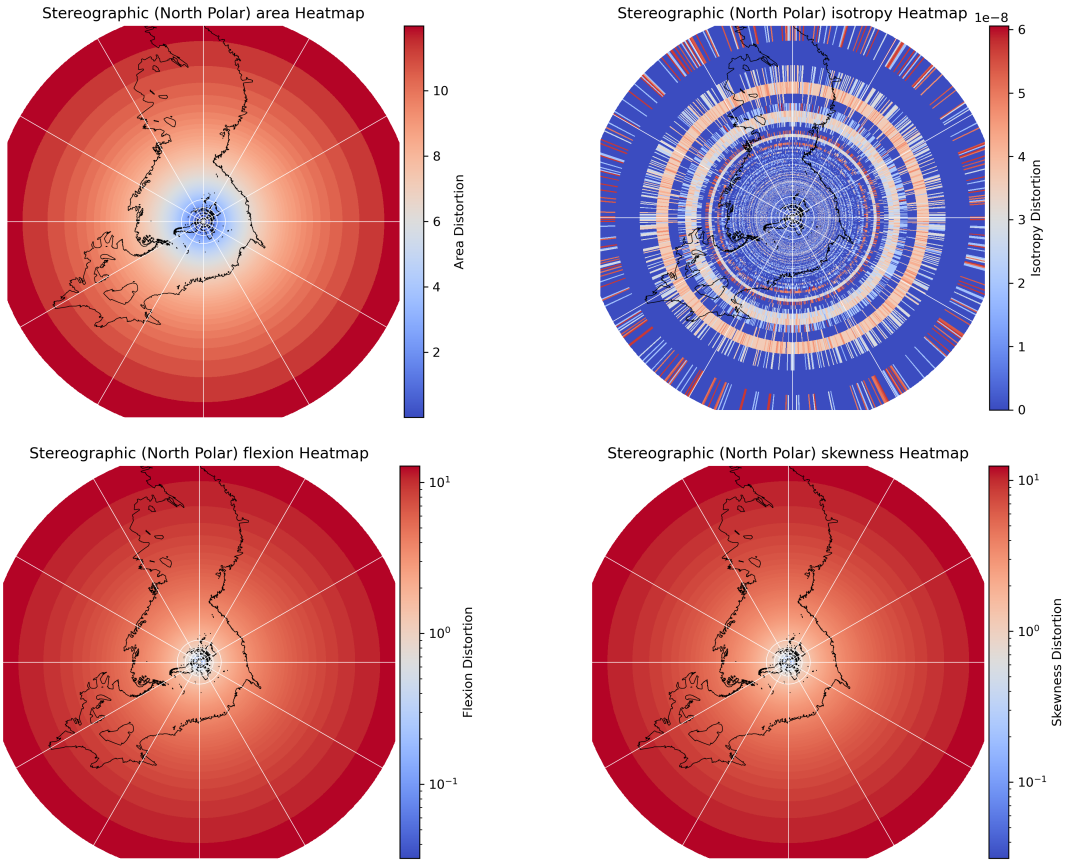


Figure A.16: Stereographic (North Polar) projection – Area, Isotropy, Flexion, Skewness

A.17. Wiechel

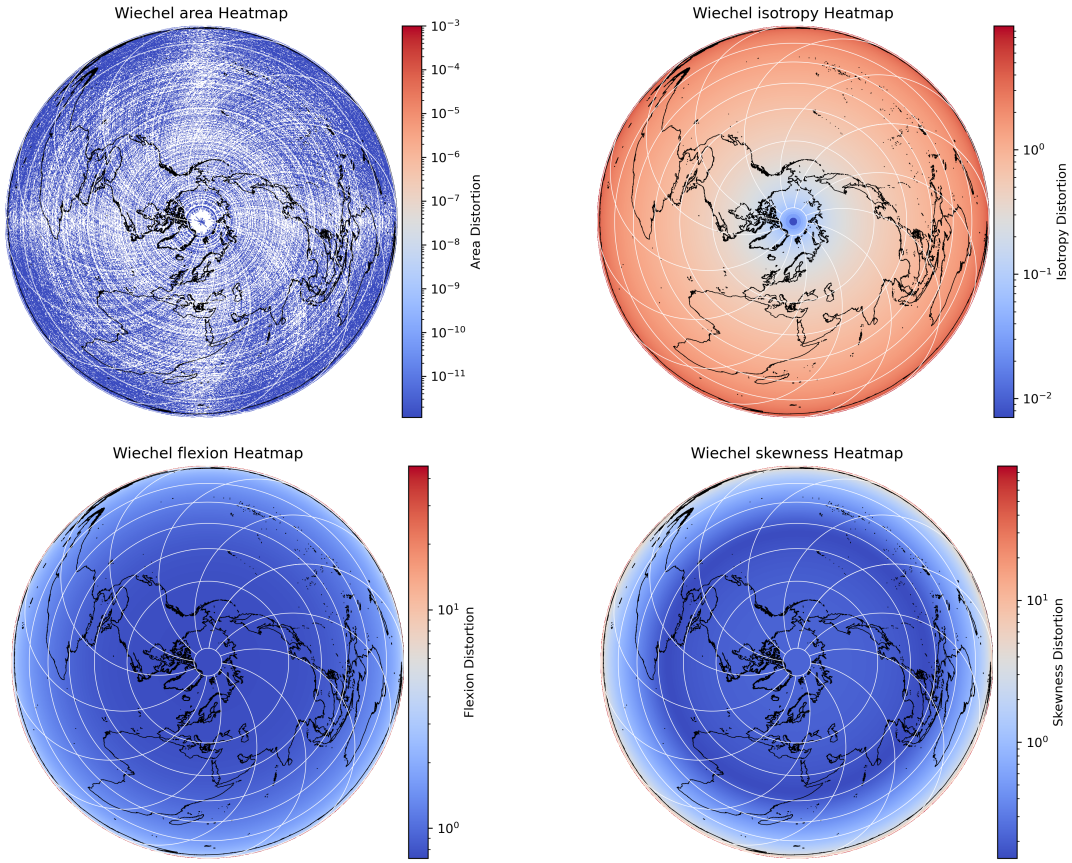


Figure A.17: Wiechel projection – Area, Isotropy, Flexion, Skewness

A.18. Winkel II

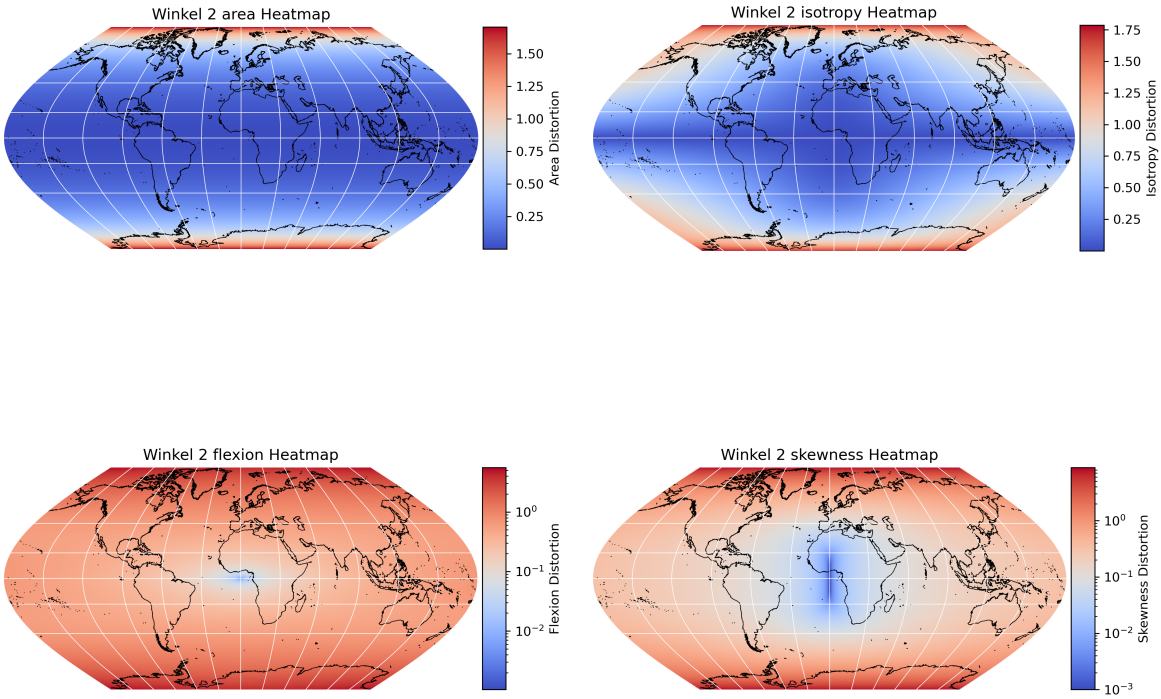


Figure A.18: Winkel II projection – Area, Isotropy, Flexion, Skewness

A.19. Winkel Tripel

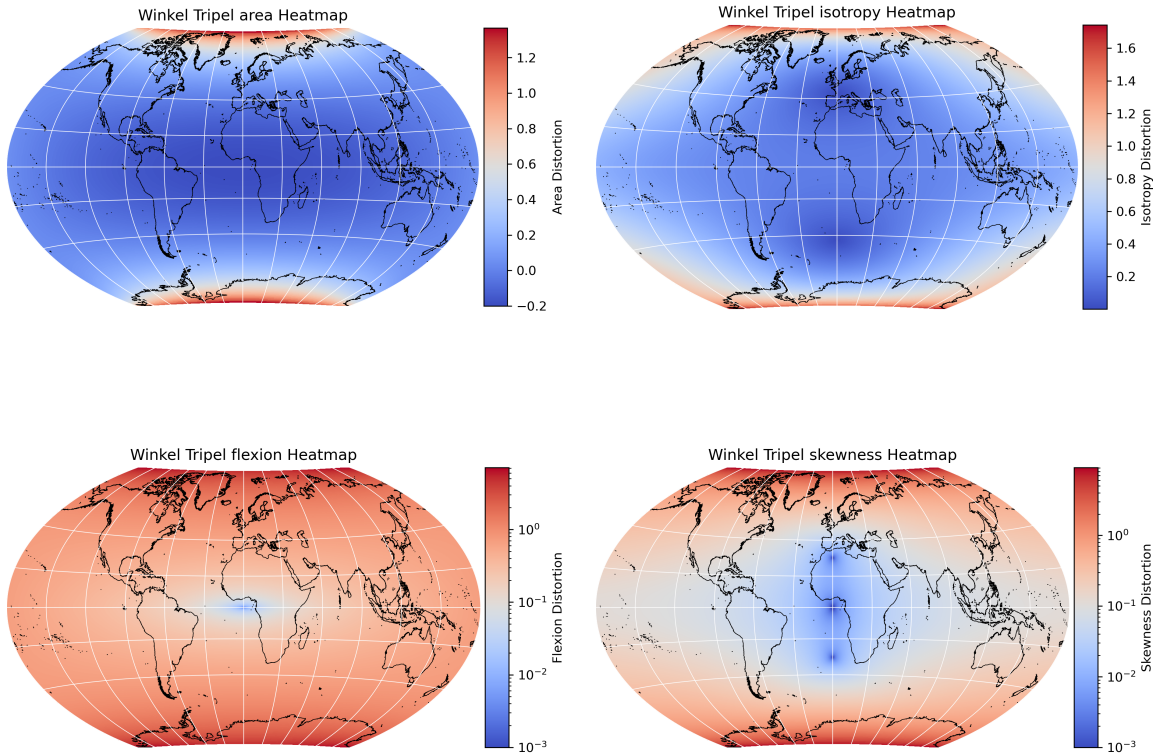


Figure A.19: Winkel Tripel projection – Area, Isotropy, Flexion, Skewness

B

Distance Plots

B.1. Aitoff

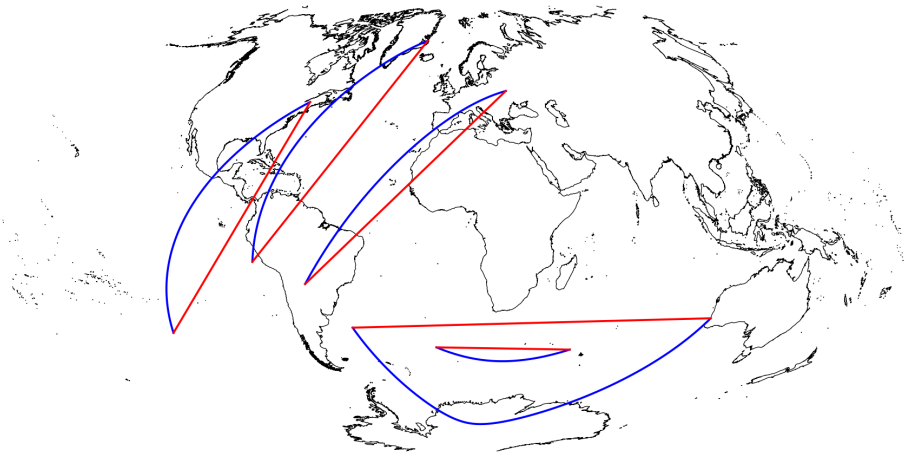


Figure B.1: Aitoff

B.2. Behrmann

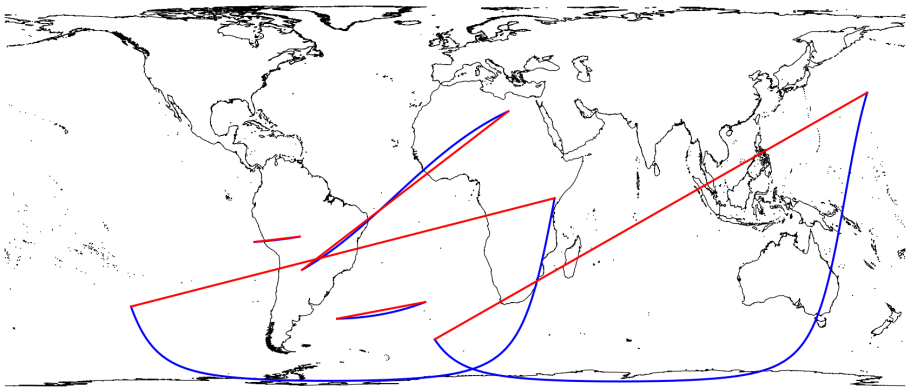


Figure B.2: Behrmann

B.3. Cassini

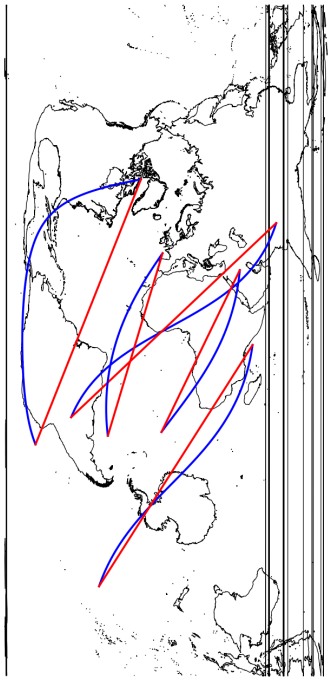


Figure B.3: Cassini

B.4. Central Cylindrical

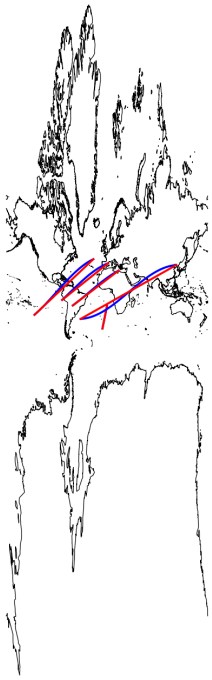


Figure B.4: Central Cylindrical

B.5. Collignon

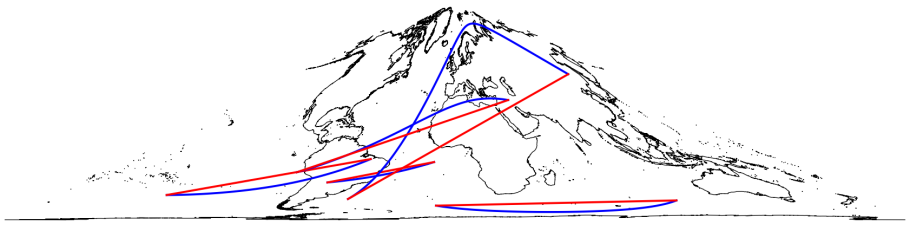


Figure B.5: Collignon

B.6. Equidistant Conic

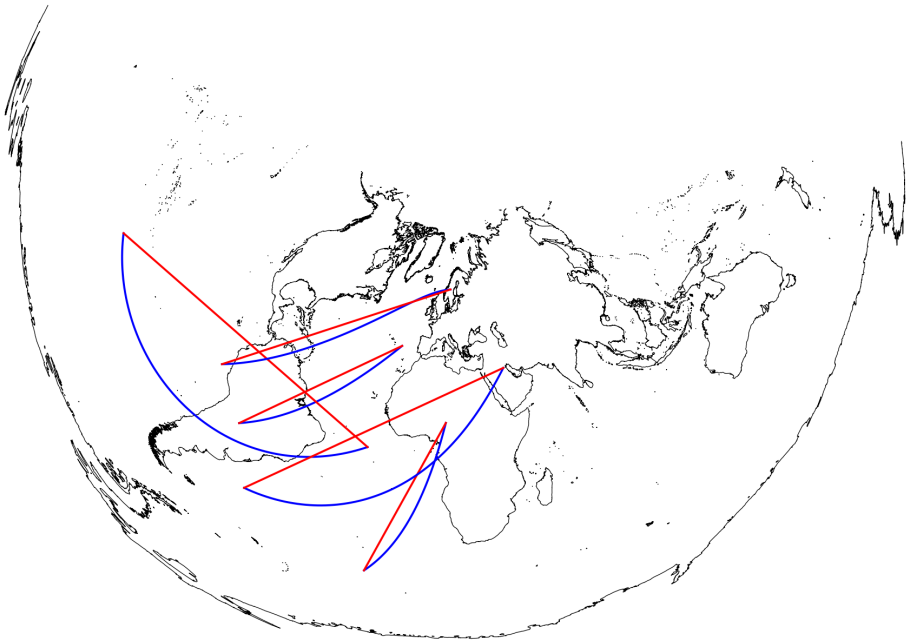


Figure B.6: Equidistant Conic

B.7. Equirectangular

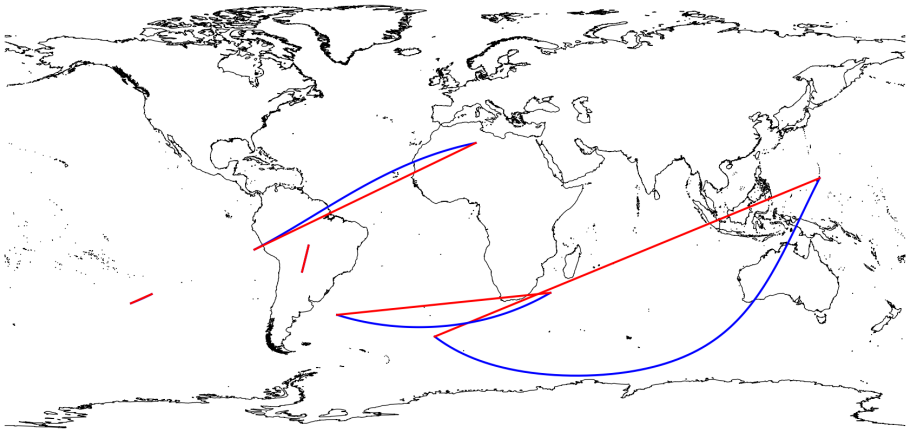


Figure B.7: Equirectangular

B.8. Gall-Peters

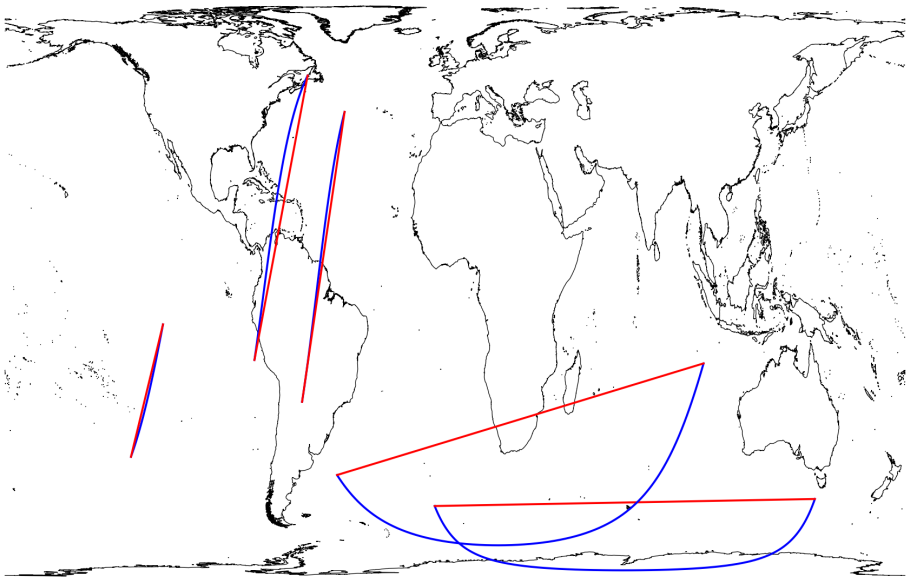


Figure B.8: Gall-Peters

B.9. Hammer

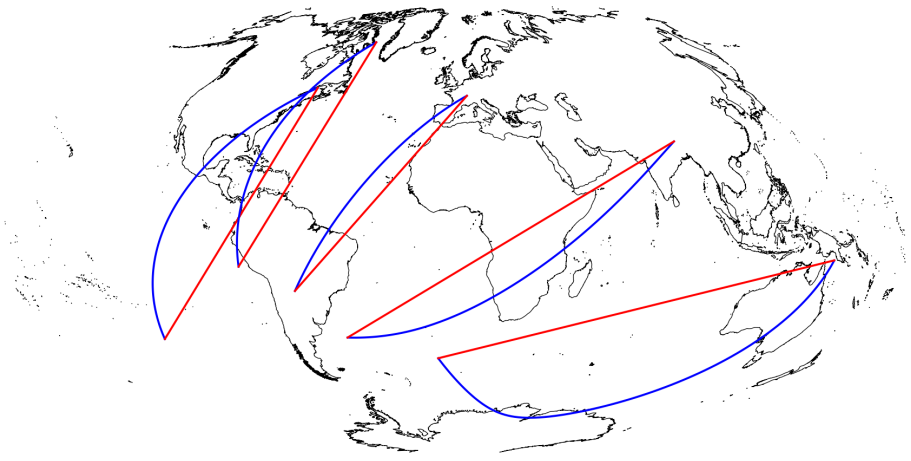


Figure B.9: Hammer

B.10. Lambert Azimuthal Equal-Area (North Polar)

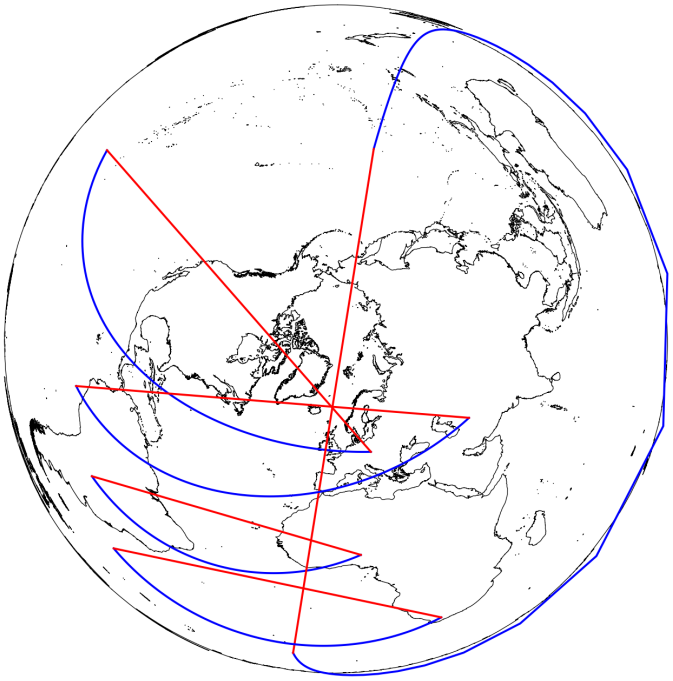


Figure B.10: Lambert Azimuthal Equal-Area (North Polar)

B.11. Lambert Conic

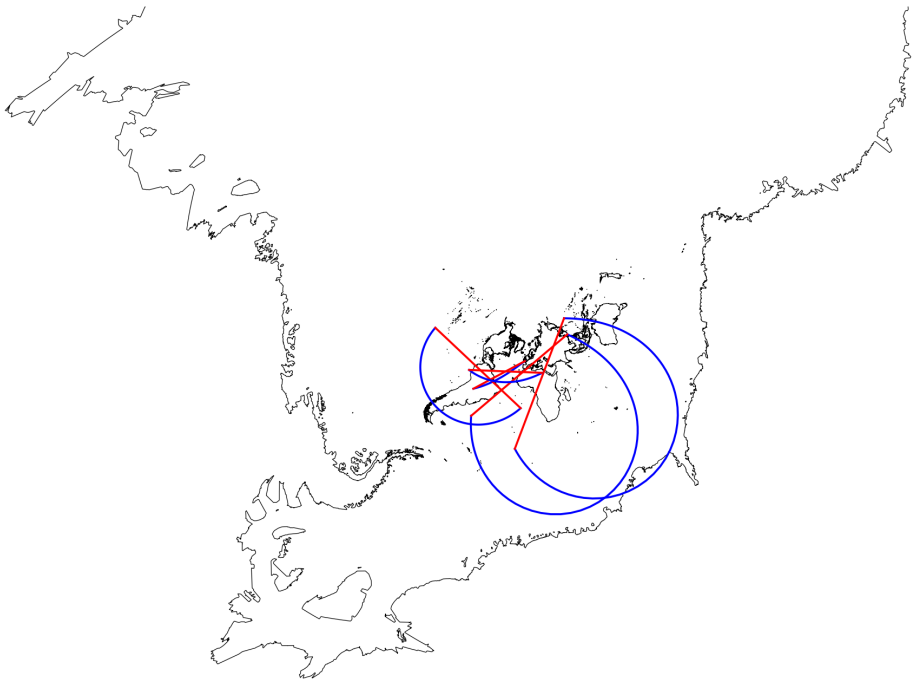


Figure B.11: Lambert Conic

B.12. Lambert Cylindrical Equal-Area

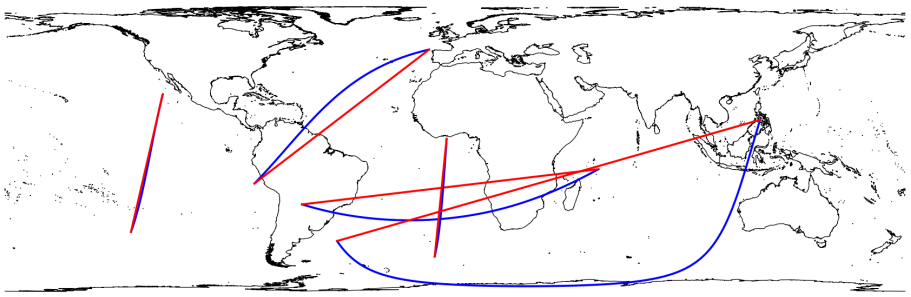


Figure B.12: Lambert Cylindrical Equal-Area

B.13. Miller Cylindrical

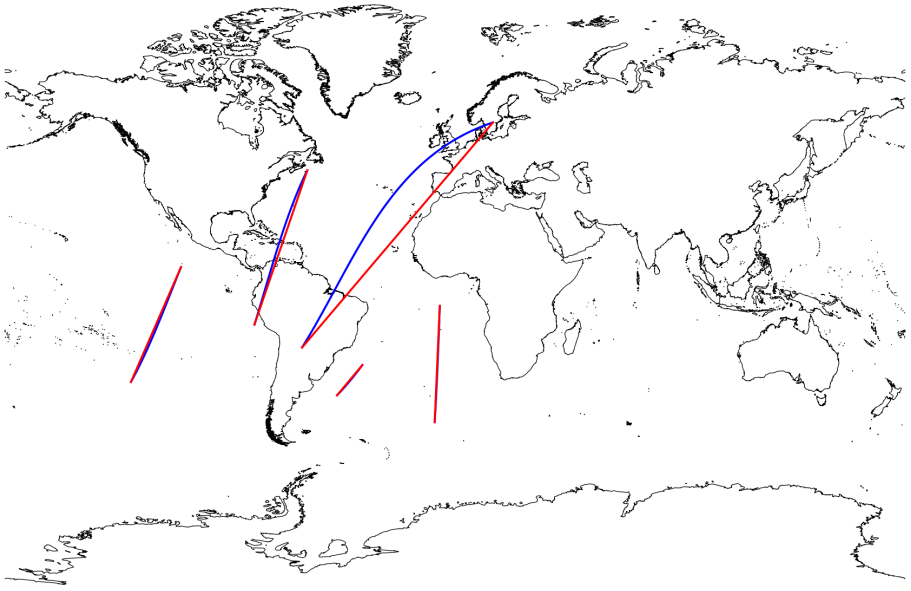


Figure B.13: Miller Cylindrical

B.14. Sinusoidal

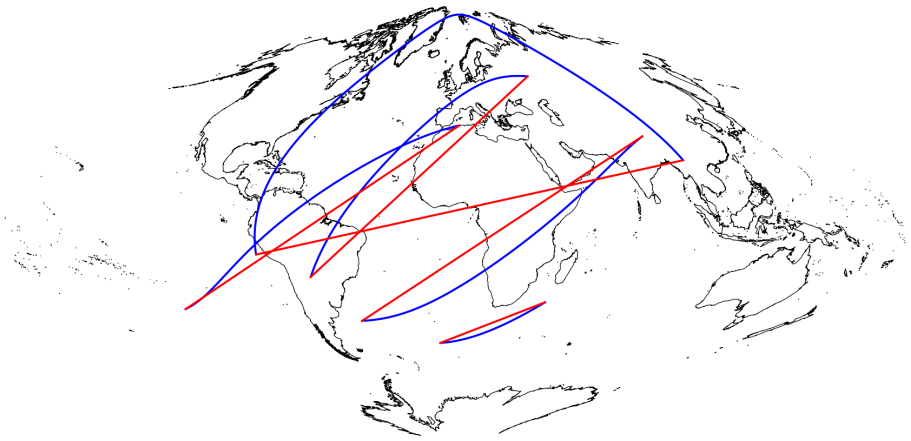


Figure B.14: Sinusoidal

B.15. Spilhaus Stereographic

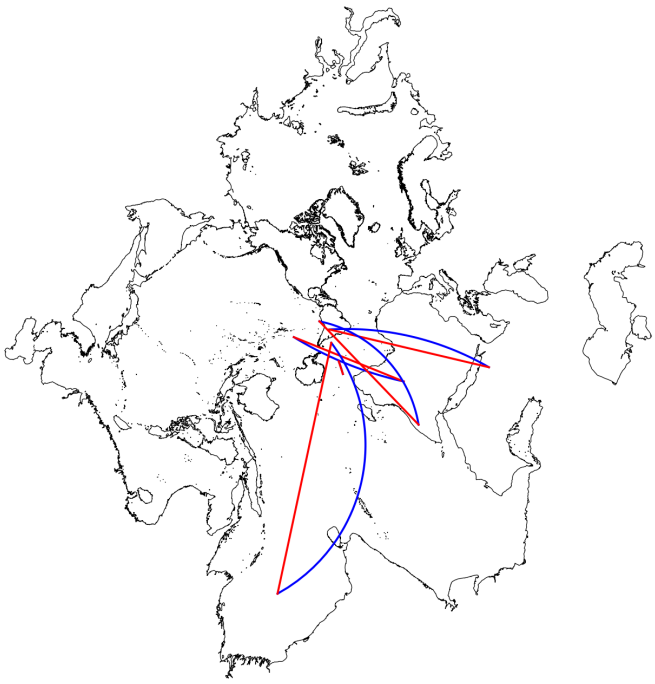


Figure B.15: Spilhaus Stereographic

B.16. Stereographic (North Polar)

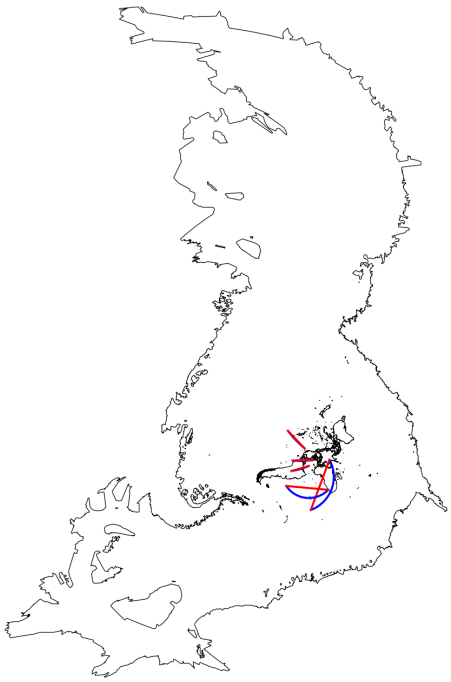


Figure B.16: Stereographic (North Polar)

B.17. Winkel II

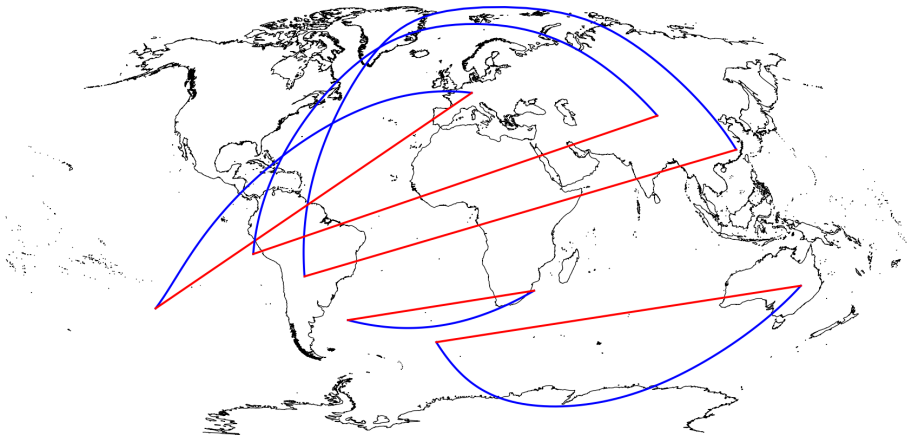


Figure B.17: Winkel II

B.18. Winkel Tripel

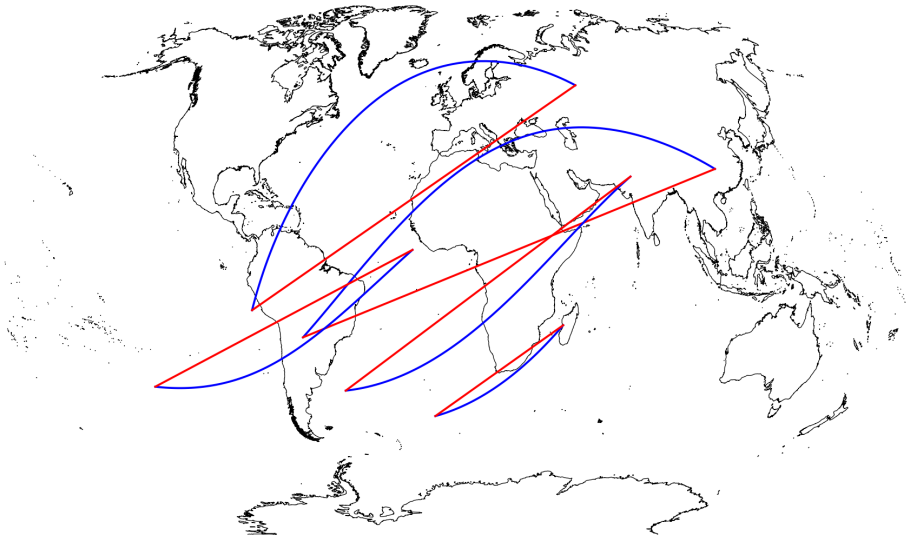


Figure B.18: Winkel Tripel

C

Error Plots

C.1. Aitoff

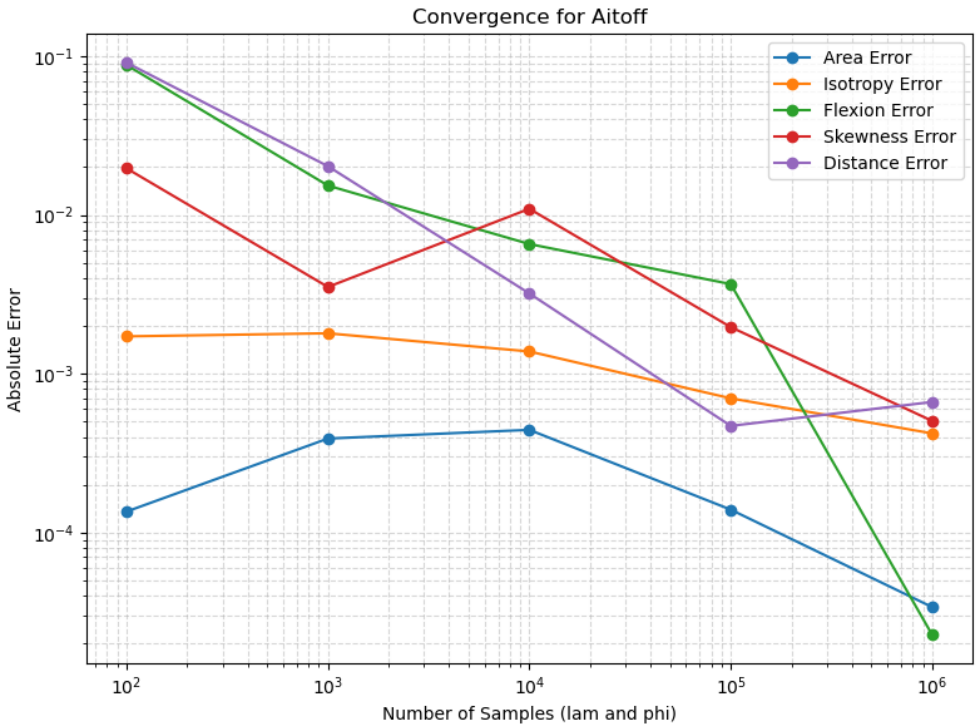


Figure C.1: Aitoff

C.2. Behrmann

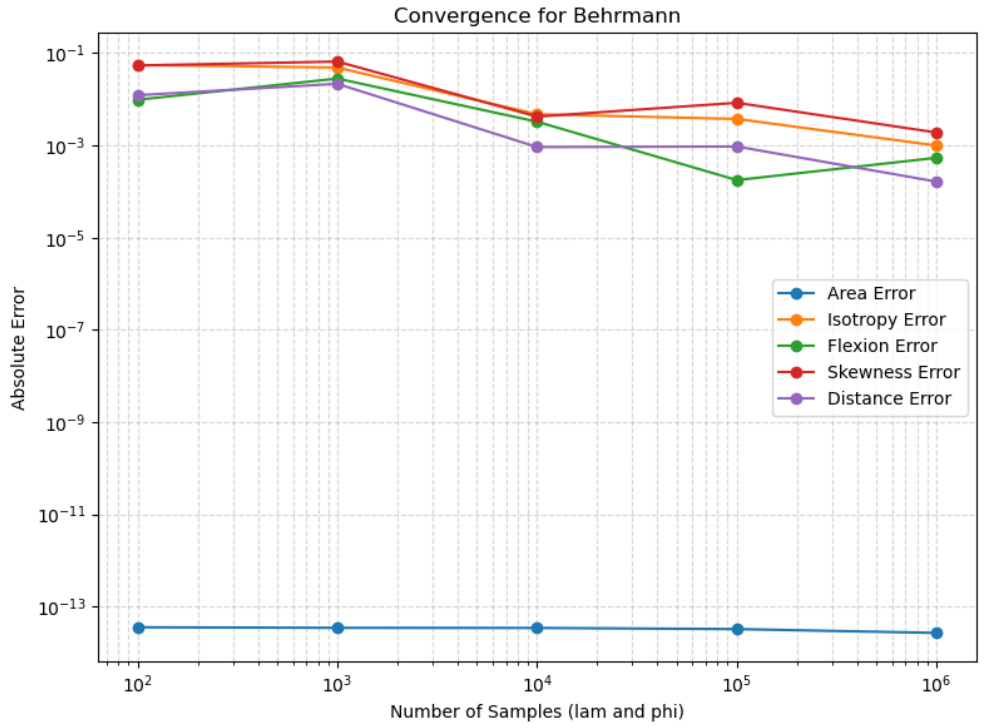


Figure C.2: Behrmann

C.3. Bonne

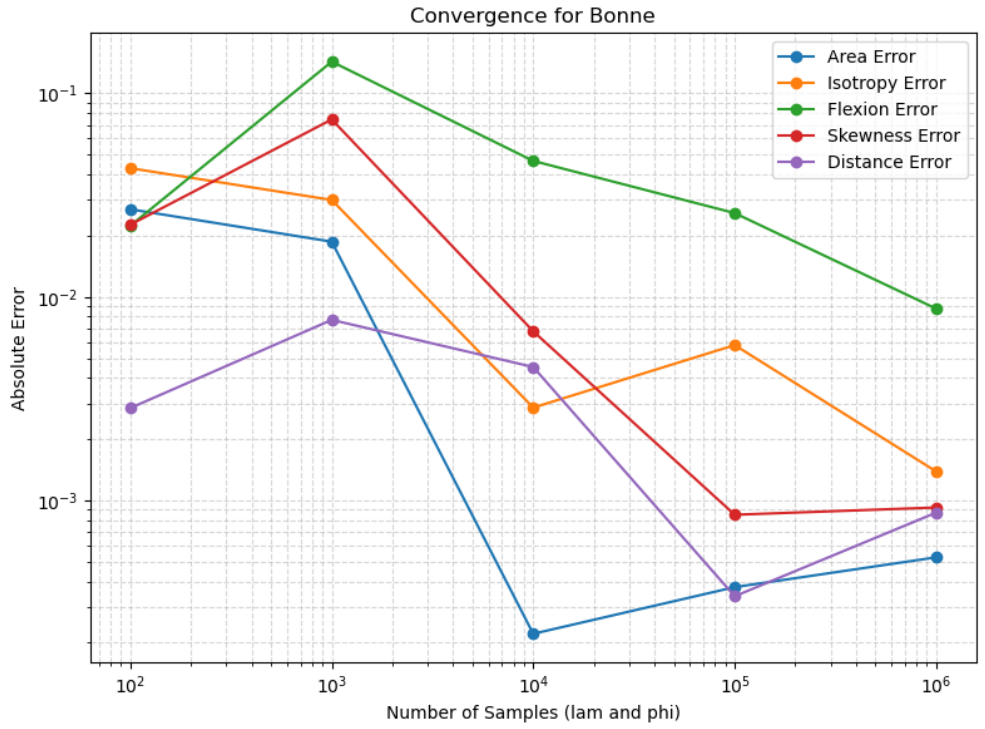


Figure C.3: Bonne

C.4. Cassini

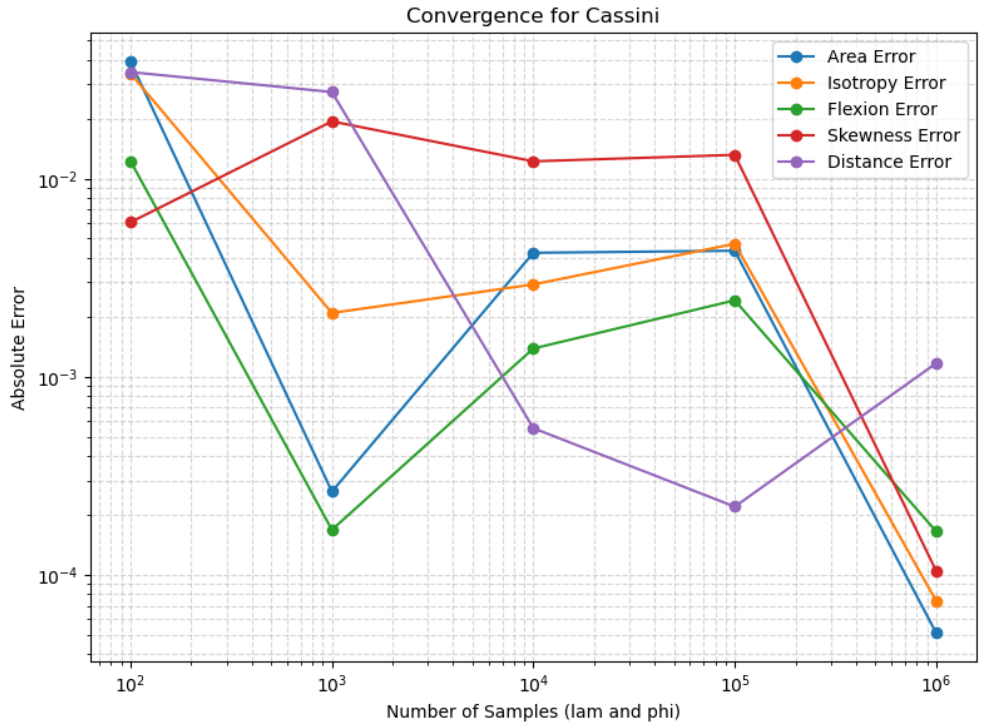


Figure C.4: Cassini

C.5. Central Cylindrical

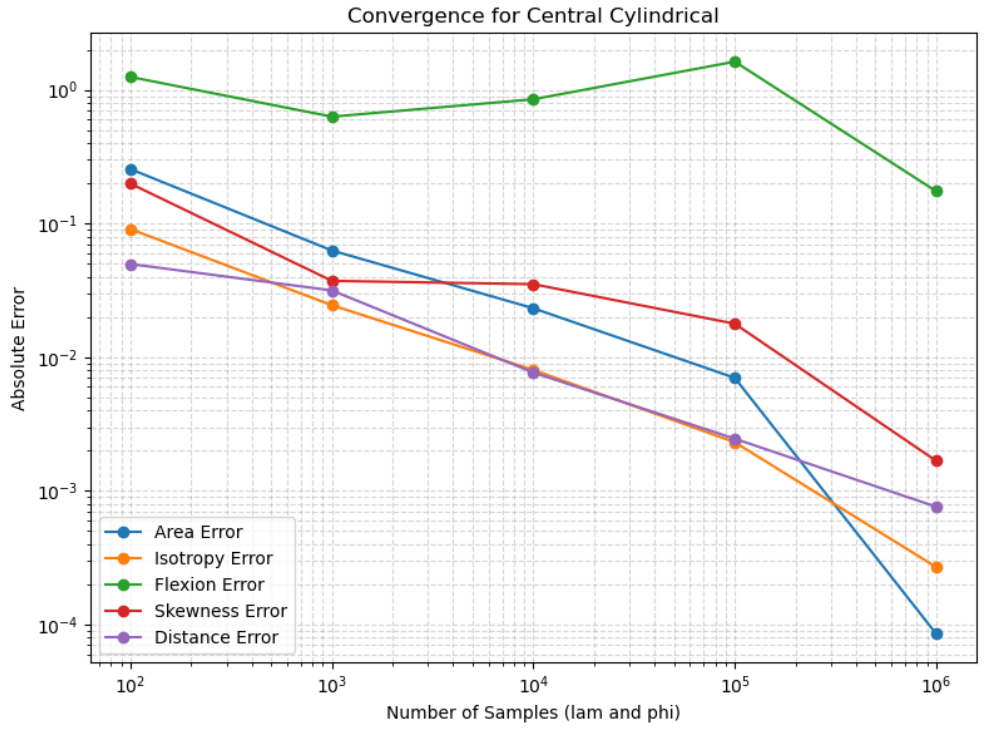


Figure C.5: Central Cylindrical

C.6. Collignon

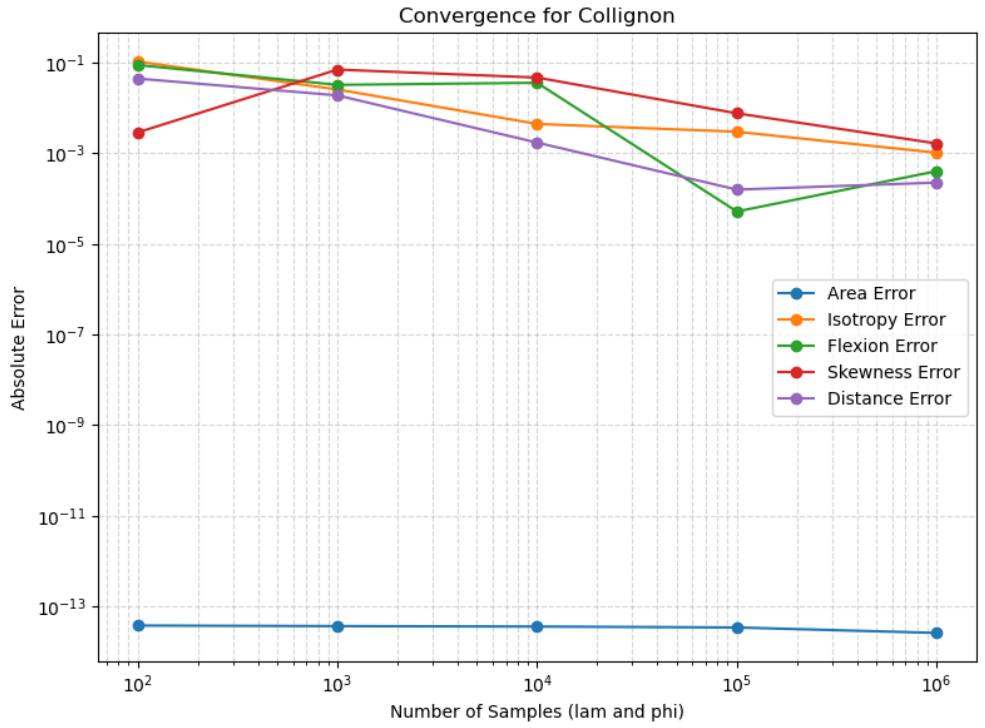


Figure C.6: Collignon

C.7. Equidistant Conic

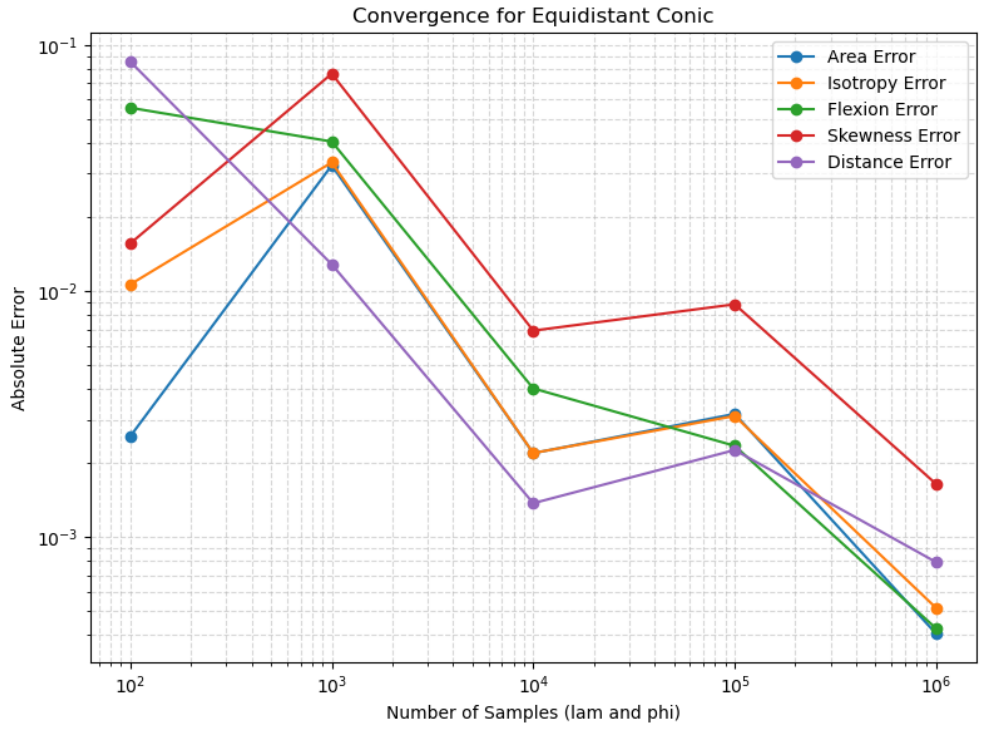


Figure C.7: Equidistant Conic

C.8. Gall-Peters

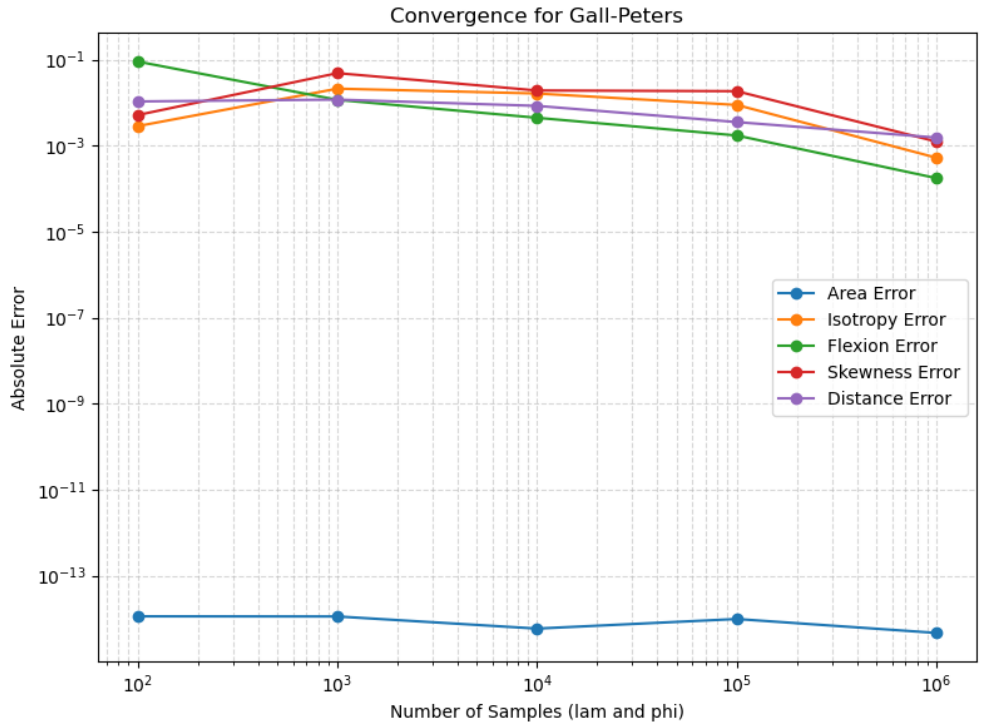


Figure C.8: Gall-Peters

C.9. Hammer

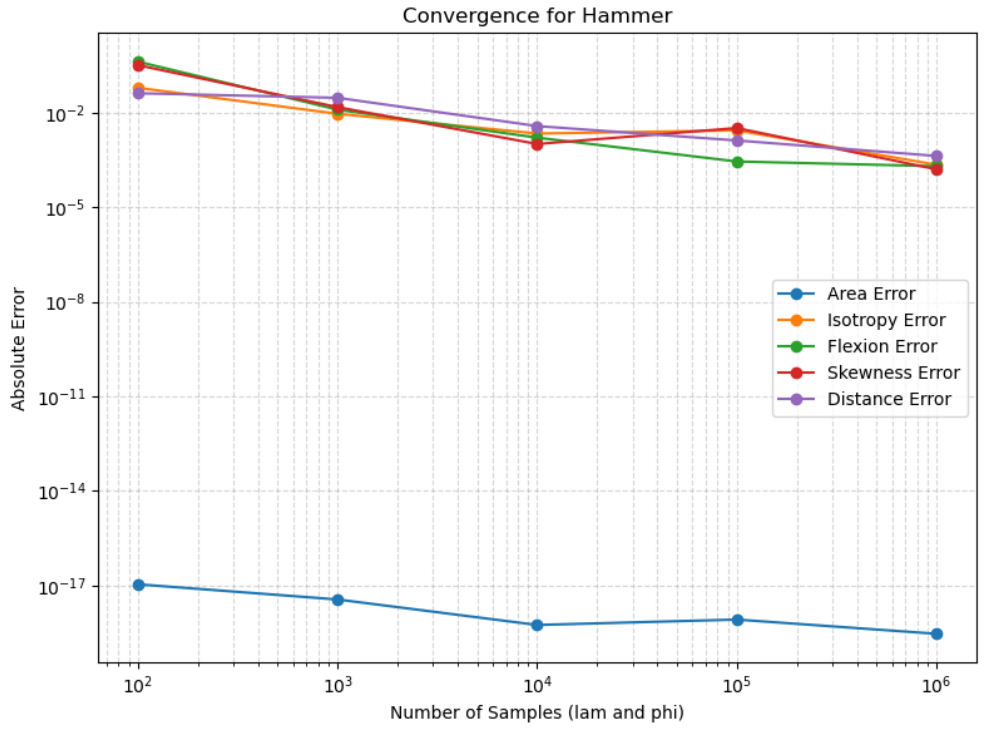


Figure C.9: Hammer

C.10. Lambert Azimuthal Equal-Area (North Polar)

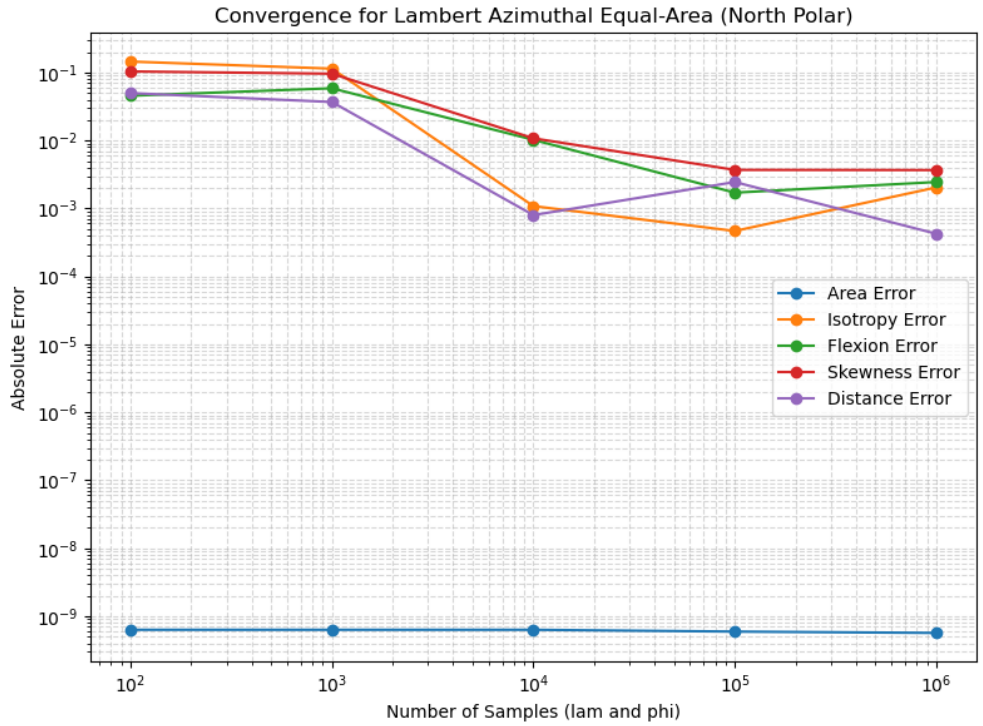


Figure C.10: Lambert Azimuthal Equal-Area (North Polar)

C.11. Lambert Conic

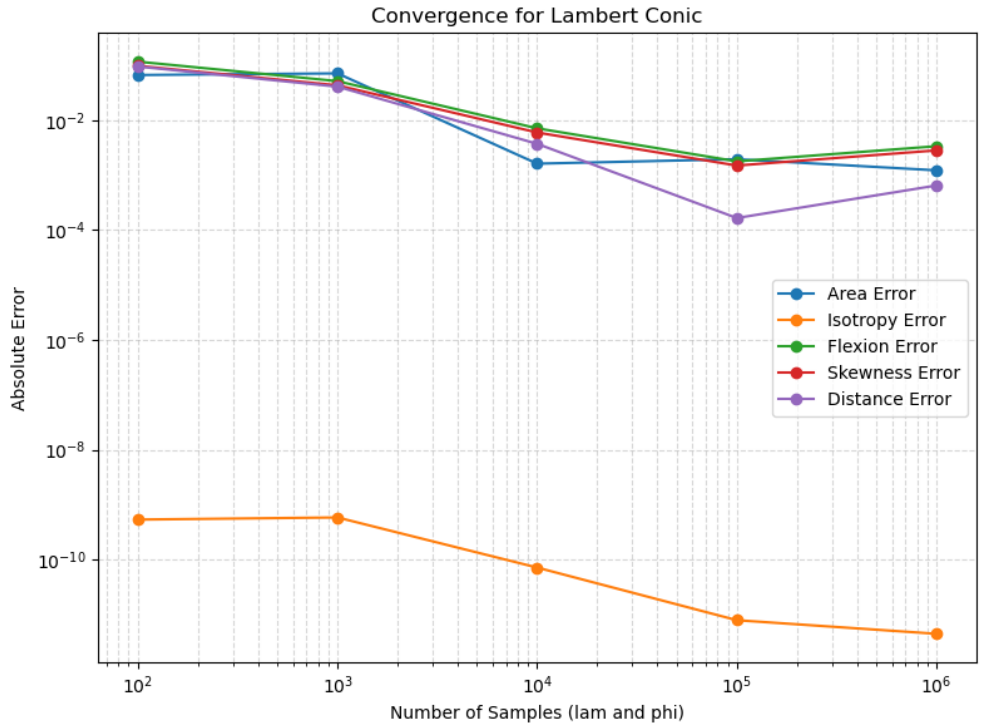


Figure C.11: Lambert Conic

C.12. Lambert Cylindrical Equal-Area

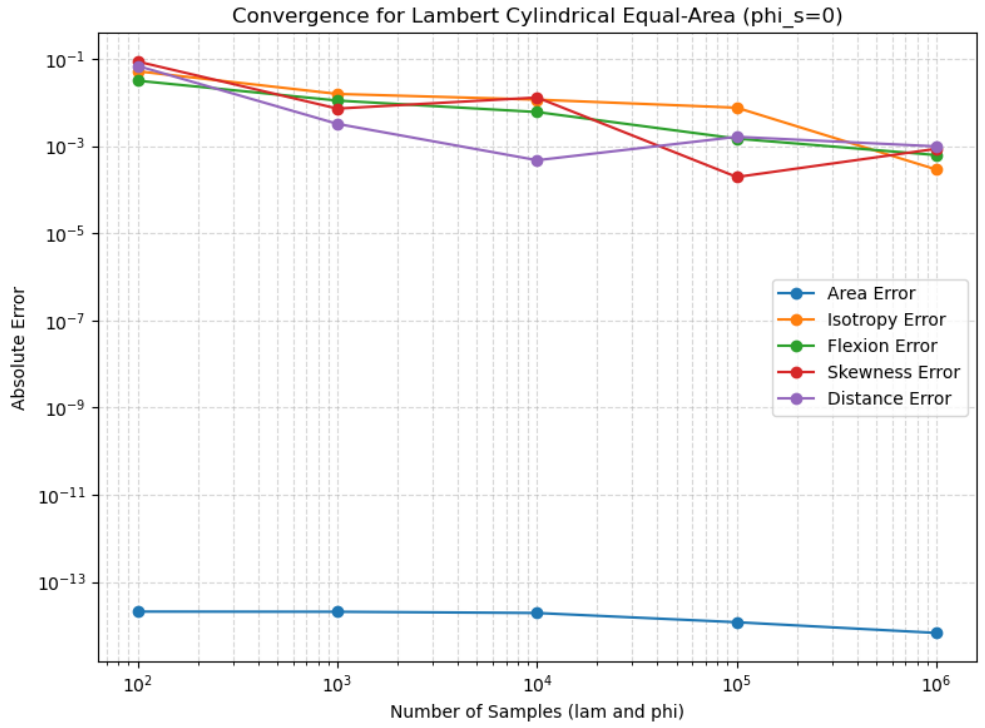


Figure C.12: Lambert Cylindrical Equal-Area

C.13. Mercator

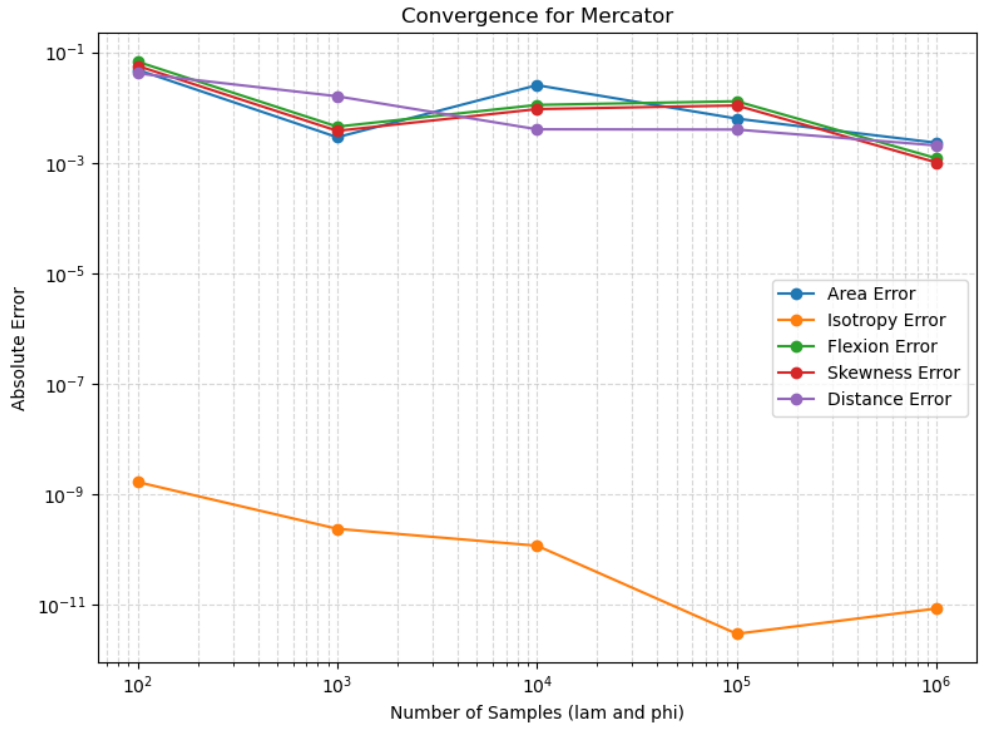


Figure C.13: Mercator

C.14. Miller Cylindrical

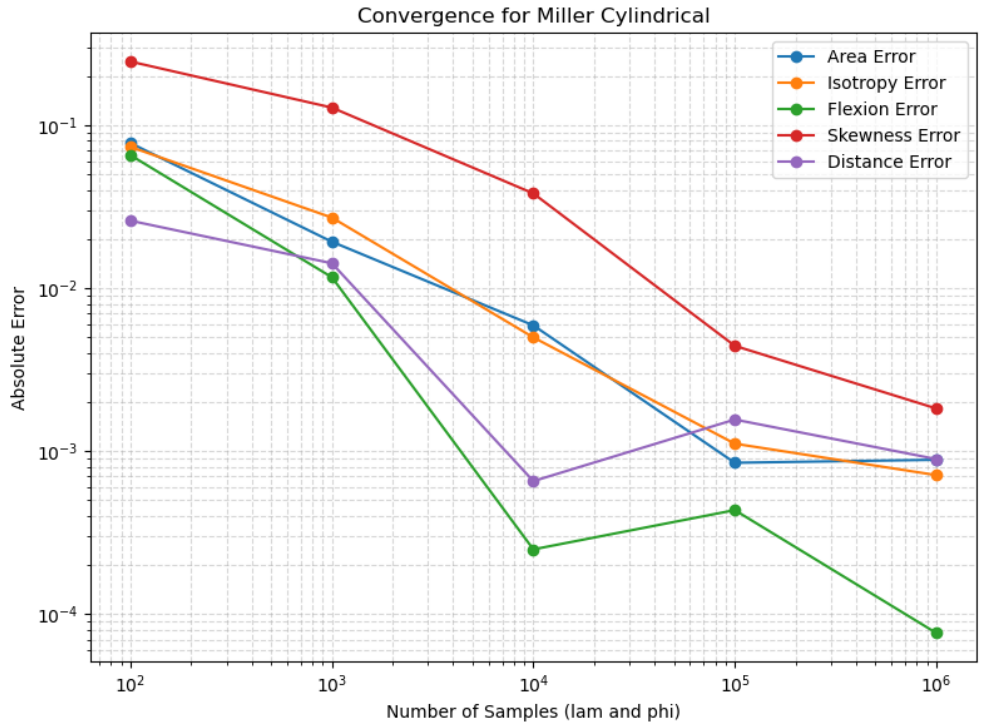


Figure C.14: Miller Cylindrical

C.15. Sinusoidal

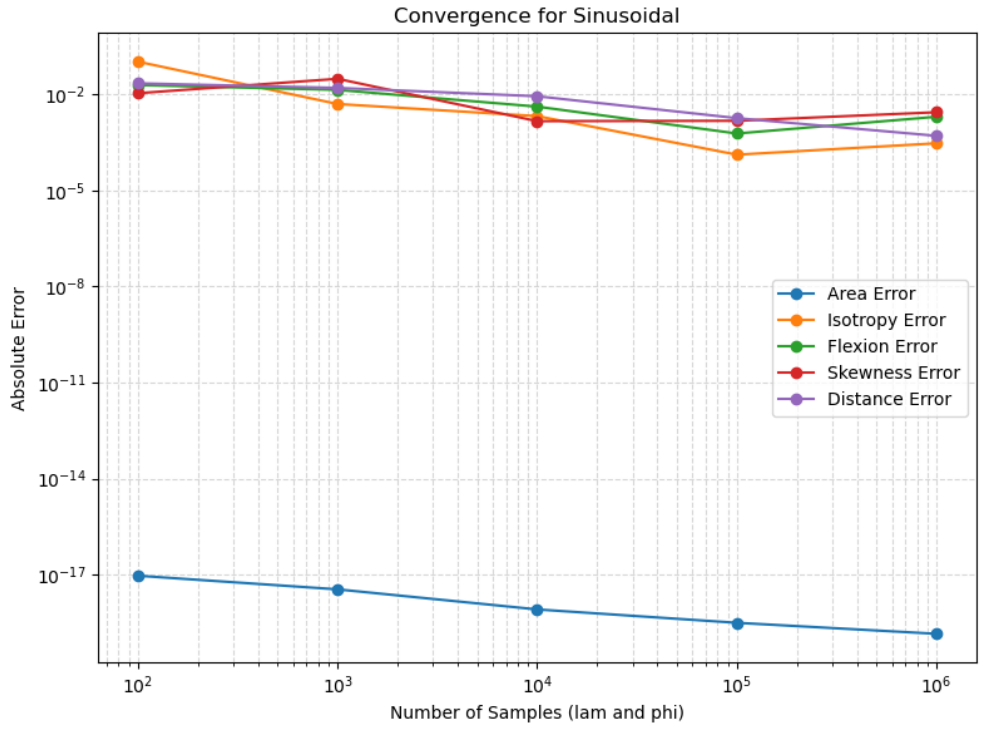


Figure C.15: Sinusoidal

C.16. Spilhaus Stereographic

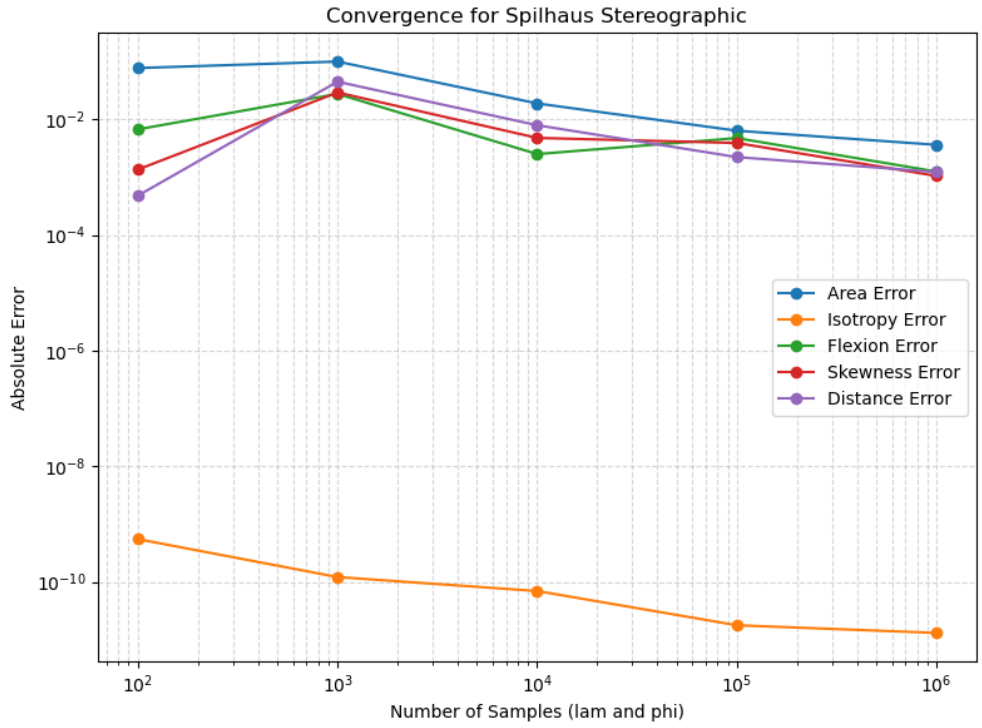


Figure C.16: Spilhaus Stereographic

C.17. Stereographic (North Polar)

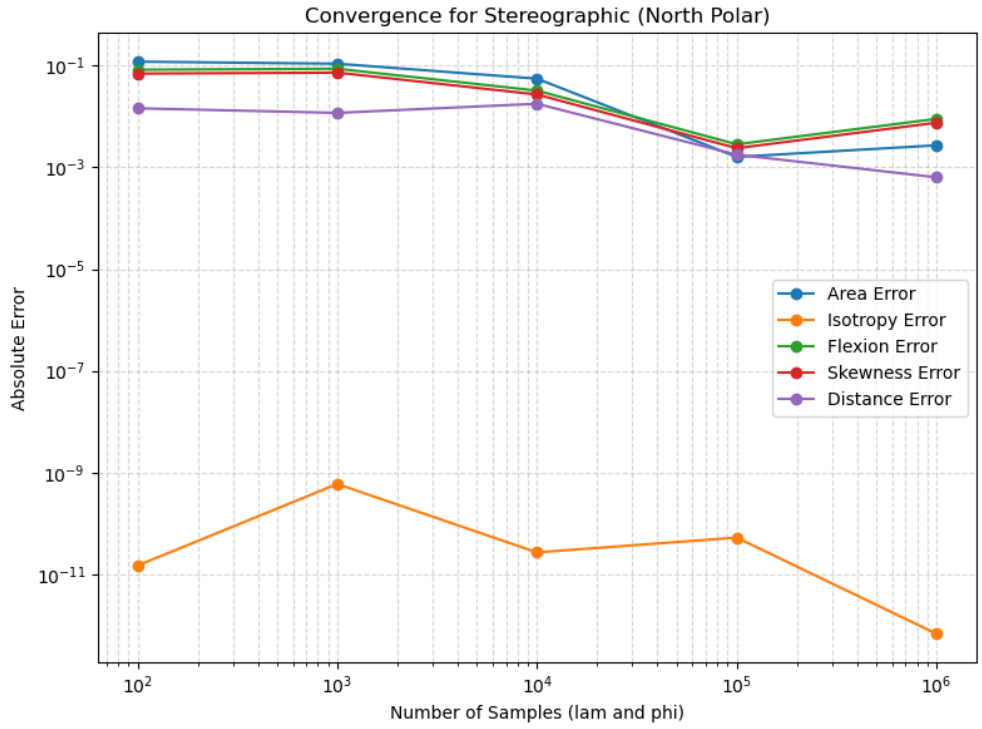


Figure C.17: Stereographic (North Polar)

C.18. Wiechel

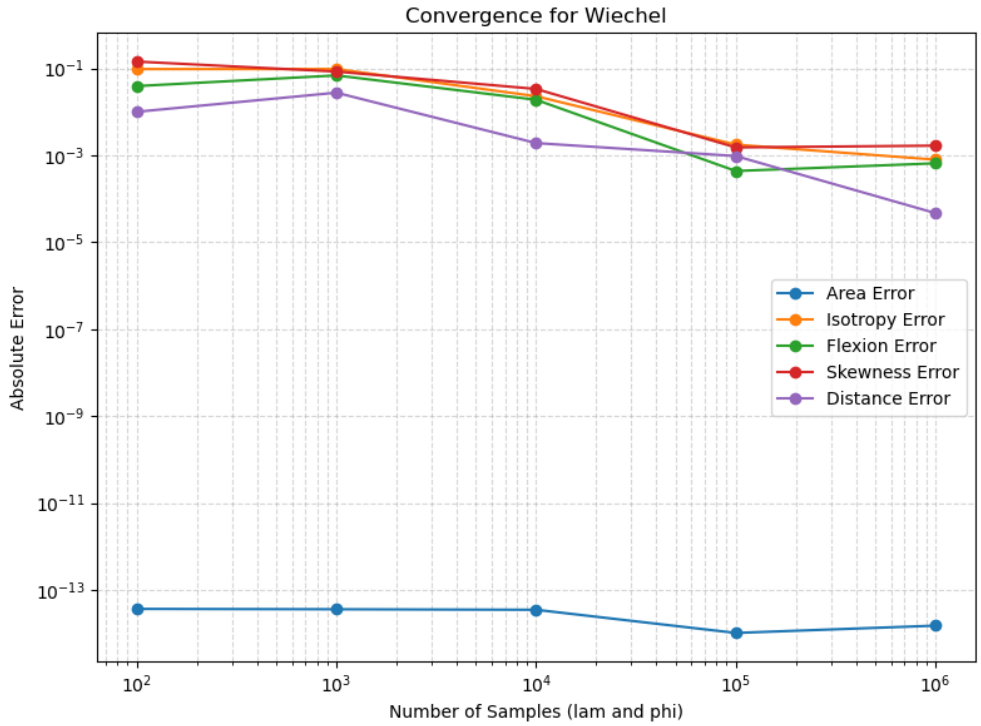


Figure C.18: Wiechel

C.19. Winkel II

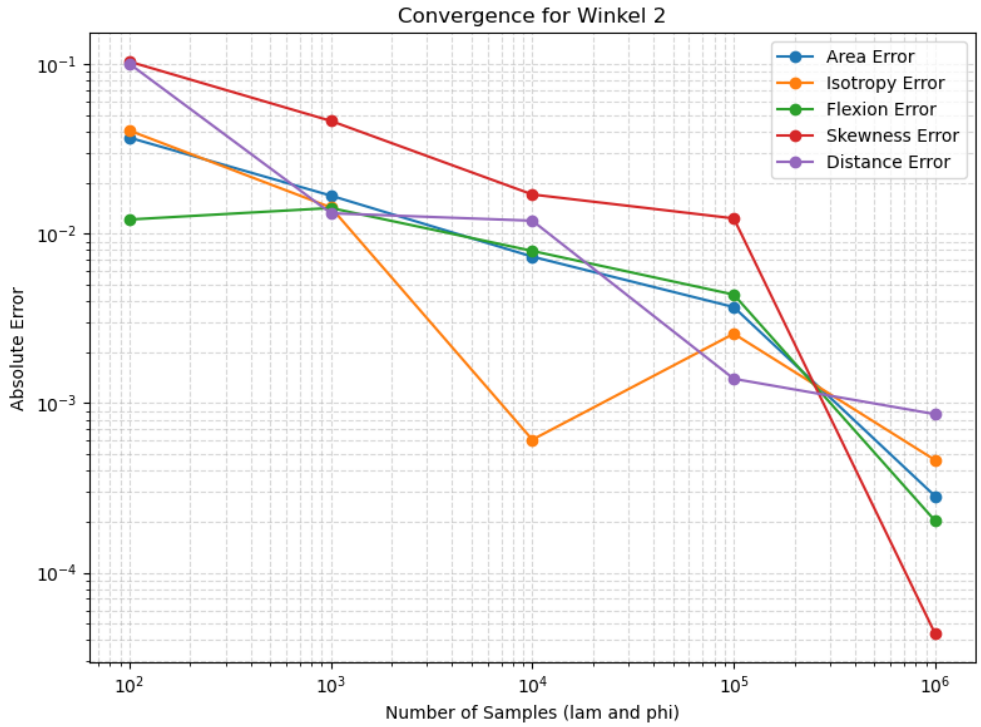


Figure C.19: Winkel II

C.20. Winkel Tripel

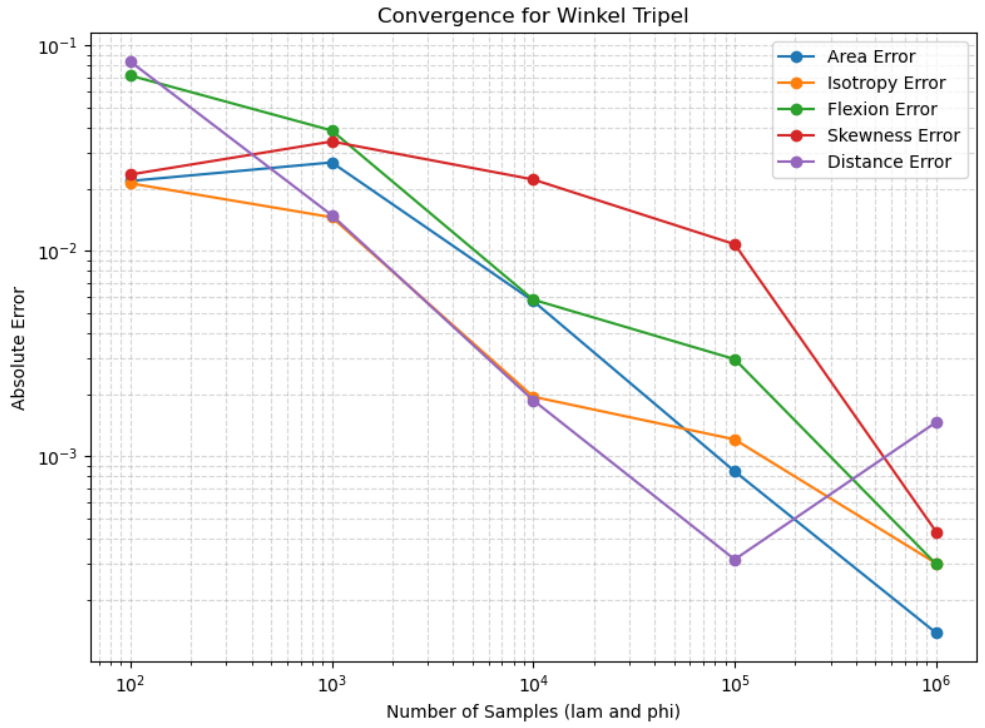


Figure C.20: Winkel Tripel

D

Fractal Dimension Plots

D.1. Aitoff

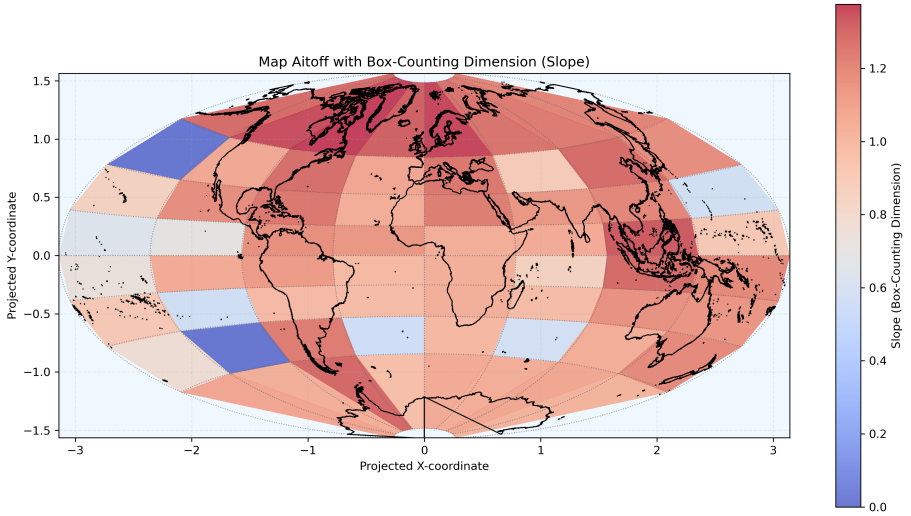


Figure D.1: Aitoff

D.2. Azimuthal Equidistant (North Polar)

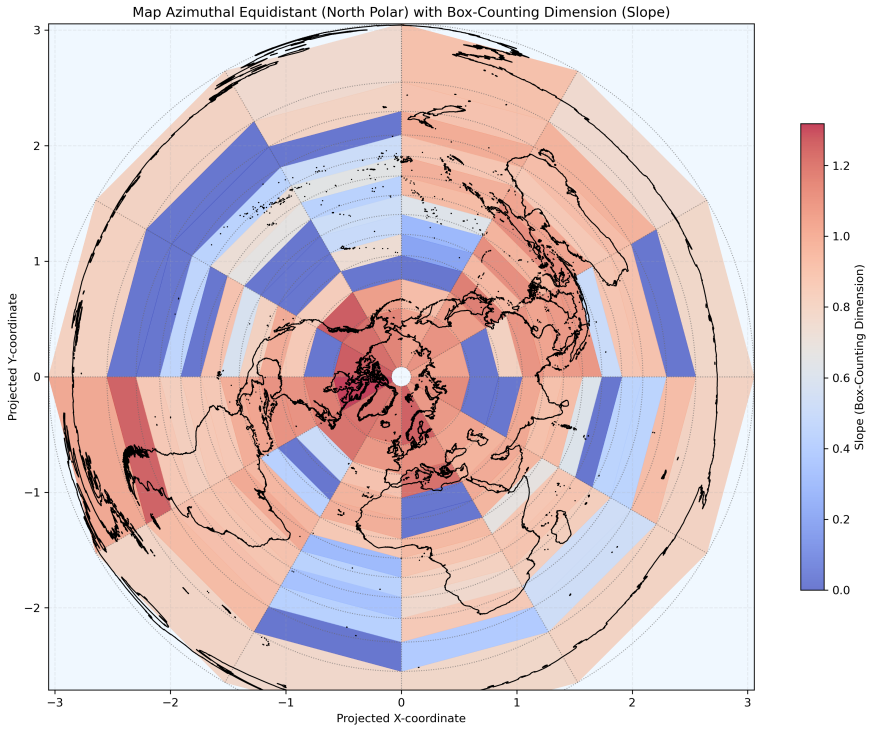


Figure D.2: Azimuthal Equidistant (North Polar)

D.3. Behrmann

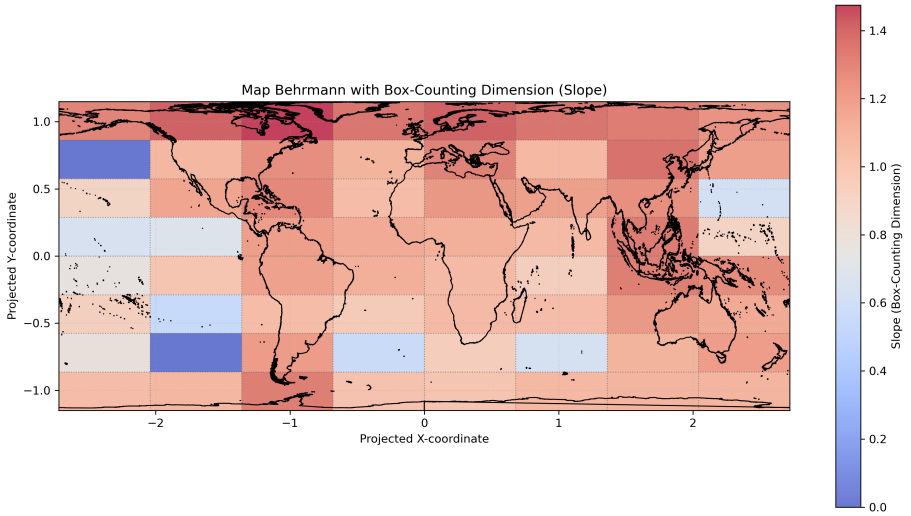


Figure D.3: Behrmann

D.4. Bonne

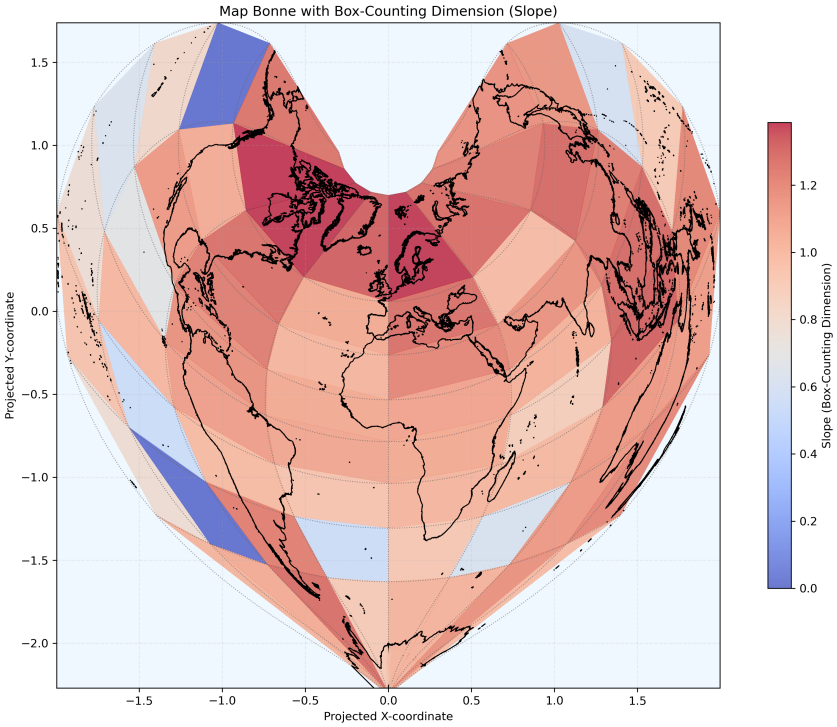


Figure D.4: Bonne

D.5. Cassini

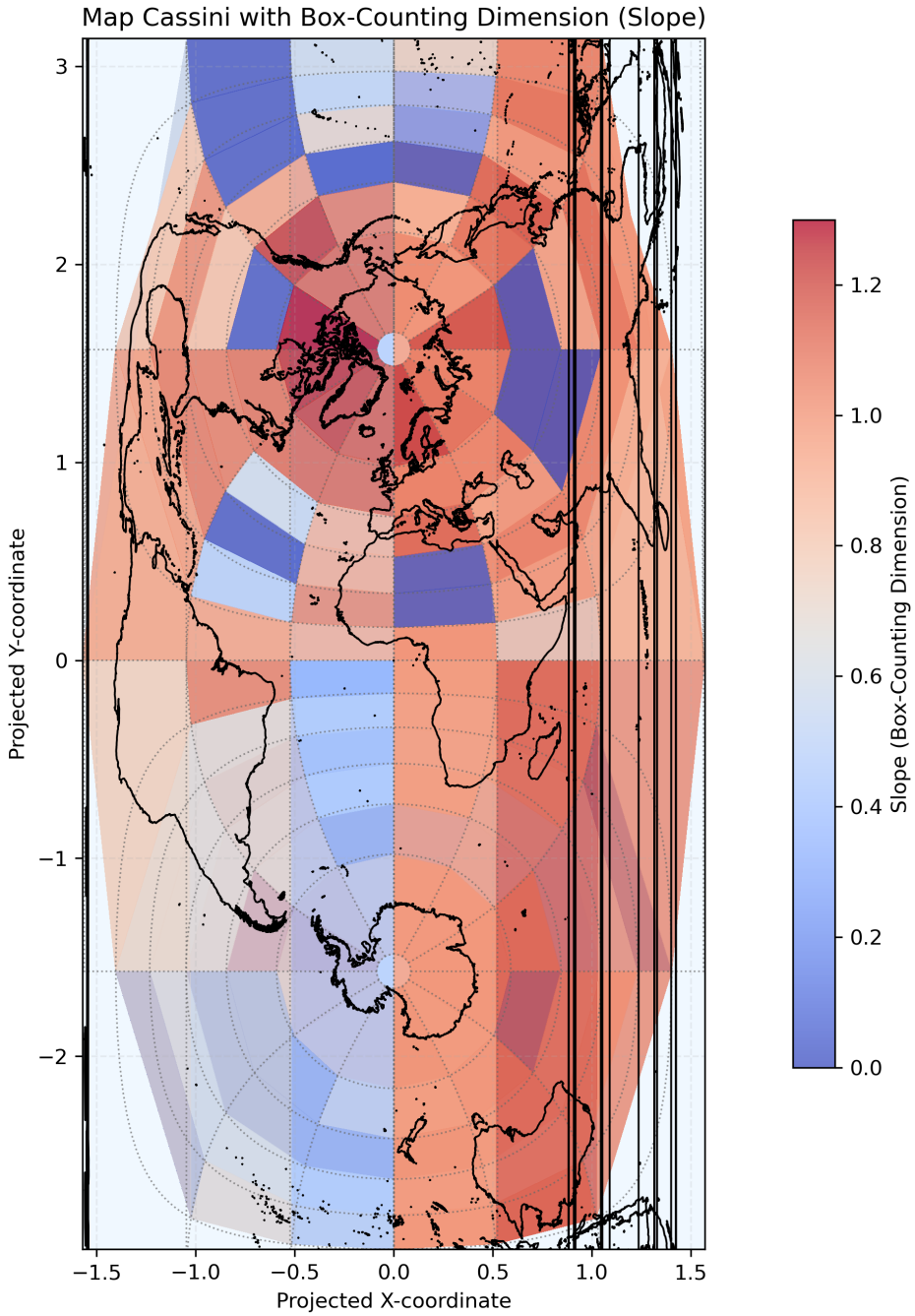


Figure D.5: Cassini

D.6. Central Cylindrical

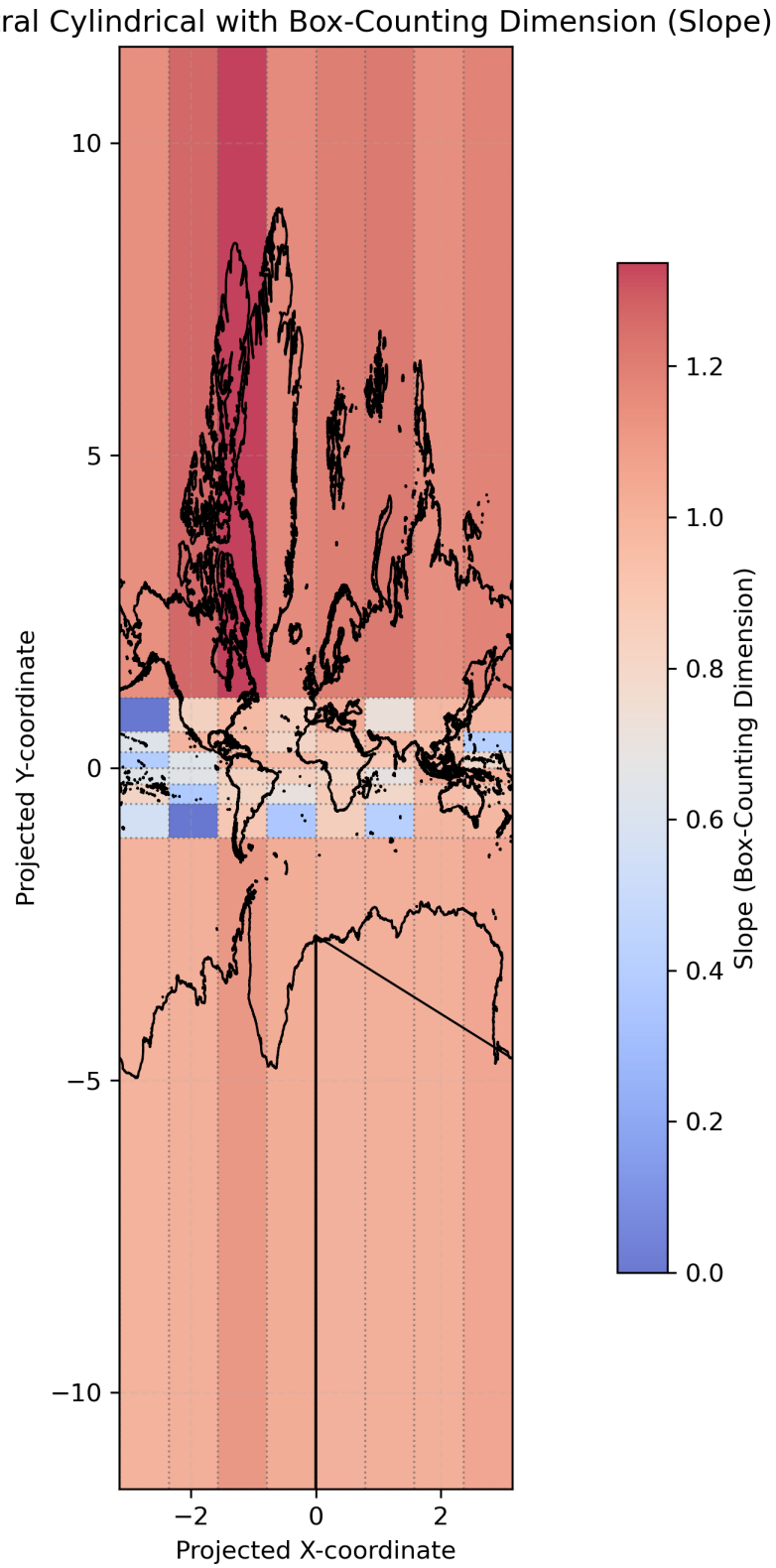


Figure D.6: Central Cylindrical

D.7. Collignon

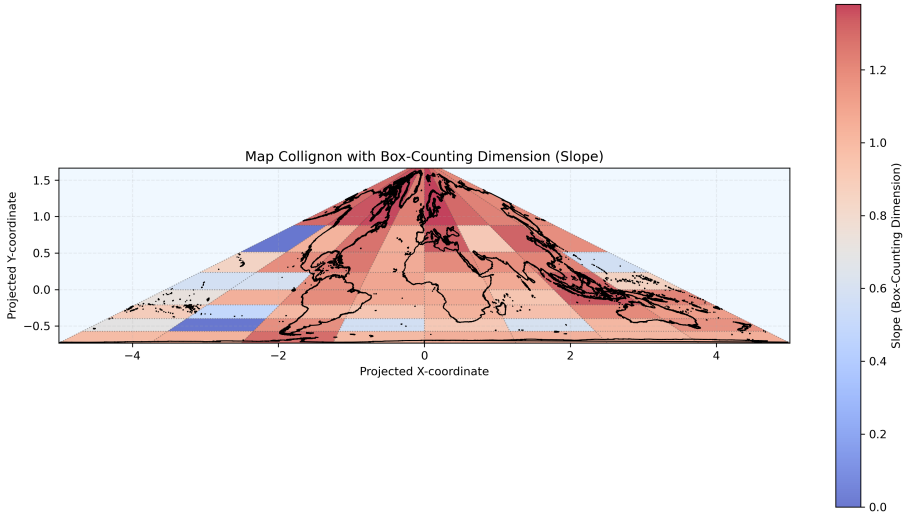


Figure D.7: Collignon

D.8. Equidistant Conic

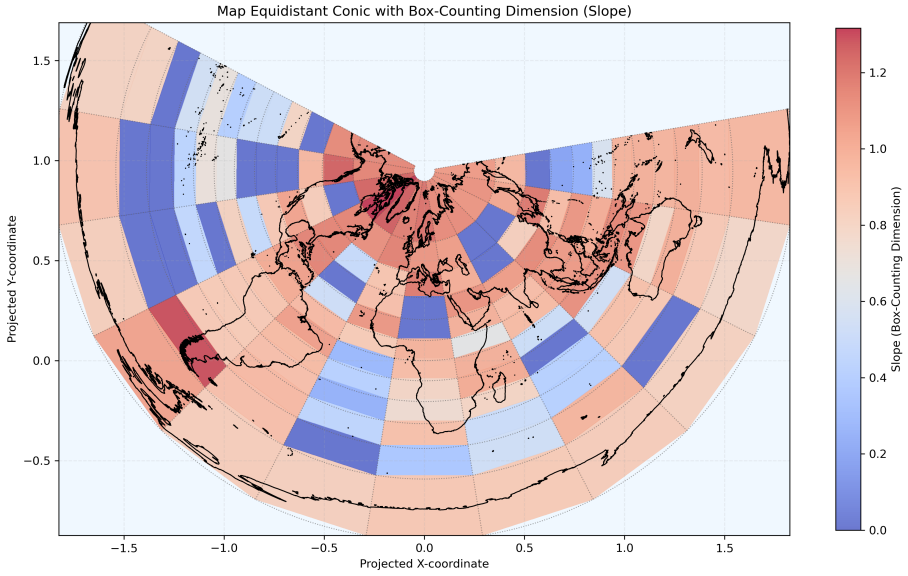


Figure D.8: Equidistant Conic

D.9. Equirectangular

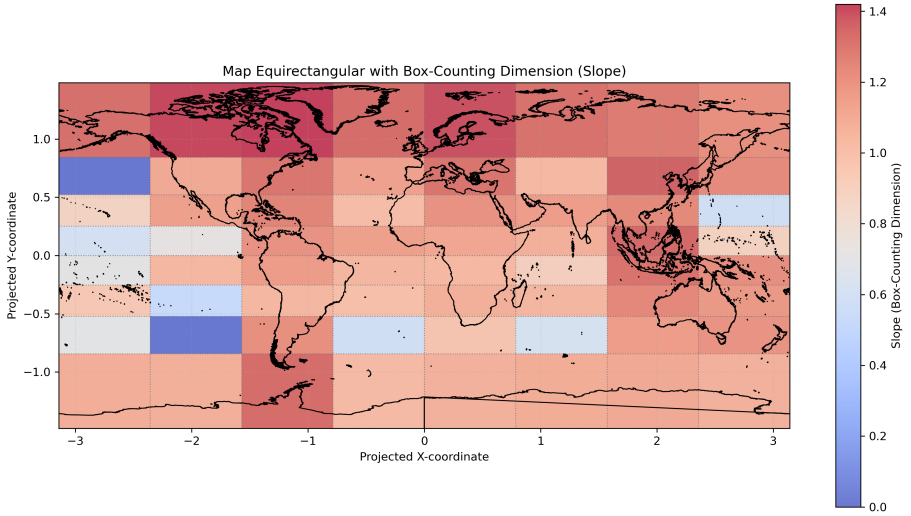


Figure D.9: Equirectangular

D.10. Gall-Peters

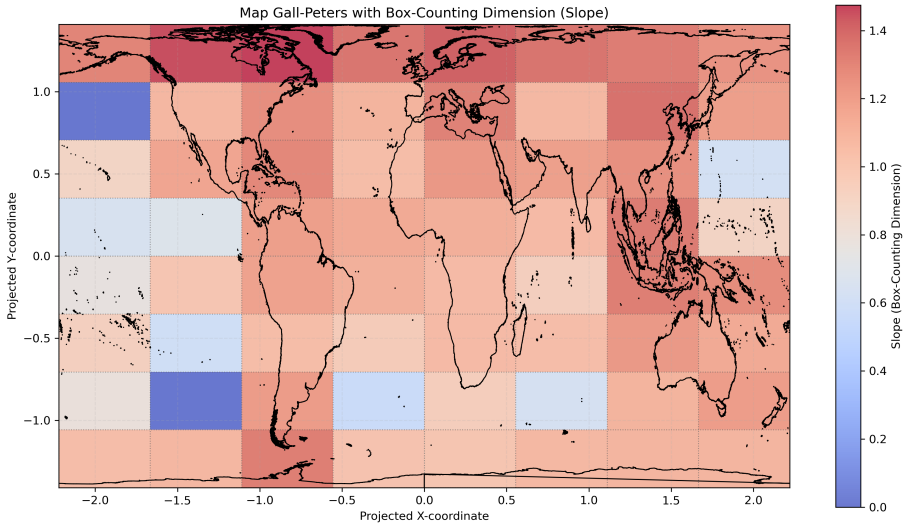


Figure D.10: Gall-Peters

D.11. Hammer

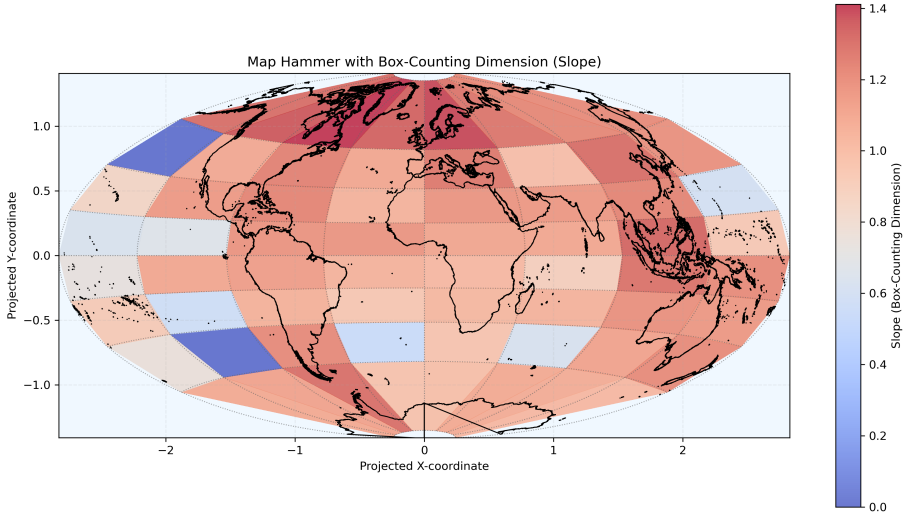


Figure D.11: Hammer

D.12. Lambert Azimuthal Equal-Area (North Polar)

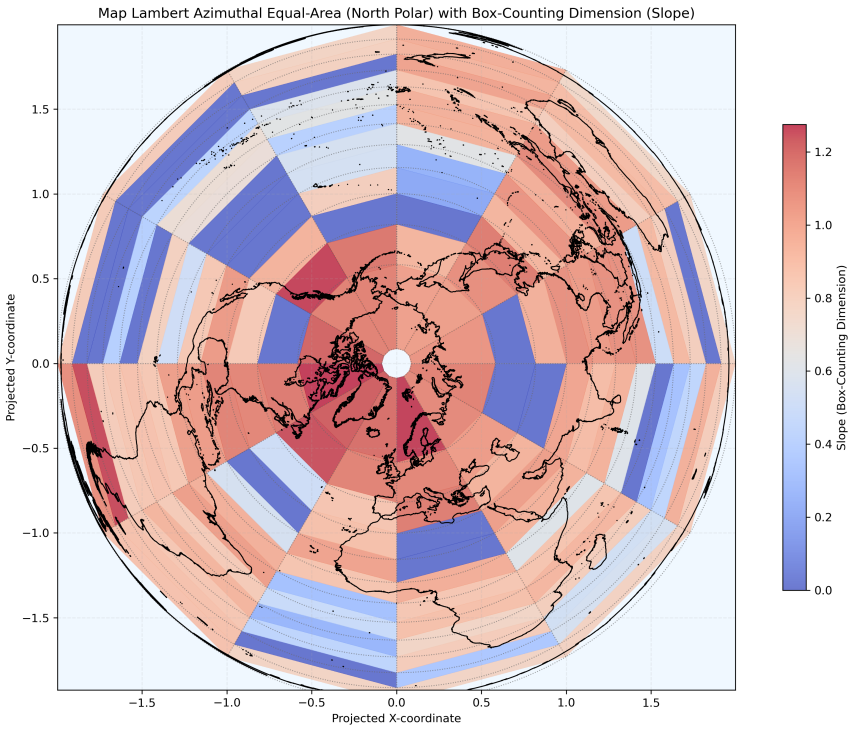


Figure D.12: Lambert Azimuthal Equal-Area (North Polar)

D.13. Lambert Cylindrical Equal-Area

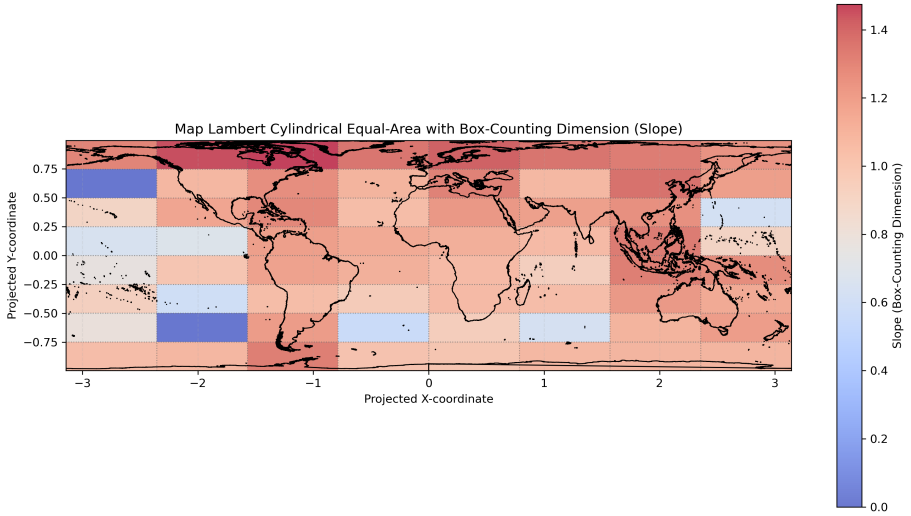


Figure D.13: Lambert Cylindrical Equal-Area

D.14. Miller Cylindrical

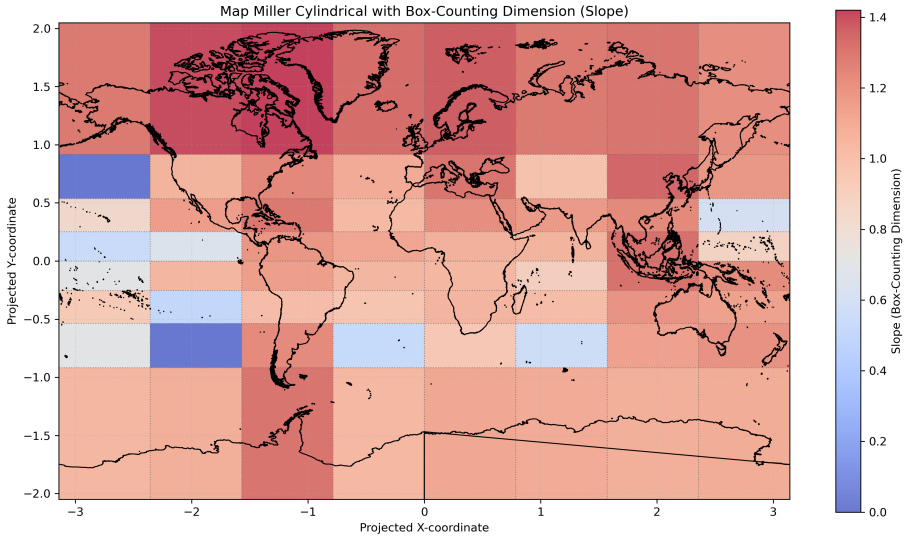


Figure D.14: Miller Cylindrical

D.15. Sinusoidal

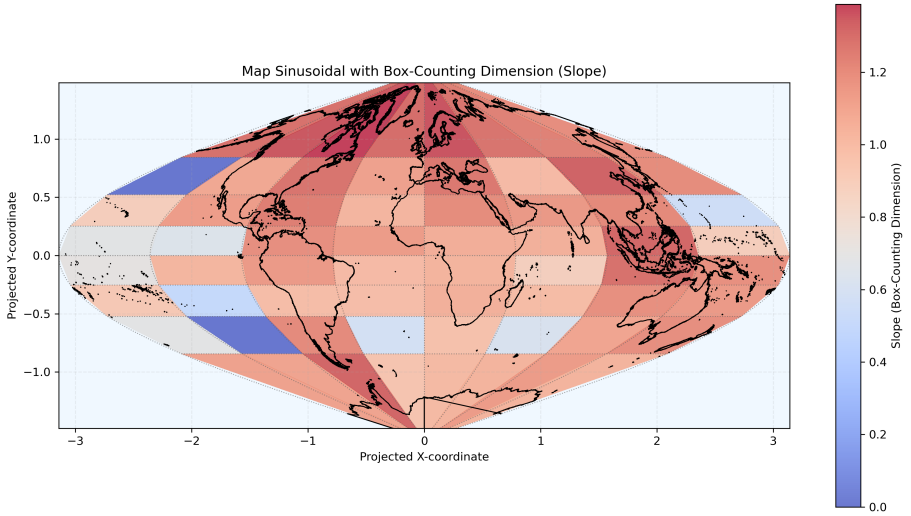


Figure D.15: Sinusoidal

D.16. Wiechel

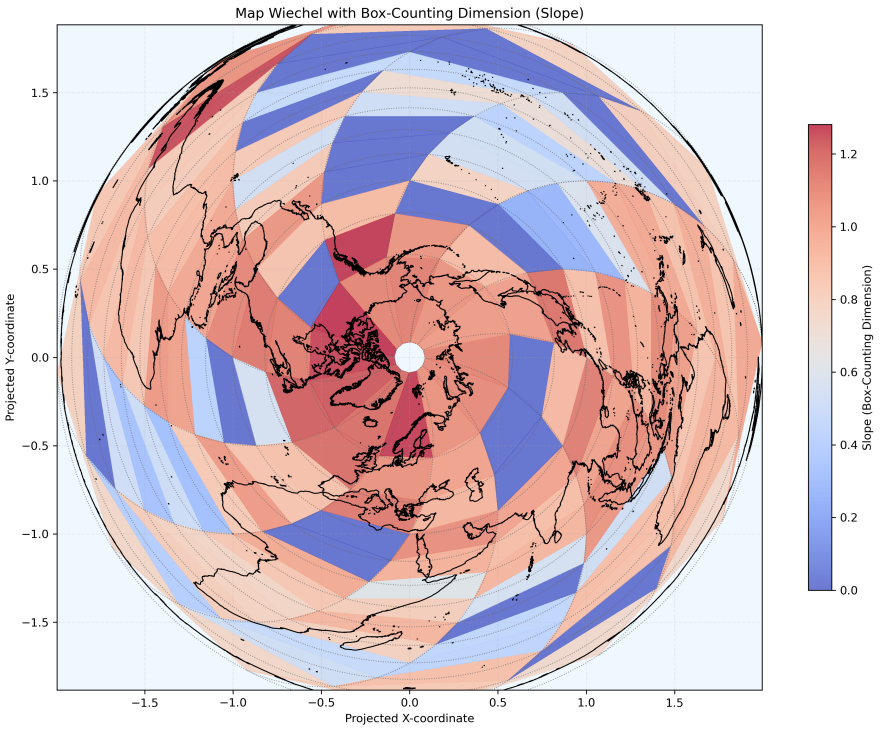


Figure D.16: Wiechel

D.17. Winkel II

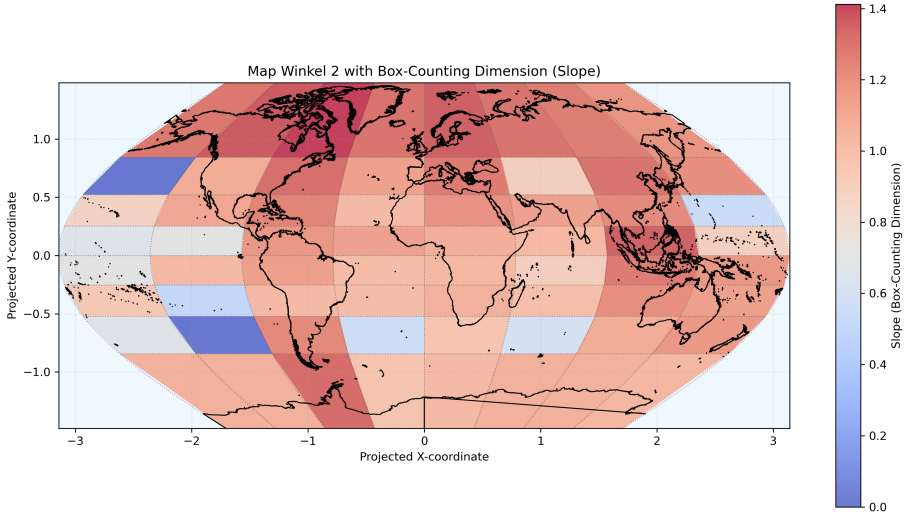


Figure D.17: Winkel II

D.18. Winkel Tripel

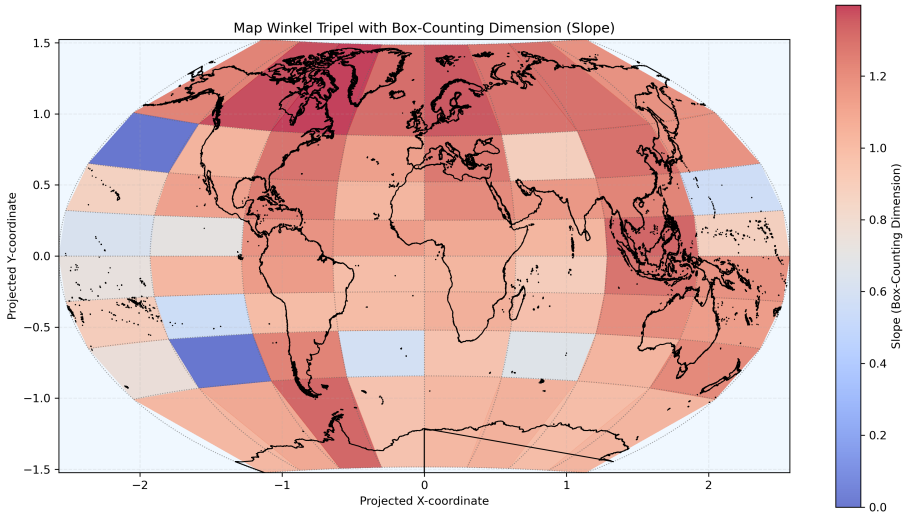


Figure D.18: Winkel Tripel

E

Code

distortions.py

```
# distortions.py
# NOT C ENCODED DISTORTIONS (SLOWER, BUT MORE INTERPRETABLE CALCULATION)
import sympy as sp
import numpy as np
import pandas as pd
from tabulate import tabulate
from map_projections import projection_formulas
import time
import matplotlib.pyplot as plt
import geopandas as gpd

# define variables
lam, phi, alpha = sp.symbols('lam phi alpha')

def plot_projected_coastlines(projection, x_func, y_func, lam_vals, phi_vals, shapefile_path):
    x_expr = sp.lambdify((lam, phi), x_func, 'numpy')
    y_expr = sp.lambdify((lam, phi), y_func, 'numpy')
    gdf = gpd.read_file(shapefile_path)
    #gdf = shapefile_path

    plt.figure()
    fig, ax = plt.subplots()

    for geom in gdf.geometry:
        if geom.geom_type == 'LineString':
            lon, lat = geom.xy
            lon = np.array(lon)
            lat = np.array(lat)

            lam_rad = np.clip(np.radians(lon), -np.pi, np.pi)
            phi_rad = np.clip(np.radians(lat), -np.pi/2+0.001, np.pi/2-0.001)


            x = x_expr(lam_rad, phi_rad)
            y = y_expr(lam_rad, phi_rad)

            plt.plot(x, y, color='black', linewidth=0.3)

    for lam1, phi1 in zip(lam_vals, phi_vals):
        lam_add = np.random.uniform(0*np.pi, np.pi)
        phi_add = np.random.uniform(0 * np.pi, 0.5 * np.pi)
```

```
        # lam_add and phi_add for plot
        #lam_add = 0 #np.pi * np.array([0,0,0,0,0,0,0])
        #phi_add = 1/2*np.pi-0.01 #np.pi * np.array([1/2, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2])

        # New point
        lam2 = (lam1 + lam_add)
        phi2 = (phi1 + phi_add)


        # Interpolate great circle
        lam_gc, phi_gc = mapping.interpolate_gc(lam1, phi1, lam2, phi2, n=100)

        # Convert to x, y
        x_gc = mapping.x_func(lam_gc, phi_gc)
        y_gc = mapping.y_func(lam_gc, phi_gc)

        # Fix antimeridian crossing by inserting NaNs where needed
        x_fixed = [x_gc[0]]
        y_fixed = [y_gc[0]]

        for i in range(1, len(x_gc)):
            # Check for discontinuity in longitude (wrap around)
            if abs(lam_gc[i] - lam_gc[i - 1]) > np.pi:
                x_fixed.append(np.nan)
                y_fixed.append(np.nan)
            x_fixed.append(x_gc[i])
            y_fixed.append(y_gc[i])

        # Plot
        ax.plot(x_fixed, y_fixed, color='blue', linewidth=.9)


        x1 = x_expr(lam1, phi1)
        y1 = y_expr(lam1, phi1)
        x2 = x_expr(lam2, phi2)
        y2 = y_expr(lam2, phi2)

        ax.plot([x1,x2],[y1,y2], color = 'red', linewidth = 0.9)

    ax.set_axis_off()
    plt.axis('equal')
    plt.subplots_adjust(left=0, right=1, top=1, bottom=0)
    plt.savefig(f"Distance_Plots/{projection}.png", dpi=300, bbox_inches='tight',
     ↪  pad_inches=0)
    plt.close()




def rms(array):
    return np.linalg.norm(array) / np.sqrt(array.size)

class distortions:
    def __init__(self, x, y):
        # Define symbolic variables
        self.lam = lam
        self.phi = phi
        self.alpha = alpha
```

```
# Define transformation formulas
self.x = x
self.y = y

self.x_func = sp.lambdify((lam, phi), self.x, 'numpy')
self.y_func = sp.lambdify((lam, phi), self.y, 'numpy')

# Define K matrix
self.K = sp.Matrix([[sp.cos(self.phi)**-1,   0],
                    [0,                      1]])

# Define M matrix
self.M = sp.Matrix([self.x, self.y])

# Define the vector of variables
self.vars = sp.Matrix([self.lam, self.phi])

# Calculate Jacobion
self.J = self.M.jacobian(self.vars)


# Calculate transformation matrix T and convert to function for faster calculations
#self.T = sp.lambdify((self.lam, self.phi), self.J * self.K, 'numpy')

# Compute symbolic expression first
T_sym = (self.J * self.K).tolist()  # Convert to list of expressions

# Lambdify each element to create a fully vectorized 2x2 function
self.T_funcs = [[sp.lambdify((lam, phi), T_sym[i][j], 'numpy')
for j in range(2)] for i in range(2)]


# Hessian for x and y
self.Hx = sp.hessian(x, (self.lam, self.phi))

self.Hy = sp.hessian(y, (self.lam, self.phi))


# Define w
self.w = self.K * sp.Matrix([sp.tan(self.phi)*sp.sin(2*self.alpha),
sp.tan(self.phi)*(-(sp.cos(self.alpha)**2))])


# Compute u and v vectors
self.u = self.K * sp.Matrix([sp.cos(self.alpha), sp.sin(self.alpha)])


# Compute v and v_ort vectors
self.v = self.J * self.u

self.v_ort    = sp.Matrix([0, 0])
self.v_ort[0] = -self.v[1]
self.v_ort[1] =  self.v[0]


# Compute a vector
#self.a = self.J * self.w + sp.Matrix([self.u.T*self.Hx*self.u,
↪  self.u.T*self.Hy*self.u])

ux = (self.u.T * self.Hx * self.u)[0]
uy = (self.u.T * self.Hy * self.u)[0]
```

```
        self.a = self.J * self.w + sp.Matrix([ux, uy])



    def singularities(self, lam_vals, phi_vals):
        # 1) Evaluate the four T-matrix components: each comes back shape (N,)
        T_00 = self.T_funcs[0][0](lam_vals, phi_vals)
        T_01 = self.T_funcs[0][1](lam_vals, phi_vals)
        T_10 = self.T_funcs[1][0](lam_vals, phi_vals)
        T_11 = self.T_funcs[1][1](lam_vals, phi_vals)

        # 2) Force them all to shape (N,) (in case some returned scalars)
        T_00, T_01, T_10, T_11 = np.broadcast_arrays(T_00, T_01, T_10, T_11)
        N = T_00.shape[0]

        # 3) Build (N,2,2) T_all
        T_all = np.empty((N, 2, 2))
        T_all[:,0,0] = T_00
        T_all[:,0,1] = T_01
        T_all[:,1,0] = T_10
        T_all[:,1,1] = T_11

        # 4) Batch-compute T^T T, trace, det
        TtT     = np.einsum('nij,nkj->nik', T_all, T_all)
        trace_T = np.trace(TtT, axis1=1, axis2=2)
        det_T   = np.linalg.det(T_all)

        abs_det = np.abs(det_T)
        left    = np.sqrt(trace_T + 2 * abs_det)
        right   = np.sqrt(np.maximum(trace_T - 2 * abs_det, 0))

        a = (left + right) / 2
        b = (left - right) / 2
        return a, b



    def area_and_isotropy(self, lam_vals, phi_vals):
        a, b = self.singularities(lam_vals, phi_vals)
        A = np.std(np.log(a*b))
        I = rms(np.log(a/b))
        return A, I


    def flexion_and_skewness(self, lam_vals, phi_vals, alpha_vals):
        # 1) create your lambdified scalar functions once
        v_x     = sp.lambdify((lam, phi, alpha), self.v[0], 'numpy')
        v_y     = sp.lambdify((lam, phi, alpha), self.v[1], 'numpy')
        v_ort_x = sp.lambdify((lam, phi, alpha), self.v_ort[0], 'numpy')
        v_ort_y = sp.lambdify((lam, phi, alpha), self.v_ort[1], 'numpy')
        a_x     = sp.lambdify((lam, phi, alpha), self.a[0], 'numpy')
        a_y     = sp.lambdify((lam, phi, alpha), self.a[1], 'numpy')

        # 2) build our grid:
        # lam_vals, phi_vals are shape (N,)
        # alpha_vals is shape (M,)
        lam_grid = lam_vals[:, None]       # shape (N,1)
        phi_grid = phi_vals[:, None]       # shape (N,1)
        alpha_grid = alpha_vals[None, :]   # shape (1,M)
```

```
    # 3) evaluate all at once:
    Vx      = v_x(lam_grid, phi_grid, alpha_grid)     # (N,M)
    Vy      = v_y(lam_grid, phi_grid, alpha_grid)
    Vortx   = v_ort_x(lam_grid, phi_grid, alpha_grid)
    Vorty   = v_ort_y(lam_grid, phi_grid, alpha_grid)
    Ax      = a_x(lam_grid, phi_grid, alpha_grid)
    Ay      = a_y(lam_grid, phi_grid, alpha_grid)

    # Broadcast all six to the same shape (N,M):
    Vx, Vy, Vortx, Vorty, Ax, Ay = np.broadcast_arrays(
        Vx, Vy, Vortx, Vorty, Ax, Ay
    )


    # Then stack:
    Vf     = np.stack((Vx,    Vy),    axis=-1)
    Vortf  = np.stack((Vortx, Vorty), axis=-1)
    Af     = np.stack((Ax,    Ay),    axis=-1)


    # 4) stack into shape (N,M,2)
    Vf     = np.stack((Vx, Vy),      axis=-1)
    Vortf  = np.stack((Vortx, Vorty), axis=-1)
    Af     = np.stack((Ax, Ay),      axis=-1)

    # 5) dot products along the last axis:
    v_dot_a     = np.sum(Vf     * Af,      axis=-1)  # (N,M)
    v_ort_dot_a = np.sum(Vortf  * Af,      axis=-1)
    v_dot_v     = np.sum(Vf     * Vf,      axis=-1)

    # 6) safe division and final f, s arrays:
    denom = np.maximum(v_dot_v, 1e-12)
    f_vals = np.abs(v_ort_dot_a) / denom
    s_vals = np.abs(v_dot_a)     / denom

    # 7) single mean
    return f_vals, s_vals


def interpolate_gc(self, lam1, phi1, lam2, phi2, n=100):
    # Returns n points interpolated along great-circle path
    # All angles in radians
    import numpy as np

    # Convert to 3D unit vectors
    def sph2cart(lam, phi):
        return np.array([np.cos(phi)*np.cos(lam),
                         np.cos(phi)*np.sin(lam),
                         np.sin(phi)])

    a = sph2cart(lam1, phi1)
    b = sph2cart(lam2, phi2)

    # Determine angle between a and b
    omega = np.arccos(np.clip(np.dot(a, b), -1.0, 1.0))
    if np.isclose(omega, 0): # Very close points are handled here
        return np.tile([[lam1], [phi1]], (1, n))

    sin_omega = np.sin(omega)
    t = np.linspace(0, 1, n)
    sin_omega = np.sin(omega)
```

```
        # Create points along the great circle
        points = ((np.sin((1 - t) * omega)[None, :] * a[:, None] +
                   np.sin(t * omega)[None, :] * b[:, None]) / sin_omega)

        # Convert back to lat/lon
        lam_gc = np.arctan2(points[1], points[0])
        phi_gc = np.arcsin(points[2])
        return lam_gc, phi_gc


    def distance(self, lam_vals, phi_vals, interpolate_steps):
        # Shifted arrays
        lam1, lam2 = lam_vals[:-1], lam_vals[1:]
        phi1, phi2 = phi_vals[:-1], phi_vals[1:]

        # 3D unit vectors on the globe (vectorized)
        x1 = np.array([np.cos(phi1) * np.cos(lam1),
                       np.cos(phi1) * np.sin(lam1),
                       np.sin(phi1)])
        x2 = np.array([np.cos(phi2) * np.cos(lam2),
                       np.cos(phi2) * np.sin(lam2),
                       np.sin(phi2)])

        # Dot product and globe distance
        dot = np.sum(x1 * x2, axis=0)
        dot = np.clip(dot, -1.0, 1.0)  # Prevent numerical domain error
        dist_globe = np.arccos(dot)

        # Improved method
        dist_map_list = []
        for lam1, lam2, phi1, phi2 in zip(lam_vals[:-1],
        lam_vals[1:], phi_vals[:-1], phi_vals[1:]):
            lam_gc, phi_gc = self.interpolate_gc(lam1, phi1, lam2, phi2, n=interpolate_steps)
            # n=10000 for 10e-3 error
            x_gc = self.x_func(lam_gc, phi_gc)
            y_gc = self.y_func(lam_gc, phi_gc)

            dx = np.diff(x_gc)
            dy = np.diff(y_gc)
            dist_map = np.sum(np.sqrt(dx**2 + dy**2))
            dist_map_list.append(dist_map)

        # Ratio and filtering
        dist_map_array = np.array(dist_map_list)
        dist = dist_map_array / dist_globe
        valid = np.isfinite(dist) & (dist > 0)
        log_dist = np.log(dist[valid])
        return np.std(log_dist) # std to make invariant under scaling


if __name__ == "__main__":
    start_time = time.time()
    # Sampling of points on the globe
    lam_vals = np.random.uniform(-np.pi, np.pi, 100_000)
    phi_vals = np.arcsin(np.random.uniform(-0.9999999, 0.9999999, 100_000))
    # To prevent division by zero
    alpha_vals = np.linspace(0, 2*np.pi, 500)

    # CHOOSE PROJECTION
    projection = "Equirectangular"
```

```
x_expr, y_expr = projection_formulas[projection] # Dictionary with projection formulas

mapping = distortions(x_expr, y_expr)

a, i = mapping.area_and_isotropy(lam_vals, phi_vals)
f, s = mapping.flexion_and_skewness(lam_vals, phi_vals, alpha_vals)
d = mapping.distance(lam_vals, phi_vals, 10000)

df = pd.DataFrame({
    "Area": [round(a,4)],
    "Isotropy": [round(i,4)],
    "Flexion": [round(f.mean(),4)],
    "Skewness": [round(s.mean(),4)],
    "Distance": [round(d,5)]
}).T.reset_index()

df.columns = ['Distortion', 'Value']

print(tabulate(df, headers = 'keys', tablefmt = 'presto'))
print("--- %s seconds ---" % (time.time() - start_time))


# Plotting of distances
# SET YOUR OWN PATH
# (this example uses natural earth data 10m, because it has smooth coastline)
coastline_data_10  = "OWN/PATH"

plot_projected_coastlines(
projection,
x_expr, y_expr,
lam_vals, phi_vals,
coastline_data_10)
```

setup.py

```
# setup.py
# THIS FILE TRANSFORMS distortions_cython.pyx TO A C ENCODED FILE
# ONE NEEDS TO DOWNLOAD IBM'S SOFTWARE TO TRANSFROM PYTHON/CYTHON TO C

from setuptools import setup, Extension
from Cython.Build import cythonize
import numpy

# Define the extension module
extensions = [
    Extension(
        "distortions_cython",  # Name of the resulting module
        ["distortions_cython.pyx"],
        include_dirs=[numpy.get_include()],
        # Optional: Add extra compile arguments for optimization if needed
        # extra_compile_args=["-O3", "-march=native"],
    )
]

setup(
    ext_modules=cythonize(extensions, compiler_directives={'language_level': "3"}),
)
```

distortions_cython.pyx

```
# distortions_cython.pyx
# distortions.py FILE IN CYTHON LANGUAGE TO CONVERT TO C
# distorsions_cython.pyx
# cython: language_level=3
# cython: cdivision=True
# cython: boundscheck=True
# cython: wraparound=True



import sympy as sp
import numpy as np
cimport numpy as np # For C-level NumPy API
cimport cython


# It's important to initialize NumPy C API
np.import_array()

# Define symbolic variables that MUST be used when creating expressions for this class
lam_sym, phi_sym, alpha_sym = sp.symbols('lam phi alpha')

def rms_py(np.ndarray[np.double_t, ndim=1] array):
    if array.size == 0:
        return 0.0
    return np.linalg.norm(array) / np.sqrt(array.size)


cdef class DistortionsCython:
    # --- Sympy symbolic variables ---
    cdef object lam
    cdef object phi
    cdef object alpha

    # --- Transformation formulas (Sympy expressions) ---
    cdef object x_expr
    cdef object y_expr

    # --- Lambdified functions (Python callables) ---
    cdef object x_func_callable
    cdef object y_func_callable

    # --- Matrices (Sympy Matrix objects) ---
    cdef object K_matrix
    cdef object M_matrix
    cdef object vars_matrix
    cdef object J_matrix
    cdef list T_funcs_list # List of lists of Python callables

    cdef object Hx_matrix
    cdef object Hy_matrix
    cdef object w_vector
    cdef object u_vector
    cdef object v_vector
    cdef object v_ort_vector
    cdef object a_vector

    # Lambdified scalar functions for flexion/skewness
    cdef object v_x_callable
    cdef object v_y_callable
```

```
    cdef object v_ort_x_callable
    cdef object v_ort_y_callable
    cdef object a_x_callable
    cdef object a_y_callable

    def __init__(self, object x_expr_in, object y_expr_in):
        # Use the symbolic variables defined at the module level of this .pyx file
        self.lam = lam_sym
        self.phi = phi_sym
        self.alpha = alpha_sym

        # Ensure x_expr_in and y_expr_in are defined using lam_sym, phi_sym from this module
        self.x_expr = x_expr_in
        self.y_expr = y_expr_in

        self.x_func_callable = sp.lambdify((self.lam, self.phi), self.x_expr, 'numpy')
        self.y_func_callable = sp.lambdify((self.lam, self.phi), self.y_expr, 'numpy')

        self.K_matrix = sp.Matrix([[sp.cos(self.phi)**-1,   0],
                                   [0,                      1]])
        self.M_matrix = sp.Matrix([self.x_expr, self.y_expr])
        self.vars_matrix = sp.Matrix([self.lam, self.phi])
        self.J_matrix = self.M_matrix.jacobian(self.vars_matrix)

        T_sym_expr = (self.J_matrix * self.K_matrix)
        # Lambdify requires a list of expressions if the matrix contains them
        T_sym_list = T_sym_expr.tolist()

        self.T_funcs_list = [[sp.lambdify((self.lam, self.phi), T_sym_list[i][j], 'numpy') for
        ↪  j in range(2)] for i in range(2)]

        self.Hx_matrix = sp.hessian(self.x_expr, (self.lam, self.phi))
        self.Hy_matrix = sp.hessian(self.y_expr, (self.lam, self.phi))

        self.w_vector = self.K_matrix * sp.Matrix([sp.tan(self.phi)*sp.sin(2*self.alpha),

                                                   ↪  sp.tan(self.phi)*(-(sp.cos(self.alpha)**2))])
        self.u_vector = self.K_matrix * sp.Matrix([sp.cos(self.alpha), sp.sin(self.alpha)])
        self.v_vector = self.J_matrix * self.u_vector

        self.v_ort_vector = sp.Matrix([0, 0]) # Create as Sympy Matrix
        self.v_ort_vector[0] = -self.v_vector[1]
        self.v_ort_vector[1] =  self.v_vector[0]

        # Ensure u_vector.T * Hx_matrix * u_vector results in a scalar expression for Matrix
        ↪  constructor
        ux_expr_scalar = (self.u_vector.T * self.Hx_matrix * self.u_vector)[0,0]
        uy_expr_scalar = (self.u_vector.T * self.Hy_matrix * self.u_vector)[0,0]
        self.a_vector = self.J_matrix * self.w_vector + sp.Matrix([ux_expr_scalar,
        ↪  uy_expr_scalar])

        self.v_x_callable     = sp.lambdify((self.lam, self.phi, self.alpha),
        ↪  self.v_vector[0], 'numpy')
        self.v_y_callable     = sp.lambdify((self.lam, self.phi, self.alpha),
        ↪  self.v_vector[1], 'numpy')
        self.v_ort_x_callable = sp.lambdify((self.lam, self.phi, self.alpha),
        ↪  self.v_ort_vector[0], 'numpy')
        self.v_ort_y_callable = sp.lambdify((self.lam, self.phi, self.alpha),
        ↪  self.v_ort_vector[1], 'numpy')
        self.a_x_callable     = sp.lambdify((self.lam, self.phi, self.alpha),
        ↪  self.a_vector[0], 'numpy')
```

```
    self.a_y_callable     = sp.lambdify((self.lam, self.phi, self.alpha),
    ↪  self.a_vector[1], 'numpy')



cpdef tuple singularities(self,
                          np.ndarray[np.double_t, ndim=1] lam_vals,
                          np.ndarray[np.double_t, ndim=1] phi_vals):

    cdef Py_ssize_t N = lam_vals.shape[0] # N is the length of the input 1D arrays
    if N == 0: # Handle empty inputs
        return np.array([], dtype=np.double), np.array([], dtype=np.double)

    # 1. Evaluate the lambdified functions.
    #    Results can be 1D arrays (if expr was non-constant) or Python scalars/0-D NumPy
    ↪  arrays (if expr was constant).
    _T00_eval = self.T_funcs_list[0][0](lam_vals, phi_vals)
    _T01_eval = self.T_funcs_list[0][1](lam_vals, phi_vals)
    _T10_eval = self.T_funcs_list[1][0](lam_vals, phi_vals)
    _T11_eval = self.T_funcs_list[1][1](lam_vals, phi_vals)

    # 2. Convert all evaluated components to NumPy arrays. This is crucial.
    #    If _Tij_eval is a Python scalar (e.g., 0 from a constant expression),
    #    np.array() will make it a 0-D NumPy array.
    #    If _Tij_eval is already a NumPy array (0-D or 1-D from a non-constant
    ↪  expression),
    #    this ensures it's a base ndarray of the correct dtype.
    _T00_np_raw = np.array(_T00_eval, dtype=np.double)
    _T01_np_raw = np.array(_T01_eval, dtype=np.double) # If _T01_eval was 0, _T01_np_raw
    ↪  is now array(0.0) (0-D)
    _T10_np_raw = np.array(_T10_eval, dtype=np.double)
    _T11_np_raw = np.array(_T11_eval, dtype=np.double)

    # 3. Broadcast these NumPy arrays together.
    #    If `lam_vals` was 1D (length N) and at least one of the `_Tij_np_raw` arrays
    #    is also 1D (length N) (e.g., from a non-constant symbolic expression like
    ↪  sec(phi)),
    #    while others are 0-D (scalars), `np.broadcast_arrays` will promote all scalars
    #    to 1D arrays of length N.
    #    The result, `_b_arrays_list`, will be a list of 1-D NumPy arrays.
    _b_arrays_list = np.broadcast_arrays(_T00_np_raw, _T01_np_raw, _T10_np_raw,
    ↪  _T11_np_raw)

    # 4. Now, assign to cdef typed 1-D arrays.
    #    The elements of _b_arrays_list are guaranteed to be at least 1-D if N > 0
    #    and broadcasting occurred as expected.
    cdef np.ndarray[np.double_t, ndim=1] T_00 = _b_arrays_list[0]
    cdef np.ndarray[np.double_t, ndim=1] T_01 = _b_arrays_list[1] # This should now
    ↪  receive a 1-D array
    cdef np.ndarray[np.double_t, ndim=1] T_10 = _b_arrays_list[2]
    cdef np.ndarray[np.double_t, ndim=1] T_11 = _b_arrays_list[3]

    # Robustness check: In an edge case where N=1 (input arrays have one element) AND
    # ALL symbolic T_ij components were constants (e.g., T-matrix is identity
    ↪  [[1,0],[0,1]]),
    # then all _Tij_np_raw would be 0-D. In this *specific* scenario,
    # np.broadcast_arrays(0D,0D,0D,0D) returns a list of 0-D arrays.
    # The cdef typing above would then fail. So, we must handle this.
    if N > 0 and T_00.ndim == 0:
        # This implies all T_ij were constants, and np.broadcast_arrays returned 0-D
        ↪  arrays.
```

```
        # We need to manually expand them to 1-D arrays of length N.
        T_00 = np.full(N, T_00.item(), dtype=np.double) # .item() extracts scalar from 0-D
        ↪  array
        T_01 = np.full(N, T_01.item(), dtype=np.double)
        T_10 = np.full(N, T_10.item(), dtype=np.double)
        T_11 = np.full(N, T_11.item(), dtype=np.double)
        # After this, T_00, T_01, T_10, T_11 are guaranteed to be 1-D.

    # Now, T_00, T_01, T_10, T_11 are definitely 1D arrays of length N.
    # (The original N came from lam_vals.shape[0])

    cdef np.ndarray[np.double_t, ndim=3] T_all = np.empty((N, 2, 2), dtype=np.double)
    T_all[:,0,0] = T_00
    T_all[:,0,1] = T_01
    T_all[:,1,0] = T_10
    T_all[:,1,1] = T_11

    # Batch-compute T^T T, trace, det using NumPy (already optimized)
    cdef np.ndarray[np.double_t, ndim=3] TtT = np.einsum('nij,nkj->nik', T_all, T_all)
    cdef np.ndarray[np.double_t, ndim=1] trace_T = np.trace(TtT, axis1=1, axis2=2)
    cdef np.ndarray[np.double_t, ndim=1] det_T = np.linalg.det(T_all)

    cdef np.ndarray[np.double_t, ndim=1] abs_det = np.abs(det_T)
    cdef np.ndarray[np.double_t, ndim=1] left_sqrt_arg = trace_T + 2 * abs_det
    # Ensure non-negative for sqrt, though theoretically trace_T + 2*|det_T| should be >=0
    np.maximum(left_sqrt_arg, 0, out=left_sqrt_arg)
    cdef np.ndarray[np.double_t, ndim=1] left = np.sqrt(left_sqrt_arg)

    cdef np.ndarray[np.double_t, ndim=1] right_sqrt_arg = trace_T - 2 * abs_det
    # Ensure non-negative argument for sqrt by clipping
    np.maximum(right_sqrt_arg, 0, out=right_sqrt_arg)
    cdef np.ndarray[np.double_t, ndim=1] right = np.sqrt(right_sqrt_arg)

    cdef np.ndarray[np.double_t, ndim=1] a_val = (left + right) / 2.0
    cdef np.ndarray[np.double_t, ndim=1] b_val = (left - right) / 2.0
    return a_val, b_val

cpdef tuple area_and_isotropy(self,
                              np.ndarray[np.double_t, ndim=1] lam_vals,
                              np.ndarray[np.double_t, ndim=1] phi_vals):
    a, b = self.singularities(lam_vals, phi_vals)
    cdef np.ndarray[np.double_t, ndim=1] a_arr = np.asarray(a, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=1] b_arr = np.asarray(b, dtype=np.double)

    if a_arr.size == 0 or b_arr.size == 0:
        return 0.0, 0.0

    cdef np.ndarray[np.double_t, ndim=1] ab_product = a_arr * b_arr
    # Filter out non-positive values for log
    cdef np.ndarray[np.double_t, ndim=1] ab_positive = ab_product[ab_product > 1e-12]
    cdef double A_val = np.std(np.log(ab_positive)) if ab_positive.size > 0 else 0.0

    cdef np.ndarray[np.double_t, ndim=1] a_over_b_ratio = np.zeros_like(a_arr)
    # Create mask for b_arr > 1e-12 to avoid division by zero and log issues
    cdef np.ndarray[np.uint8_t, ndim=1] mask_b_ok = (b_arr > 1e-12) & (a_arr > 1e-12) #
    ↪  ensure a is also positive for a/b > 0

    if np.any(mask_b_ok):
         a_over_b_ratio[mask_b_ok] = a_arr[mask_b_ok] / b_arr[mask_b_ok]
```

```
    cdef np.ndarray[np.double_t, ndim=1] a_over_b_positive = a_over_b_ratio[a_over_b_ratio
    ↪  > 1e-12]
    cdef double I_val = rms_py(np.log(a_over_b_positive)) if a_over_b_positive.size > 0
    ↪  else 0.0
    return A_val, I_val

cpdef tuple flexion_and_skewness(self,
                                 np.ndarray[np.double_t, ndim=1] lam_vals,
                                 np.ndarray[np.double_t, ndim=1] phi_vals,
                                 np.ndarray[np.double_t, ndim=1] alpha_vals):

    cdef np.ndarray[np.double_t, ndim=2] lam_grid = lam_vals[:, np.newaxis]
    cdef np.ndarray[np.double_t, ndim=2] phi_grid = phi_vals[:, np.newaxis]
    cdef np.ndarray[np.double_t, ndim=2] alpha_grid = alpha_vals[np.newaxis, :]

    Vx_arr    = self.v_x_callable(lam_grid, phi_grid, alpha_grid)
    Vy_arr    = self.v_y_callable(lam_grid, phi_grid, alpha_grid)
    Vortx_arr = self.v_ort_x_callable(lam_grid, phi_grid, alpha_grid)
    Vorty_arr = self.v_ort_y_callable(lam_grid, phi_grid, alpha_grid)
    Ax_arr    = self.a_x_callable(lam_grid, phi_grid, alpha_grid)
    Ay_arr    = self.a_y_callable(lam_grid, phi_grid, alpha_grid)

    cdef np.ndarray[np.double_t, ndim=2] Vx = np.asarray(Vx_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=2] Vy = np.asarray(Vy_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=2] Vortx = np.asarray(Vortx_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=2] Vorty = np.asarray(Vorty_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=2] Ax = np.asarray(Ax_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=2] Ay = np.asarray(Ay_arr, dtype=np.double)

    Vx, Vy, Vortx, Vorty, Ax, Ay = np.broadcast_arrays(Vx, Vy, Vortx, Vorty, Ax, Ay)
    if Vx.size == 0: # Handle empty inputs after broadcasting
         return np.array([], dtype=np.double), np.array([], dtype=np.double)

    cdef np.ndarray[np.double_t, ndim=3] Vf    = np.stack((Vx, Vy), axis=-1)
    cdef np.ndarray[np.double_t, ndim=3] Vortf = np.stack((Vortx, Vorty), axis=-1)
    cdef np.ndarray[np.double_t, ndim=3] Af    = np.stack((Ax, Ay), axis=-1)

    cdef np.ndarray[np.double_t, ndim=2] v_dot_a = np.sum(Vf * Af, axis=-1)
    cdef np.ndarray[np.double_t, ndim=2] v_ort_dot_a = np.sum(Vortf * Af, axis=-1)
    cdef np.ndarray[np.double_t, ndim=2] v_dot_v = np.sum(Vf * Vf, axis=-1)

    cdef np.ndarray[np.double_t, ndim=2] denom = np.maximum(v_dot_v, 1e-12)
    cdef np.ndarray[np.double_t, ndim=2] f_vals = np.abs(v_ort_dot_a) / denom
    cdef np.ndarray[np.double_t, ndim=2] s_vals = np.abs(v_dot_a) / denom

    return f_vals, s_vals

cpdef np.ndarray[np.double_t, ndim=1] distance(self,
                                               np.ndarray[np.double_t, ndim=1] lam_vals,
                                               np.ndarray[np.double_t, ndim=1] phi_vals):
    if lam_vals.size < 2:
        return np.array([], dtype=np.double)

    cdef np.ndarray[np.double_t, ndim=1] lam1 = lam_vals[:-1]
    cdef np.ndarray[np.double_t, ndim=1] lam2 = lam_vals[1:]
    cdef np.ndarray[np.double_t, ndim=1] phi1 = phi_vals[:-1]
    cdef np.ndarray[np.double_t, ndim=1] phi2 = phi_vals[1:]

    cdef np.ndarray[np.double_t, ndim=2] x1_coords = np.array([np.cos(phi1) *
    ↪  np.cos(lam1),
```

```
                                                        np.cos(phi1) *
                                                        ↪  np.sin(lam1),
                                                        np.sin(phi1)])
    cdef np.ndarray[np.double_t, ndim=2] x2_coords = np.array([np.cos(phi2) *
    ↪  np.cos(lam2),
                                                        np.cos(phi2) *
                                                        ↪  np.sin(lam2),
                                                        np.sin(phi2)])

    cdef np.ndarray[np.double_t, ndim=1] dot_product = np.sum(x1_coords * x2_coords,
    ↪  axis=0)
    dot_product = np.clip(dot_product, -1.0, 1.0)
    cdef np.ndarray[np.double_t, ndim=1] dist_globe = np.arccos(dot_product)

    x1m_arr = self.x_func_callable(lam1, phi1)
    y1m_arr = self.y_func_callable(lam1, phi1)
    x2m_arr = self.x_func_callable(lam2, phi2)
    y2m_arr = self.y_func_callable(lam2, phi2)

    cdef np.ndarray[np.double_t, ndim=1] x1m = np.asarray(x1m_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=1] y1m = np.asarray(y1m_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=1] x2m = np.asarray(x2m_arr, dtype=np.double)
    cdef np.ndarray[np.double_t, ndim=1] y2m = np.asarray(y2m_arr, dtype=np.double)

    cdef np.ndarray[np.double_t, ndim=1] dist_map = np.sqrt((x2m - x1m)**2 + (y2m -
    ↪  y1m)**2)

    cdef np.ndarray[np.double_t, ndim=1] dist_ratio = np.zeros_like(dist_map,
    ↪  dtype=np.double)
    # Create a boolean mask for where dist_globe is acceptably large
    cdef np.ndarray[np.uint8_t, ndim=1] mask_globe_nonzero = (dist_globe > 1e-12)

    dist_ratio[mask_globe_nonzero] = dist_map[mask_globe_nonzero] /
    ↪  dist_globe[mask_globe_nonzero]

    # Filter for valid ratios before log
    cdef np.ndarray[np.uint8_t, ndim=1] valid_mask_bool = np.isfinite(dist_ratio) &
    ↪  (dist_ratio > 1e-12)
    cdef np.ndarray[np.double_t, ndim=1] log_dist_valid =
    ↪  np.log(dist_ratio[valid_mask_bool])

    if log_dist_valid.size == 0:
        return np.array([], dtype=np.double)

    return log_dist_valid - np.mean(log_dist_valid)



cpdef double boundary_cut(self): # Original returns int, matching here.
    B = 0.25
    return B
```

c_distortions.py

```
    # c_distortions
# THIS CODE USES THE DISTORTION FUNCTIONS FROM THE CYTHON FILE
import numpy as np
import pandas as pd
from map_projections import projection_formulas
from tabulate import tabulate
import time
import distortions_cython


lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym

def rms(array): # Original rms
    return np.linalg.norm(array) / np.sqrt(array.size)


if __name__ == "__main__":
    start_time = time.time()

    # Choose projection
    projection = "Equirectangular"
    x_expr, y_expr = projection_formulas[projection]


    # Generate sample data
    N_points = 10**8
    lam_vals = np.random.uniform(-np.pi, np.pi, N_points)
    phi_vals = np.arcsin(np.random.uniform(-0.9999999, 0.9999999, N_points))
    alpha_vals = np.linspace(0, 2*np.pi, 360)


    mapping = distortions_cython.DistortionsCython(x_expr,y_expr)


    a_parts = []
    i_parts = []
    f_parts = []
    s_parts = []
    d_parts = []


    batch_size = 100000

    for j in range(0, N_points, batch_size):
        a, i = mapping.area_and_isotropy(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size])
        f, s = mapping.flexion_and_skewness(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size],
            alpha_vals)
        d =  mapping.distance(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size])

        a_parts.append(a)
        i_parts.append(i)
        f_parts.append(f)
```

```
        s_parts.append(s)
        d_parts.append(d)

    a_cy = np.mean(a_parts)
    i_cy = np.mean(i_parts)
    f_cy = np.concatenate(f_parts)
    s_cy = np.concatenate(s_parts)
    d_cy = np.concatenate(d_parts)


    df_cy = pd.DataFrame({
        "Area": [round(a_cy,5)],
        "Isotropy": [round(i_cy,5)],
        "Flexion": [round(np.mean(f_cy),5)],
        "Skewness": [round(np.mean(s_cy),5)],
        "Distance": [round(rms(d_cy),5)],
        "Boundary Cut": [mapping.boundary_cut()] # Example
    }).T.reset_index()

    df_cy.columns = ['Distortion', 'Value']

    print(tabulate(df_cy, headers = 'keys', tablefmt = 'presto'))

    print("--- Cython version took %s seconds ---" % (time.time() - start_time))
```

heatmapping.py

```
# heatmapping.py
# MAKES THE HEATMAPS
from matplotlib.colors import LogNorm
from map_projections import projection_formulas
from mpl_toolkits.axes_grid1 import make_axes_locatable
import distortions_cython
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
import geopandas as gpd

def rms(array):
    return np.linalg.norm(array) / np.sqrt(array.size)


#lam, phi, alpha = sp.symbols('lam phi alpha')

lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym



num_points = 1000
lam_vals = np.linspace(-np.pi, np.pi, num_points)
phi_vals = np.arcsin(np.linspace(-0.9999, 0.9999, num_points))
#phi_vals = np.linspace(-np.pi/2 + 0.0001, np.pi/2 - 0.0001, num_points)
alpha_vals = np.linspace(0, 2 * np.pi, 50)

Lam, Phi    = np.meshgrid(lam_vals, phi_vals)
L_flat      = Lam.ravel()
P_flat      = Phi.ravel()



def plot_coastlines(x_func, y_func):
    # Load the coastline shapefile
    gdf = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
    ↪  Technology/BEP/coastline_data/ne_10m_coastline.shp")  # <- adjust path

    for geom in gdf.geometry:
        if geom.geom_type == 'LineString':
            lon, lat = geom.xy
            lon = np.array(lon)
            lat = np.array(lat)
        elif geom.geom_type == 'MultiLineString':
            for line in geom:
                lon, lat = line.xy
                lon = np.array(lon)
                lat = np.array(lat)
                phi = np.radians(lat)
                lam = np.radians(lon)
                x = x_func(lam, phi)
                y = y_func(lam, phi)
                plt.plot(x, y, color='black', linewidth=0.5)
            continue
        else:
            continue

        phi = np.radians(lat)
```

```
        lam = np.radians(lon)

        lam = np.clip(np.radians(lon), -np.pi, np.pi)
        phi = np.clip(np.radians(lat), -np.pi/2+0.1, np.pi/2-0.1)

        x = x_func(lam, phi)
        y = y_func(lam, phi)

        plt.plot(x, y, color='black', linewidth=0.5)

def plot_heatmap(name : str, distortion : str, use_log : bool = False):
    x_expr, y_expr = projection_formulas[name]
    mapping = distortions_cython.DistortionsCython(x_expr, y_expr)

    if distortion == 'area':
        a, b = mapping.singularities(L_flat, P_flat)
        label = "Area Distortion"
        A_flat = np.log(a*b)
        values = A_flat.reshape(num_points, num_points)

    elif distortion == 'isotropy':
        a, b = mapping.singularities(L_flat, P_flat)
        label = "Isotropy Distortion"
        I_flat = np.log(a/b)
        values = I_flat.reshape(num_points, num_points)

    elif distortion == 'flexion':
        F_flat, S_flat = mapping.flexion_and_skewness(L_flat, P_flat, alpha_vals)
        label = "Flexion Distortion"
        F_mean = F_flat.mean(axis=1) # mean of alphas
        values = F_mean.reshape(num_points, num_points)

    elif distortion == 'skewness':
        F_flat, S_flat = mapping.flexion_and_skewness(L_flat, P_flat, alpha_vals)
        label = "Skewness Distortion"
        S_mean = S_flat.mean(axis=1) # mean of alphas
        values = S_mean.reshape(num_points, num_points)

    else:
        print("Not one of the distortions")
        return  # Exit early if distortion type is invalid

    x_func = sp.lambdify((lam, phi), x_expr, 'numpy')
    y_func = sp.lambdify((lam, phi), y_expr, 'numpy')

    X = x_func(Lam, Phi)
    Y = y_func(Lam, Phi)

    X_min = np.min(X)
    X_max = np.max(X)
    Y_min = np.min(Y)
    Y_max = np.max(Y)

    scale_norm = LogNorm(vmin=max(values.min(), 1e-3), vmax=values.max()) if use_log else None

    fig, ax = plt.subplots(dpi=300)
    pcm = ax.pcolormesh(X, Y, values, shading='auto', cmap='coolwarm', norm=scale_norm)
    plot_coastlines(x_func, y_func)

    # Create colorbar axis that matches height of the plot
    divider = make_axes_locatable(ax)
```

```
    cax = divider.append_axes("right", size="5%", pad=0.05)
    cbar = fig.colorbar(pcm, cax=cax, label=label)

    # Rest of the plot (meridians, parallels, formatting)
    num_lines = 13
    meridians = np.linspace(-np.pi, np.pi, num_lines)
    parallels = np.arcsin(np.linspace(-0.99, 0.99, 7))

    for lon in meridians:
        phi_line = np.arcsin(np.linspace(-0.99, 0.99, 500))
        lam_line = np.full_like(phi_line, lon)
        x_vals = x_func(lam_line, phi_line)
        y_vals = y_func(lam_line, phi_line)
        ax.plot(x_vals, y_vals, color='white', linestyle='-', linewidth=0.5)

    for lat in parallels:
        lam_line = np.linspace(-np.pi, np.pi, 500)
        phi_line = np.full_like(lam_line, lat)
        x_vals = x_func(lam_line, phi_line)
        y_vals = y_func(lam_line, phi_line)
        ax.plot(x_vals, y_vals, color='white', linestyle='-', linewidth=0.5)

    for spine in ax.spines.values():
        spine.set_visible(False)

    ax.axis('equal')
    ax.set_xlim(X_min, X_max)
    ax.set_ylim(Y_min, Y_max)
    ax.set_title(f'{name} {distortion} Heatmap')
    ax.tick_params(length=0)
    ax.axis('off')

    plt.tight_layout()
    #plt.savefig(f"Heatmaps/{name}_{distortion}.png")
    plt.show()

if __name__ == "__main__":
    proj_name = "Wiechel"
    plot_heatmap(proj_name, 'area', True)
    plot_heatmap(proj_name, 'isotropy', True)
    plot_heatmap(proj_name, 'flexion', True)
    plot_heatmap(proj_name, 'skewness', True)
```

plot_error

```python
# plot_error.py
# FIND NUMERICAL ERROR
from map_projections import projection_formulas
import distortions_cython
import numpy as np
import matplotlib.pyplot as plt


lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym



# Sampling functions
def sample_lam(N):
    return np.random.uniform(-np.pi, np.pi, N)
def sample_phi(N):
    # uniform in sin phi
    return np.arcsin(np.random.uniform(-0.99999999, 0.99999999, N))

def sample_alp(M):
    return np.linspace(0, 2 * np.pi, M)

def rms(array):
    return np.linalg.norm(array) / np.sqrt(array.size)

def batching(mapping, lam_vals, phi_vals, alpha_vals):
    a_parts = []
    i_parts = []
    f_parts = []
    s_parts = []
    d_parts = []

    N_points = len(lam_vals)

    batch_size = 100_000

    for j in range(0, N_points, batch_size):
        a, i = mapping.area_and_isotropy(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size])

        f, s = mapping.flexion_and_skewness(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size],
            alpha_vals)
        d =  mapping.distance(
            lam_vals[j:j+batch_size],
            phi_vals[j:j+batch_size])


        a_parts.append(a)
        i_parts.append(i)
        f_parts.append(f)
        s_parts.append(s)
        d_parts.append(d)

    a_cy = np.mean(a_parts)
    i_cy = np.mean(i_parts)
```

```
    f_cy = np.concatenate(f_parts)
    s_cy = np.concatenate(s_parts)
    d_cy = np.concatenate(d_parts)
    return a_cy, i_cy, np.mean(f_cy), np.mean(s_cy), rms(d_cy)


# Convergence test & plotting
def plot_error_lam_phi(x_expr, y_expr, title):
    mapping = distortions_cython.DistortionsCython(x_expr, y_expr)

    # reference with a large N
    N_ref = 10**8
    lam_ref = sample_lam(N_ref)
    phi_ref = sample_phi(N_ref)
    alpha_vals = sample_alp(10)

    A_ref, I_ref, F_ref, S_ref, D_ref = batching(mapping, lam_ref, phi_ref, alpha_vals)

    # sample sizes (log-spaced)
    Ns = np.unique(np.logspace(2, 6, 5, dtype=int))  # 10^2 to 10^6
    A_err = []
    I_err = []
    F_err = []
    S_err = []
    D_err = []

    for N in Ns:
        lam_s = sample_lam(N)
        phi_s = sample_phi(N)

        A, I, F, S, D = batching(mapping, lam_s, phi_s, alpha_vals)

        A_err.append(abs(A - A_ref))
        I_err.append(abs(I - I_ref))
        F_err.append(abs(F.mean() - F_ref))
        S_err.append(abs(S.mean() - S_ref))
        D_err.append(abs(D - D_ref))

    # plot
    plt.loglog(Ns, A_err, '-o', label='Area Error')
    plt.loglog(Ns, I_err, '-o', label='Isotropy Error')
    plt.loglog(Ns, F_err, '-o', label='Flexion Error')
    plt.loglog(Ns, S_err, '-o', label='Skewness Error')
    plt.loglog(Ns, D_err, '-o', label='Distance Error')
    plt.xlabel('Number of Samples (lam and phi)')
    plt.ylabel('Absolute Error')
    plt.title(f'Convergence for {title}')
    plt.grid(True, which='both', ls='--', alpha=0.5)
    plt.legend()
    plt.tight_layout()
    plt.show()


plt.figure(figsize=(8,6))
name ='Equirectangular'
x_expr, y_expr = projection_formulas[name]

plot_error_lam_phi(x_expr, y_expr, name)
```

get_extracted_coastline_latlon.py

```
# get_extracted_coastline_latlon.py
# EXTRACTS GEOPANDAS DATA AND CONVERTS TO ARRAYS OF LAMBDA AND PHI VALUES
import numpy as np
import os
import pandas as pd
import geopandas as gpd
import distortions_cython

coastline_data_10  = gpd.read_file("SET/PATH/ne_10m_coastline.shp")
coastline_data_110 = gpd.read_file("SET/PATH/ne_110m_coastline.shp")

#high
coastline_f_l1 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_f_l5 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L5.shp")  # Antarctica ice-front
↪  coastline
# Combine into one GeoDataFrame
coastline_f = gpd.GeoDataFrame(pd.concat([coastline_f_l1, coastline_f_l5], ignore_index=True),
↪  crs="EPSG:4326")

#medium
coastline_h_l1 = gpd.read_file("SET/PATH/GSHHS_shp/h/GSHHS_h_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_h_l5 = gpd.read_file(SET/PATH/GSHHS_shp/h/GSHHS_h_L5.shp")
coastline_h = gpd.GeoDataFrame(pd.concat([coastline_h_l1, coastline_h_l5], ignore_index=True),
↪  crs="EPSG:4326")

#low
coastline_i_l1 = gpd.read_file("SET/PATH/GSHHS_shp/i/GSHHS_i_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_i_l5 = gpd.read_file("SET/PATH/GSHHS_shp/i/GSHHS_i_L5.shp")
coastline_i = gpd.GeoDataFrame(pd.concat([coastline_i_l1, coastline_i_l5], ignore_index=True),
↪  crs="EPSG:4326")

lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym

# Coastline sample data
def extract_coastline_latlon(data):
    """
    Extracts coastline coordinates from a GeoDataFrame, ensuring the output array
    length is perfectly synchronized with the midpoint distance calculation.

    - For open LineStrings, it extracts all points.
    - For closed shapes (LineStrings, Polygon exteriors/interiors), it extracts
      only the unique points, discarding the duplicate closing coordinate.
    - It correctly handles single and multi-part geometries (MultiLineString, MultiPolygon).
    """
    gdf = data
    all_lons = []
    all_lats = []

    for geom in gdf.geometry:
        if geom is None or geom.is_empty:
            continue

        # --- Handle LineString and MultiLineString ---
        if geom.geom_type in ('LineString', 'MultiLineString'):
```

```
            # Treat a single LineString as a list containing one item for consistent
            ↪ processing
            lines = list(geom.geoms) if geom.geom_type == 'MultiLineString' else [geom]
            for line in lines:
                lon, lat = line.xy
                # A line needs at least 2 points to have a distance/weight
                if len(lon) < 2:
                    continue

                # Check if the line is closed
                if (lon[0], lat[0]) == (lon[-1], lat[-1]):
                    # For closed lines, weights are based on unique points. Drop the duplicate
                    ↪ endpoint.
                    all_lons.extend(lon[:-1])
                    clipped_lat = np.clip(lat, -89.9, 89.9)
                    all_lats.extend(clipped_lat[:-1])

                else:
                    # For open lines, a weight is calculated for each point. Keep all points.
                    all_lons.extend(lon)
                    clipped_lat = np.clip(lat, -89.9, 89.9)
                    all_lats.extend(clipped_lat)


        # --- Handle Polygon and MultiPolygon ---
        elif geom.geom_type in ('Polygon', 'MultiPolygon'):
            # Treat a single Polygon as a list containing one item for consistent processing
            polygons = list(geom.geoms) if geom.geom_type == 'MultiPolygon' else [geom]
            for poly in polygons:
                # Process the exterior boundary and all interior boundaries (holes)
                rings = [poly.exterior] + list(poly.interiors)
                for ring in rings:
                    lon, lat = ring.xy
                    # A valid ring needs at least 4 points (e.g., A-B-C-A)
                    if len(lon) < 4:
                        continue
                    clipped_lat = np.clip(lat, -89.9, 89.9)
                    # All polygon rings are closed. Drop the duplicate endpoint.
                    all_lons.extend(lon[:-1])
                    all_lats.extend(clipped_lat[:-1])


    lams = np.radians(np.array(all_lons))
    phis = np.radians(np.array(all_lats))


    return lams, phis


def get_extracted_coastline_10():
    if os.path.exists("lam_10.npy") and os.path.exists("phi_10.npy"):
        return np.load("lam_10.npy"), np.load("phi_10.npy")
    else:
        lam, phi = extract_coastline_latlon(coastline_data_10)
        np.save("lam_10.npy", lam)
        np.save("phi_10.npy", phi)
        return lam, phi

def get_extracted_coastline_110():
    if os.path.exists("lam_110.npy") and os.path.exists("phi_110.npy"):
        return np.load("lam_110.npy"), np.load("phi_110.npy")
```

```
    else:
        lam, phi = extract_coastline_latlon(coastline_data_110)
        np.save("lam_110.npy", lam)
        np.save("phi_110.npy", phi)
        return lam, phi

def get_extracted_coastline_high():
    if os.path.exists("lam_.npy") and os.path.exists("phi_.npy"):
        return np.load("lam_h.npy"), np.load("phi_h.npy")
    else:
        lam, phi = extract_coastline_latlon(coastline_f)
        np.save("lam_h.npy", lam)
        np.save("phi_h.npy", phi)
        return lam, phi

def get_extracted_coastline_medium():
    if os.path.exists("lam_.npy") and os.path.exists("phi_.npy"):
        return np.load("lam_m.npy"), np.load("phi_m.npy")
    else:
        lam, phi = extract_coastline_latlon(coastline_h)
        np.save("lam_m.npy", lam)
        np.save("phi_m.npy", phi)
        return lam, phi

def get_extracted_coastline_low():
    if os.path.exists("lam_.npy") and os.path.exists("phi_.npy"):
        return np.load("lam_l.npy"), np.load("phi_l.npy")
    else:
        lam, phi = extract_coastline_latlon(coastline_i)
        np.save("lam_l.npy", lam)
        np.save("phi_l.npy", phi)
        return lam, phi
```

FractalDimension.py

```
# FractalDimension.py
# COMPUTES FRACTAL DIMENSION
import numpy as np
import sympy as sp
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import linregress
import geopandas as gpd
from map_projections import projection_formulas

# SET YOU OWN PATH TO DOWNLOADED COASTLINE DATA
coastline_data_10  = gpd.read_file("SET/PATH/ne_10m_coastline.shp")
coastline_data_110 = gpd.read_file("SET/PATH/ne_110m_coastline.shp")

#high
coastline_f_l1 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_f_l5 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L5.shp")  # Antarctica ice-front
↪  coastline
# Combine into one GeoDataFrame
coastline_f = gpd.GeoDataFrame(pd.concat([coastline_f_l1, coastline_f_l5], ignore_index=True),
↪  crs="EPSG:4326")

#medium
coastline_h_l1 = gpd.read_file("SET/PATH/GSHHS_shp/h/GSHHS_h_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_h_l5 = gpd.read_file(SET/PATH/GSHHS_shp/h/GSHHS_h_L5.shp")
coastline_h = gpd.GeoDataFrame(pd.concat([coastline_h_l1, coastline_h_l5], ignore_index=True),
↪  crs="EPSG:4326")

#low
coastline_i_l1 = gpd.read_file("SET/PATH/GSHHS_shp/i/GSHHS_i_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_i_l5 = gpd.read_file("SET/PATH/GSHHS_shp/i/GSHHS_i_L5.shp")
coastline_i = gpd.GeoDataFrame(pd.concat([coastline_i_l1, coastline_i_l5], ignore_index=True),
↪  crs="EPSG:4326")

lam, phi = sp.symbols('lam phi')

def extract_coastline_latlon(data):
    gdf = data
    projection = "Equirectangular"
    x_func, y_func = projection_formulas[projection]
    x_expr = sp.lambdify((lam, phi), x_func, 'numpy')
    y_expr = sp.lambdify((lam, phi), y_func, 'numpy')
    all_lons = []
    all_lats = []

    for geom in gdf.geometry:
        if geom.geom_type == 'LineString':
            lon, lat = geom.xy
            all_lons.extend(lon)
            all_lats.extend(lat)

        elif geom.geom_type == 'MultiLineString':
            for line in geom:
                lon, lat = line.xy
                all_lons.extend(lon)
                all_lats.extend(lat)
```

```
        elif geom.geom_type == 'Polygon':
            coords = np.array(geom.exterior.coords)
            lon = coords[:,0]
            lat = coords[:,1]

            lat = np.clip(lat, -89.99, 89.99)

            all_lons.extend(lon)
            all_lats.extend(lat)

        elif geom.geom_type == 'MultiPolygon':
            for poly in geom.geoms:
                rings = [poly.exterior] + list(poly.interiors)
                for ring in rings:
                    lon, lat = ring.xy
                    all_lons.extend(lon)
                    all_lats.extend(lat)

    lams = np.radians(np.array(all_lons), dtype=np.float64)
    phis = np.radians(np.array(all_lats), dtype=np.float64)

    x = x_expr(lams, phis)
    y = y_expr(lams, phis)

    return x, y


def box_count(points, epsilons):
    counts = []
    for eps in epsilons:
        boxes = set()
        for x, y in points:
            ix = int(x / eps)
            iy = int(y / eps)
            boxes.add((ix, iy))
        counts.append(len(boxes))
    return counts


data = coastline_f
# coastline
x, y = extract_coastline_latlon(data)
x = x - np.min(x)
y = y - np.min(y)
x = x / np.max(abs(x))
y = y / np.max(abs(y))
points = np.column_stack((x,y))

epsilons = [2**(-i) for i in range(4, 15)]  # Smaller boxes for finer resolution
counts = box_count(points, epsilons)


log_eps = np.log2(1 / np.array(epsilons, dtype=np.float64))
log_counts = np.log2(np.array(counts, dtype=np.float64))
slope, intercept, r_value, p_value, std_err = linregress(log_eps, log_counts)

plt.figure(figsize=(8, 6))
plt.plot(log_eps, log_counts, 'o-', label=f"Dimension approx {slope:.6f}")
#plt.plot(s,t, color = "red")
plt.xlabel("log(1/epsilon)")
plt.ylabel("log(N(epsilon))")
```

```
plt.title("Box-Counting Dimension")
plt.legend()
plt.grid(True, alpha=0.2)
plt.tight_layout()
plt.show()

# Print final dimension result
print(f"Estimated Dimension: {slope:.6f}")
print(f"Standard Error: {std_err:.6f}")
print(f"R-squared: {r_value**2:.4f}")
```

Fractal_Dimension_Plot.py

```
# Fractal_Dimension_Plot.py
# COMPUTES AND VISUALISES LOCAL FRACTAL DIMENSION
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import geopandas as gpd
import sympy as sp
from scipy.stats import linregress # For calculating the slope
from map_projections import projection_formulas

# --- Define Sympy symbols for projection formulas ---
lam, phi = sp.symbols('lam phi')

# SET YOU OWN PATH
coastline_l1 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L1.shp")  # Global coastlines
↪  excluding Antarctica
coastline_l5 = gpd.read_file("SET/PATH/GSHHS_shp/f/GSHHS_f_L5.shp")  # Antarctica ice-front
↪  coastline

# Combine into one GeoDataFrame
coastline_data = gpd.GeoDataFrame(pd.concat([coastline_l1, coastline_l5], ignore_index=True),
↪  crs="EPSG:4326")


# --- Box_count function (remains largely the same) ---
def box_count_analysis(points, epsilons_list):
    counts = []
    if not points:
        return [0] * len(epsilons_list)
    for eps in epsilons_list:
        if eps <= 0:
            counts.append(0)
            continue
        boxes = set()
        for p_x, p_y in points:
            if np.isnan(p_x) or np.isnan(p_y):
                continue
            ix = int(p_x / eps)
            iy = int(p_y / eps)
            boxes.add((ix, iy))
        counts.append(len(boxes))
    return counts

# --- Define Epsilon values for Box Counting (for NORMALIZED [0,1] space) ---
EPSILONS_FOR_BOXCOUNT = np.array([2**(-i) for i in range(4, 15)], dtype=np.float64)
LOG_INV_EPSILONS = np.log2(1.0 / EPSILONS_FOR_BOXCOUNT) # Precompute log(1/eps)


def calculate_cell_slope(normalized_points_in_cell):
    """
    Calculates the slope of log_counts vs log(1/eps) for the given normalized points.
    """
    if not normalized_points_in_cell or len(normalized_points_in_cell) < 10:  # <--- Increase
    ↪  threshold from 2 to 10
        return 0.0  # or np.nan if you want to skip them entirely

    counts = box_count_analysis(normalized_points_in_cell, EPSILONS_FOR_BOXCOUNT)
    counts_arr = np.array(counts, dtype=np.float64)

    valid_mask = counts_arr > 0
```

```
    if np.sum(valid_mask) < 2:
        return 0.0  # again, use np.nan if you prefer to ignore in plot

    log_counts_valid = np.log2(counts_arr[valid_mask])
    log_inv_eps_valid = LOG_INV_EPSILONS[valid_mask]

    if len(log_counts_valid) < 2 or np.all(log_inv_eps_valid == log_inv_eps_valid[0]):
        return 0.0

    try:
        slope, *_ = linregress(log_inv_eps_valid, log_counts_valid)
        return slope
    except ValueError:
        return 0.0


# --- Function to extract raw coastline data (in radians) ---
def extract_raw_coastline_latlon(coastline_data):
    gdf = coastline_data
    all_lons_deg, all_lats_deg = [], []
    for geom in gdf.geometry:
        if geom is None or geom.is_empty: continue
        if geom.geom_type == 'LineString':
            lon, lat = geom.xy
            all_lons_deg.extend(lon); all_lats_deg.extend(lat)
            all_lons_deg.append(np.nan); all_lats_deg.append(np.nan)
        elif geom.geom_type == 'MultiLineString':
            for line in geom.geoms:
                if line is None or line.is_empty: continue
                lon, lat = line.xy
                all_lons_deg.extend(lon); all_lats_deg.extend(lat)
                all_lons_deg.append(np.nan); all_lats_deg.append(np.nan)
        elif geom.geom_type == 'Polygon':
            coords = np.array(geom.exterior.coords)
            lon = coords[:,0]
            lat = coords[:,1]

            lat = np.clip(lat, -89.99, 89.99)

            all_lons_deg.extend(lon); all_lats_deg.extend(lat)
            all_lons_deg.append(np.nan); all_lats_deg.append(np.nan)

        elif geom.geom_type == 'MultiPolygon':
            for poly in geom.geoms:
                rings = [poly.exterior] + list(poly.interiors)
                for ring in rings:
                    lon, lat = ring.xy
                    all_lons_deg.extend(lon); all_lats_deg.extend(lat)
                    all_lons_deg.append(np.nan); all_lats_deg.append(np.nan)
    return np.radians(np.array(all_lons_deg)), np.radians(np.array(all_lats_deg))

# FOR-LOOP OVER ALL MAP PROJECTIONS IN DICTIONARY
for proj_name, (x_expr, y_expr) in list(projection_formulas.items())[22:]:
    print(f"Processing: {proj_name}")
    # --- CHOOSE THE PROJECTION FOR THE ENTIRE MAP ---
    CHOSEN_PROJECTION = "Azimuthal Equidistant (North Polar)"
    if CHOSEN_PROJECTION not in projection_formulas:
        raise ValueError(f"Projection '{CHOSEN_PROJECTION}' not defined in
        ↪  projection_formulas.")
    x_formula_sym, y_formula_sym = projection_formulas[CHOSEN_PROJECTION]
    x_func = sp.lambdify((lam, phi), x_formula_sym, 'numpy')
```

```
y_func = sp.lambdify((lam, phi), y_formula_sym, 'numpy')
try:
    print("Loading and preparing all coastline data globally...")
    try:
        coastline_lon_rad_ALL, coastline_lat_rad_ALL =
        ↪  extract_raw_coastline_latlon(coastline_data)
        print(f"Successfully loaded
        ↪  {len(coastline_lon_rad_ALL[~np.isnan(coastline_lon_rad_ALL)])} raw coastline
        ↪  coordinate pairs.")
    except FileNotFoundError:
        print(f"ERROR: Coastline file not found at {coastline_data}. Using dummy data.")
        # Dummy data covering a small area for better testing of normalization
        lon_segment = np.linspace(-0.5, -0.4, 100)
        lat_segment = np.linspace(0.5, 0.6, 100)
        coastline_lon_rad_ALL = np.concatenate([lon_segment, [np.nan], lon_segment +
        ↪  0.05])
        coastline_lat_rad_ALL = np.concatenate([lat_segment, [np.nan], lat_segment +
        ↪  0.05])

    # Filter out NaNs used as separators, for processing
    valid_indices_ALL = ~np.isnan(coastline_lon_rad_ALL)
    lon_rad_proc = coastline_lon_rad_ALL[valid_indices_ALL]
    lat_rad_proc = coastline_lat_rad_ALL[valid_indices_ALL]

    if len(lon_rad_proc) == 0:
        print("No valid coastline data points found after filtering NaNs. Exiting.")
        exit()

    # Project all valid points
    x_proj_ALL = x_func(lon_rad_proc, lat_rad_proc)
    y_proj_ALL = y_func(lon_rad_proc, lat_rad_proc)

    # Filter out any NaNs/Infs that might result from projection (e.g., Mercator poles)
    finite_proj_mask = np.isfinite(x_proj_ALL) & np.isfinite(y_proj_ALL)
    x_proj_finite = x_proj_ALL[finite_proj_mask]
    y_proj_finite = y_proj_ALL[finite_proj_mask]
    # Important: Keep the original lat/lon for these finite points for later cell
    ↪  assignment
    lon_rad_finite = lon_rad_proc[finite_proj_mask]
    lat_rad_finite = lat_rad_proc[finite_proj_mask]

    if len(x_proj_finite) == 0:
        print("No valid coastline data points after projection and finite filtering.
        ↪  Exiting.")
        exit()

    # Global normalization of these finite projected points
    x_min_global = np.min(x_proj_finite)
    y_min_global = np.min(y_proj_finite)

    x_shifted_global = x_proj_finite - x_min_global
    y_shifted_global = y_proj_finite - y_min_global

    max_x_shifted = np.max(x_shifted_global)
    max_y_shifted = np.max(y_shifted_global)

    if max_x_shifted == 0: # All x projected points were the same
        x_norm_global = np.zeros_like(x_shifted_global)
    else:
        x_norm_global = x_shifted_global / max_x_shifted
```

```
if max_y_shifted == 0: # All y projected points were the same
    y_norm_global = np.zeros_like(y_shifted_global)
else:
    y_norm_global = y_shifted_global / max_y_shifted

print(f"Global normalisation complete. {len(x_norm_global)} points available for cell
↪  analysis.")
# Now, (lon_rad_finite, lat_rad_finite) correspond to (x_norm_global, y_norm_global)


# --- Setup for plotting grid ---
fig, ax = plt.subplots(figsize=(12, 9), dpi = 300)
ax.set_aspect('equal')
ax.set_facecolor('aliceblue')

num_lines = 13
meridians_plot = np.linspace(-np.pi, np.pi, num_lines)
max_lat_rad = np.deg2rad(85.05)
parallels_plot = np.arcsin(np.linspace(-np.sin(max_lat_rad), np.sin(max_lat_rad),
↪  num_lines))

# --- Plotting meridians and parallels ---
print("Plotting graticules...")
line_style_args = {'color': 'dimgray', 'linestyle': ':', 'linewidth': 0.8, 'alpha':
↪  0.7}
for lon_val in meridians_plot:
    phi_line = np.linspace(parallels_plot[0], parallels_plot[-1], 200)
    ax.plot(x_func(np.full_like(phi_line, lon_val), phi_line),
    ↪  y_func(np.full_like(phi_line, lon_val), phi_line), **line_style_args)
for lat_val in parallels_plot:
    lam_line = np.linspace(-np.pi, np.pi, 200)
    ax.plot(x_func(lam_line, np.full_like(lam_line, lat_val)), y_func(lam_line,
    ↪  np.full_like(lam_line, lat_val)), **line_style_args)

# --- Calculating slope for each cell ---
cell_meridians = meridians_plot
cell_parallels = parallels_plot
cell_values_slope = np.full((len(cell_parallels) - 1, len(cell_meridians) - 1),
↪  np.nan)

print(f"Calculating slopes for {cell_values_slope.size} cells...")
for i in range(len(cell_parallels) - 1):
    lat_bottom, lat_top = cell_parallels[i], cell_parallels[i+1]
    for j in range(len(cell_meridians) - 1):
        lon_left, lon_right = cell_meridians[j], cell_meridians[j+1]

        # Identify which of the *globally processed* points fall into this geographic
        ↪  cell
        mask_in_cell = (
            (lon_rad_finite >= lon_left) & (lon_rad_finite < lon_right) &
            (lat_rad_finite >= lat_bottom) & (lat_rad_finite < lat_top)
        )

        # Get the globally normalized coordinates for these points
        x_points_for_cell_norm = x_norm_global[mask_in_cell]
        y_points_for_cell_norm = y_norm_global[mask_in_cell]

        normalized_points_in_cell = list(zip(x_points_for_cell_norm,
        ↪  y_points_for_cell_norm))

        slope_for_cell = calculate_cell_slope(normalized_points_in_cell)
```

```
        cell_values_slope[i, j] = slope_for_cell

# --- Plotting the cell slope values using pcolormesh ---
X_pcolormesh_corners, Y_pcolormesh_corners = np.meshgrid(cell_meridians,
↪  cell_parallels)
X_pcolormesh_transformed = x_func(X_pcolormesh_corners, Y_pcolormesh_corners)
Y_pcolormesh_transformed = y_func(X_pcolormesh_corners, Y_pcolormesh_corners)

if np.any(np.isnan(X_pcolormesh_transformed)) or
↪  np.any(np.isnan(Y_pcolormesh_transformed)):
    print("Warning: NaN values found in transformed pcolormesh corners for
    ↪  graticules.")

if np.nansum(np.isfinite(cell_values_slope)) > 0: # Check if there are any valid
↪  slopes
    print("Plotting cell slope values with pcolormesh...")
    min_slope, max_slope = np.nanmin(cell_values_slope), np.nanmax(cell_values_slope)

    cmap = plt.cm.coolwarm # Example colormap for slopes (diverging might be good)
    cmap.set_bad(color='lightgrey', alpha=0.4)

    # Handle normalization for color scale carefully, especially if range is small or
    ↪  all NaNs
    if np.isnan(min_slope) or np.isnan(max_slope) or min_slope == max_slope:
        norm = plt.Normalize(vmin= (min_slope - 0.1) if not np.isnan(min_slope) else
        ↪  0,
                             vmax= (max_slope + 0.1) if not np.isnan(max_slope) else
                             ↪  1) # Default range
        if not np.isnan(min_slope) and min_slope == max_slope:
            norm = plt.Normalize(vmin=min_slope - 0.1, vmax=max_slope + 0.1) # Center
            ↪  single value

    else:
        norm = plt.Normalize(vmin=min_slope, vmax=max_slope)

    quadmesh = ax.pcolormesh(
        X_pcolormesh_transformed, Y_pcolormesh_transformed, cell_values_slope,
        cmap=cmap, norm=norm,
        alpha=0.75, shading='flat'
    )
    if not (np.isnan(min_slope) or np.isnan(max_slope)):
         cb = fig.colorbar(quadmesh, ax=ax, label="Slope (Box-Counting Dimension)",
         ↪  orientation="vertical", shrink=0.7)
         if min_slope == max_slope: cb.set_ticks([min_slope])
    else:
        print("Skipping colorbar as min/max of slope values are NaN or no data.")
else:
    print("No valid slope values to plot with pcolormesh or all slopes are NaN.")

# --- Transform and plot the actual coastline data (using original projection, not
↪  normalization) ---
print("Transforming and plotting coastlines...")
x_coast_transformed = x_func(coastline_lon_rad_ALL, coastline_lat_rad_ALL) # Use ALL
↪  version with NaNs
y_coast_transformed = y_func(coastline_lon_rad_ALL, coastline_lat_rad_ALL)
ax.plot(x_coast_transformed, y_coast_transformed, color='black', linewidth=0.9,
↪  label="Coastline")

# --- Final plot adjustments ---
ax.set_title(f"Map {CHOSEN_PROJECTION} with Box-Counting Dimension (Slope)")
valid_x_graticule = X_pcolormesh_transformed[~np.isnan(X_pcolormesh_transformed)]
```

```
        valid_y_graticule = Y_pcolormesh_transformed[~np.isnan(Y_pcolormesh_transformed)]
        if len(valid_x_graticule) > 0 and len(valid_y_graticule) > 0:
            ax.set_xlim(np.min(valid_x_graticule), np.max(valid_x_graticule))
            y_min_plot = np.percentile(valid_y_graticule, 0.5) if len(valid_y_graticule) > 1
             ↪  else np.min(valid_y_graticule)
            y_max_plot = np.percentile(valid_y_graticule, 99.5) if len(valid_y_graticule) > 1
             ↪  else np.max(valid_y_graticule)
            if y_min_plot < y_max_plot: ax.set_ylim(y_min_plot, y_max_plot)
            else: ax.set_ylim(np.min(valid_y_graticule), np.max(valid_y_graticule))

        plt.xlabel("Projected X-coordinate")
        plt.ylabel("Projected Y-coordinate")
        plt.grid(True, linestyle='--', alpha=0.2)
        plt.tight_layout()
        #plt.show()
        plt.savefig(f"Fractal_Plots/{CHOSEN_PROJECTION}.png")
        plt.close()

    except Exception as e:
        print(f"Failed for {proj_name}: {e}")
```

calc_midpoint_distances.py

```
# calc_midpoint_distances.py
# CALCULATES THE DISTANCES BETWEEN POINTS FROM THE COASTLINE DATA USING THE MIDPOINT RULE
# TO NORMALISE THE DISTORTION MEASURES ALONG THE ENTIRE COASTLINE
import numpy as np
import os
import pandas as pd
import geopandas as gpd
from pyproj import Geod
import distortions_cython

# SET YOU OWN PATH TO DOWNLOADED COASTLINE DATA
coastline_data_10  = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/ne_10m_coastline.shp")
coastline_data_110 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/ne_110m_coastline.shp")

#best
coastline_f_l1 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/f/GSHHS_f_L1.shp")  # Global coastlines excluding
↪  Antarctica
coastline_f_l5 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/f/GSHHS_f_L5.shp")  # Antarctica ice-front
↪  coastline
# Combine into one GeoDataFrame
coastline_f = gpd.GeoDataFrame(pd.concat([coastline_f_l1, coastline_f_l5], ignore_index=True),
↪  crs="EPSG:4326")

#medium
coastline_h_l1 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/h/GSHHS_h_L1.shp")  # Global coastlines excluding
↪  Antarctica
coastline_h_l5 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/h/GSHHS_h_L5.shp")
coastline_h = gpd.GeoDataFrame(pd.concat([coastline_h_l1, coastline_h_l5], ignore_index=True),
↪  crs="EPSG:4326")

#low
coastline_i_l1 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/i/GSHHS_i_L1.shp")  # Global coastlines excluding
↪  Antarctica
coastline_i_l5 = gpd.read_file("C:/Users/daanb/OneDrive - Delft University of
↪  Technology/BEP/coastline_data/GSHHS_shp/i/GSHHS_i_L5.shp")
coastline_i = gpd.GeoDataFrame(pd.concat([coastline_i_l1, coastline_i_l5], ignore_index=True),
↪  crs="EPSG:4326")


lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym

geod = Geod(ellps="WGS84") #best current model of calculatin distances, but maybe a perfect
↪  sphere is better


# It is important to check which linestrings are closed such that we don't ignore coastlines
# Therefore a if statement for lon[0] == lon[-1] is applied
# These non closed pieces are harder to evaluate because of the midpoint rule
# At the endpoints midrule distance = distance to next/previous point
# Also very important!! Positions of distances must stay the same
# Say we have distances (closed) = [3, 4, 5], then a -> b = 3, b -> c = 4, c -> a = 5
```

```python
# So when determining midpoint, we must get [4, 3.5, 4.5] since the coordinates must stay on
↪  same indices
# See it as: we went from right point rule to midpoint rule. Therefore % (modulo) is important
def calc_midpoint_dist_coastline(data):
    """
    Calculates the midpoint distance for coastline segments from a GeoDataFrame.

    This function is extended to handle LineString, Polygon, and MultiPolygon geometries.
    - For LineStrings, it differentiates between closed and open shapes.
    - For Polygons, it treats the exterior and any interior rings as closed coastlines.
    - For MultiPolygons, it processes each component polygon individually.

    Args:
        data (gpd.GeoDataFrame): A GeoDataFrame containing coastline geometries.

    Returns:
        np.array: A NumPy array of all calculated midpoint distances.
    """
    gdf = data
    midpoint_distances = []

    for geom in gdf.geometry:
        # Skip empty or invalid geometries
        if geom is None or geom.is_empty:
            continue

        geom_midpoints = []

        # --- Handle LineString Geometries ---
        if geom.geom_type == 'LineString':
            distances = []
            lon, lat = geom.xy

            # Calculate segment distances
            for i in range(1, len(lon)):
                lon1, lat1, lon2, lat2 = lon[i-1], lat[i-1], lon[i], lat[i]
                dist = geod.inv(lon1, lat1, lon2, lat2)[2]
                distances.append(dist)

            if not distances:
                continue

            # Check if the LineString is closed
            if (lon[0], lat[0]) == (lon[-1], lat[-1]):
                # Apply midpoint rule with circular indexing for closed lines
                geom_midpoints = [(distances[(j-1) % len(distances)] + distances[j]) / 2 for j
                ↪  in range(len(distances))]
            else:
                # Apply midpoint rule for open lines, preserving endpoints
                midpoints = [distances[0]]
                for k in range(1, len(distances)):
                    mid = (distances[k-1] + distances[k]) / 2
                    midpoints.append(mid)
                midpoints.append(distances[-1])
                geom_midpoints = midpoints

        # --- Handle Polygon Geometries (NEW) ---
        elif geom.geom_type == 'Polygon':
            # Process the exterior boundary as a closed ring
            rings = [geom.exterior] + list(geom.interiors)
            for ring in rings:
```

```
                distances = []
                lon, lat = ring.xy
                for i in range(1, len(lon)):
                    lon1, lat1, lon2, lat2 = lon[i-1], lat[i-1], lon[i], lat[i]
                    dist = geod.inv(lon1, lat1, lon2, lat2)[2]
                    distances.append(dist)

                if not distances:
                    continue

                # Apply midpoint rule with circular indexing
                midpoints = [(distances[(j-1) % len(distances)] + distances[j]) / 2 for j in
                ↪  range(len(distances))]
                geom_midpoints.extend(midpoints)

        # --- Handle MultiPolygon Geometries (NEW) ---
        elif geom.geom_type == 'MultiPolygon':
            for poly in geom.geoms:
                rings = [poly.exterior] + list(poly.interiors)
                for ring in rings:
                    distances = []
                    lon, lat = ring.xy
                    for i in range(1, len(lon)):
                        lon1, lat1, lon2, lat2 = lon[i-1], lat[i-1], lon[i], lat[i]
                        dist = geod.inv(lon1, lat1, lon2, lat2)[2]
                        distances.append(dist)

                    if not distances:
                        continue

                    # Apply midpoint rule with circular indexing
                    midpoints = [(distances[(j-1) % len(distances)] + distances[j]) / 2 for j
                    ↪  in range(len(distances))]
                    geom_midpoints.extend(midpoints)

        midpoint_distances.extend(geom_midpoints)

    return np.array(midpoint_distances)


#dist_10  = calc_midpoint_dist_coastline(coastline_data_10)
#dist_110 = calc_midpoint_dist_coastline(coastline_data_110)
#dist_best = calc_midpoint_dist_coastline(coastline_data_best)


def get_dist_10():
    if os.path.exists("dist_10.npy"):
        return np.load("dist_10.npy")
    else:
        data = coastline_data_10
        dist = calc_midpoint_dist_coastline(data)
        np.save("dist_10.npy", dist)
        return dist

def get_dist_110():
    if os.path.exists("dist_110.npy"):
        return np.load("dist_110.npy")
    else:
        data = coastline_data_110
        dist = calc_midpoint_dist_coastline(data)
        np.save("dist_110.npy", dist)
```

```
        return dist

def get_dist_high():
    if os.path.exists("dist_h.npy"):
        return np.load("dist_h.npy")
    else:
        data = coastline_f
        dist = calc_midpoint_dist_coastline(data)
        np.save("dist_h.npy", dist)
        return dist

def get_dist_medium():
    if os.path.exists("dist_m.npy"):
        return np.load("dist_m.npy")
    else:
        data = coastline_h
        dist = calc_midpoint_dist_coastline(data)
        np.save("dist_m.npy", dist)
        return dist

def get_dist_low():
    if os.path.exists("dist_l.npy"):
        return np.load("dist_l.npy")
    else:
        data = coastline_i
        dist = calc_midpoint_dist_coastline(data)
        np.save("dist_l.npy", dist)
        return dist
```

coastline_distortions.py

```
# coastline_distortions.py
# COMPUTES THE DISTORTION ALONG THE ENTIRE COASTLINE
import numpy as np
import pandas as pd
from map_projections import projection_formulas
# IMPORT MIDPOINT DISTANCES AND COASTLINE DATA
from calc_midpoint_distances import get_dist_10, get_dist_110, get_dist_high, get_dist_medium,
↪  get_dist_low
from get_extracted_coastline_latlon import get_extracted_coastline_10,
↪  get_extracted_coastline_110,
↪  get_extracted_coastline_high,get_extracted_coastline_medium,get_extracted_coastline_low
from tabulate import tabulate
import distortions_cython

lam = distortions_cython.lam_sym
phi = distortions_cython.phi_sym
alpha = distortions_cython.alpha_sym

def rms(array): # Original rms
    return np.linalg.norm(array) / np.sqrt(array.size)

def evaluate_projection(projection_name, lam_vals, phi_vals, dist, alpha_vals=np.linspace(0,
↪  2*np.pi, 180)):
    x_expr, y_expr = projection_formulas[projection_name]
    mapping = distortions_cython.DistortionsCython(x_expr, y_expr)

    N_points = len(lam_vals)
    batch_size = 100000

    a_parts = []
    i_parts = []
    f_parts = []
    s_parts = []

    for j in range(0, N_points, batch_size):
        a, b = mapping.singularities(lam_vals[j:j+batch_size], phi_vals[j:j+batch_size])
        f, s = mapping.flexion_and_skewness(lam_vals[j:j+batch_size],
        ↪  phi_vals[j:j+batch_size], alpha_vals)

        eps = 1e-12
        safe_a = np.clip(a, eps, None)
        safe_b = np.clip(b, eps, None)


        a_parts.extend(np.log(safe_a * safe_b))
        i_parts.extend(np.log(safe_a / safe_b))
        f_parts.extend(f.mean(axis=1))
        s_parts.extend(s.mean(axis=1))

    a_arr = np.array(a_parts)
    i_arr = np.array(i_parts)
    f_arr = np.array(f_parts)
    s_arr = np.array(s_parts)

    a_arr = np.nan_to_num(a_arr, nan=0.0, posinf=0.0, neginf=0.0)
    i_arr = np.nan_to_num(i_arr, nan=0.0, posinf=0.0, neginf=0.0)
    f_arr = np.nan_to_num(f_arr, nan=0.0, posinf=0.0, neginf=0.0)
    s_arr = np.nan_to_num(s_arr, nan=0.0, posinf=0.0, neginf=0.0)
    dist  = np.nan_to_num(dist,  nan=0.0, posinf=0.0, neginf=0.0)
```

```
    dist = np.array(dist)
    dist[dist > 10000] = 0

    A_mean = sum(a_arr * dist) / sum(dist)
    A = np.sqrt(sum((a_arr - A_mean)**2 * dist) / sum(dist))
    I = np.sqrt(sum(i_arr**2 * dist) / sum(dist))
    F = sum(f_arr * dist) / sum(dist)
    S = sum(s_arr * dist) / sum(dist)
    total = A**2 + I**2 + F**2 + S**2

    return {
        "Projection": projection_name,
        "Area": round(A, 3),
        "Isotropy": round(I, 3),
        "Flexion": round(F, 3),
        "Skewness": round(S, 3),
        "Total": round(total, 3)
    }


# CHOOSE LOW, MEDIUM OR HIGH RESOLUTION DATA
lam_vals, phi_vals = get_extracted_coastline_low()
dist = get_dist_low()

results = []

for proj_name in list(projection_formulas.keys())[20:]:
    print(f"Processing: {proj_name}")
    try:
        metrics = evaluate_projection(proj_name, lam_vals, phi_vals, dist)
        results.append(metrics)
    except Exception as e:
        print(f"Failed for {proj_name}: {e}")

df = pd.DataFrame(results)
df_sorted = df.sort_values("Total")

df_sorted.to_latex("projection_results.tex", index=False, escape=False)
df_sorted.to_excel("projection_results.xlsx", index=False)
```