VILNIUS UNIVERSITY
FACULTY OF CHEMISTRY AND GEOSCIENCES
DEPARTMENT OF CARTOGRAPHY AND GEOINFORMATICS

Motiejus Jakštys

A Thesis Presented for the Degree of Master in Cartography

WANG–MÜLLER ALGORITHM REALIZATION FOR CARTOGRAPHIC LINE GENERALIZATION

Supervisor Dr. Andrius Balčiūnas

2021-05-20 (revision 6c9b5e24603c)

Abstract

Currently available line simplification algorithms are rooted in mathematics and geometry, and are unfit for bendy map features like rivers and coastlines. Wang and Müller observed how cartographers simplify these natural features and created an algorithm. We implemented this algorithm and documented it in great detail. Our implementation makes Wang–Müller algorithm freely available in PostGIS, and this paper explains it.

Šiuo metu esami linijų supaprastinimo algoritmai yra kilę iš matematikos ir geometrijos, bet nėra tinkami lankstiems geografiniams objektams, tokiems kaip upės ir pakrantės, atvaizduoti. Wang ir Müller ištyrė, kaip kartografai atlieka upių generalizaciją, ir sukūrė algoritmą. Mes realizavome šį algoritmą ir išsamiai jį dokumentavome. Mūsų Wang–Müller realizacija ir dokumentacija yra nemokamos ir laisvai prieinamos, naudojant PostGIS platformą.

Contents

List of Tables

List of Listings

1 Introduction

When creating small-scale maps, often the detail of the data source is greater than desired for the map. While many features can be removed or simplified, it is more tricky with natural features that have many bends, like coastlines, rivers, or forest boundaries.

To create a small-scale map from a large-scale data source, features need to be simplified, i.e., detail should be reduced. While performing the simplification, it is important to retain the "defining" shape of the original feature. Otherwise, if the simplified feature looks too different from the original, the result will look unrealistic. Simplification problem for some objects can often be solved by non-geometric means:

- Towns and cities can be filtered by the number of inhabitants.
- Roads can be eliminated by the road length, number of lanes, or classification of the road (local, regional, international).

However, things are not as simple for natural features like rivers or coastlines. If a river is nearly straight, it should remain such after simplification. An overly straightened river will look like a canal, and the other way around — too curvy would not reflect the natural shape. Conversely, if the river originally is highly wiggly, the number of bends should be reduced, but not removed altogether. Natural line simplification problem can be viewed as a task of finding a delicate balance between two competing goals:

- Reduce detail by removing or simplifying "less important" features.
- Retain enough detail, so the original is still recognizable.

Given the discussed complexities with natural features, a fine line between under-simplification (leaving an object as-is) and over-simplification (making a straight line) needs to be found. Therein lies the complexity of simplification algorithms: all have different trade-offs.

The purpose of the thesis is to implement a cartographic line generalization algorithm on the basis of Wang–Müller algorithm, using open-source software. Tasks:

- Evaluate existing line simplification algorithms.
- Identify main river generalization problems, using classical line simplification algorithms.
- Define the method of the Wang–Müller technical implementation.
- Realize Wang–Müller algorithm technically, explaining the geometric transformations in detail.
- Apply the created algorithm for different datasets and compare the results with national datasets.

Scientific relevance of this work — the simplification processes (steps) described by the Wang–Müller algorithm — are analyzed in detail, practically implemented, and the implementation is described. That expands the knowledge of cartographic theory about the generalization of natural objects' boundaries after their natural defining properties.

In the original Wang–Müller article introducing the algorithm, the steps are not detailed in a way that can be put into practice for specific data; the steps are specified in this work. Practically, this work makes it possible to use open-source software to perform cartographic line generalization. The developed specialized cartographic line simplification algorithm can be applied by cartographers to implement automatic data generalization solutions. Given the open-source nature of this work, the algorithm implementation can be modified freely.

2 Literature Review And Problematic

2.1 Available Algorithms

This section reviews the classical line simplification algorithms, which, besides being around for a long time, offer easily accessible implementations, as well as more modern ones, which only theorize, but do not provide an implementation.

2.1.1 Douglas & Peucker, Visvalingam–Whyatt and Chaikin's

Douglas & Peucker[1] and Visvalingam–Whyatt[2] are "classical" line simplification computer graphics algorithms. They are relatively simple to implement and require few runtime resources. Both of them accept a single parameter based on desired scale of the map, which makes them straightforward to adjust for different scales.

Both algorithms are available in PostGIS, a free-software GIS suite:

- Douglas & Peucker via PostGIS ST_SIMPLIFY.
- Visvalingam–Whyatt via PostGIS ST_SIMPLIFYVW.

It may be worthwhile to post-process those through Chaikin's line smoothing algorithm[3] via PostGIS ST_CHAIKINSMOOTHING.

In generalization examples, we will use two rivers: Šalčia and Visinčia. These rivers were chosen because they have both large and small bends, and thus are convenient to analyze for both small- and large-scale generalization. Figure 1 on page 6 illustrates the original two rivers without any simplification.

Same rivers, unprocessed but in higher scales (1:50 000 and 1:250 000), are depicted in Figure 2. Some river features are so compact that a reasonably thin line depicting the river is touching itself, creating a thicker line. We can assume that some simplification for scale 1:50 000 and especially for 1:250 000 is worthwhile.

Figure 3 illustrates the same river bend, but simplified using Douglas & Peucker and Visvalingam–Whyatt algorithms. The resulting lines are jagged, and thus the resulting line looks unlike a real river. To smoothen the jaggedness, traditionally, Chaikin's[3] is applied after generalization, illustrated in Figure 4.

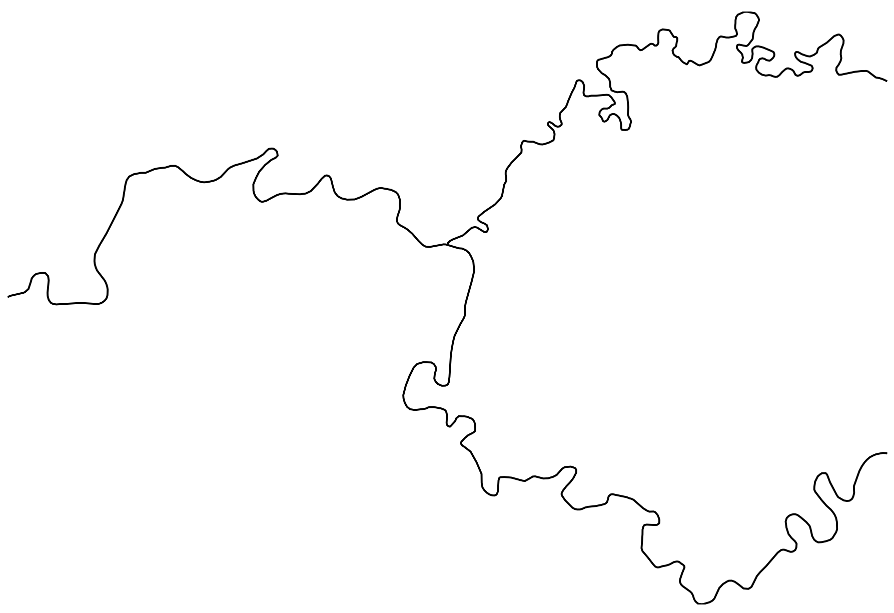


Figure 1: Example rivers for visual tests (1:25 000).

The resulting simplified and smoothened example (Figure 4 on page 7) yields a more aesthetically pleasing result; however, it obscures natural river features.

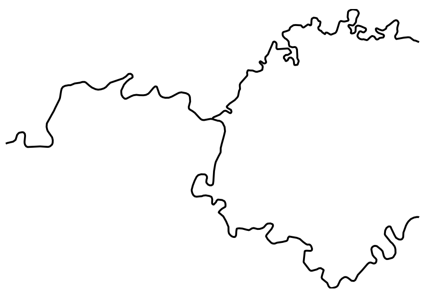
Given the absence of rocks, the only natural features that influence the river direction are topographic:

- Relatively straight river (completely straight or with small-angled bends over a relatively long distance) implies greater slope, more water, and/or faster flow.
- Bendy river, on the contrary, implies slower flow, slighter slope, and/or less water.

Both Visvalingam–Whyatt and Douglas & Peucker have a tendency to remove the small bends altogether, removing a valuable characterization of the river.

Sometimes low-water rivers in slender slopes have many bends next to each other. In low resolutions (either in small-DPI screens or paper, or when the river is sufficiently zoomed out, or both), the small bends will amalgamate to a unintelligible blob. Figure 6 illustrates a real-world example where a bendy river, normally 1 or 2 pixels wide, creates a wide area, of which the shapes of the bend become unintelligible. In this example, classical algorithms would remove these bends altogether. A cartographer would retain a few of those distinctive bends, but would increase the distance between the bends, remove some of the bends, or both.

For the reasons discussed in this section, the "classical" Douglas & Peucker and Visvalingam–Whyatt are not well-suited for natural river generalization, and a more robust line generalization algorithm is worthwhile to look for.


(a) Example scaled 1:50 000.

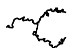
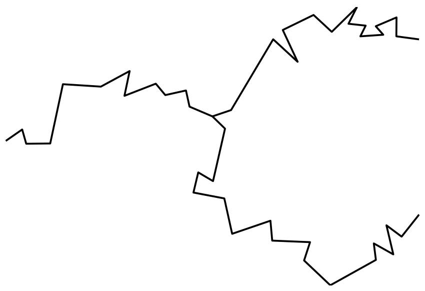

(b) Example scaled 1:250 000.

Figure 2: Down-scaled original river.


(a) Using Douglas & Peucker.

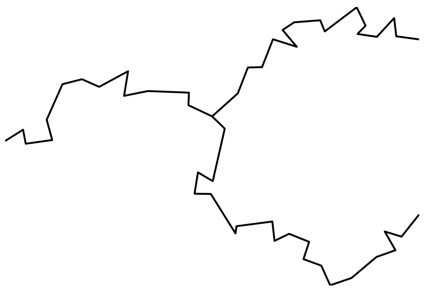
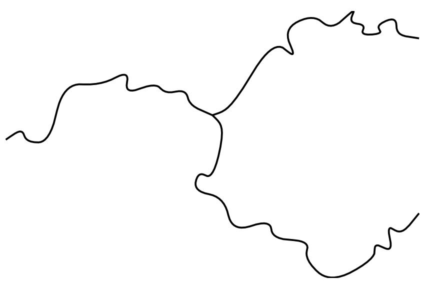

(b) Using Visvalingam–Whyatt.

Figure 3: Simplified using classical algorithms (1:50 000).


(a) Douglas & Peucker and Chaikin's.

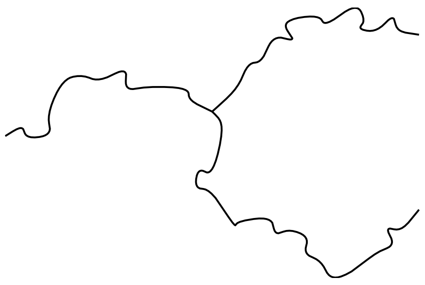
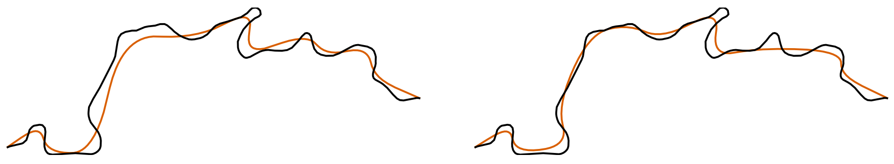

(b) Visvalingam–Whyatt and Chaikin's.

Figure 4: Simplified and smoothened river (1:50 000).



(a) Original (fig. 2a) and simplified (fig. 4a).

(b) Original (fig. 2a) and simplified (fig. 4b.)

Figure 5: Zoomed-in simplified and smoothened river and original.

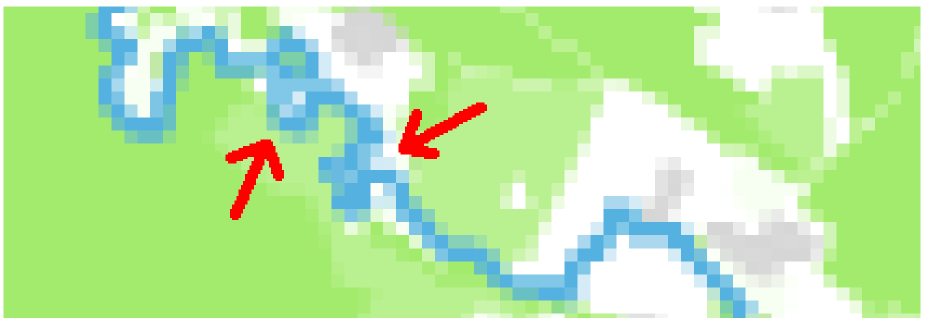


Figure 6: Narrow bends amalgamating into thick unintelligible blobs.

2.1.2 Modern Approaches

Due to their simplicity and ubiquity, Douglas & Peucker and Visvalingam–Whyatt have been established as go-to algorithms for line generalization. During recent years, alternatives have emerged. These modern replacements fall into roughly two categories:

- Cartographic knowledge was encoded to an algorithm (bottom-up approach). One among these are "Line generalization based on analysis of shape characteristics"[4], also known as Wang–Müller's algorithm.
- Mathematical shape transformation which yields a more cartographic result. E.g., "Line simplification using self-organizing maps"[5], "Simultaneous curve simplification"[6], "Dynamic simplification and visualization of large maps"[7], "Morphing polylines: A step towards continuous generalization"[8], "A New Algorithm for Cartographic Simplification of Streams and Lakes Using Deviation Angles and Error Bands"[9].

Authors of most of the aforementioned articles have implemented the generalization algorithm, at least to generate the illustrations in the articles. However, code is not available for evaluation with a desired dataset, much less for use as a basis for creating new maps. To the author's knowledge, Wang–Müller[4] is available in a commercial product, but requires a purchase of the commercial product suite, without a way to license the standalone algorithm.

Wang–Müller algorithm was created by encoding professional cartographers' knowledge into a computer algorithm. It has a few main properties which make it especially suitable for generalization of natural linear features:

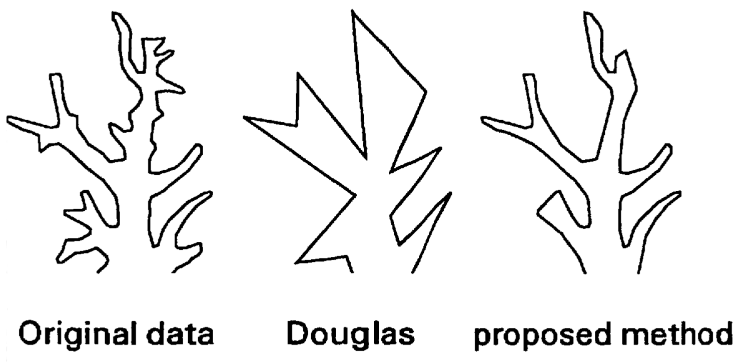


Figure 7: Figure 12.5 in [4]: example of cartographic line generalization.

- Small bends are not always removed, but either combined (e.g., 3 bends into 2), exaggerated, or removed, depending on the neighboring bends.
- Long and gentle bends are not straightened, but kept as-is.

As a result of these properties, Wang–Müller algorithm retains the defining properties of the natural features: high-current rivers keep their appearance as such, instead of becoming canals; low-stream bendy rivers retain their frequent small bends.

Figure 7, sub-figure labeled "proposed method" (from the original "Line generalization based on analysis of shape characteristics"[4]) illustrates the Wang–Müller algorithm.

2.2 Problematic with Generalization of Rivers

This section introduces the reader to simplification and generalization, and discusses two main problems with current-day automatic cartographic line generalization:

- Currently available line simplification algorithms were created to simplify geometries, but do not encode cartographic knowledge.
- Existing cartographic line generalization algorithms are not freely accessible.

2.2.1 Simplification versus Generalization

It is important to note the distinction between simplification, line generalization, and cartographic generalization.

Simplification reduces an object's detail in isolation, not taking the object's natural properties or surrounding objects into account. For example, if a river is simplified, it may have an approximate shape of the original river, but lose some shapes that define it. For example:

- Low-water rivers in slender slopes have many small bends next to each other. A non-cartographic line simplification may remove all of them, thus losing an important river's characteristic feature: after such simplification, it will be hard to tell that the original river was low-water in a slender slope.
- Low-angle river bend river over a long distance differs significantly from a completely straight canal. Non-cartographic line simplification may replace that bend with a straight line, making the river more similar to a canal than a river.

In other words, simplification processes the line, ignoring its geographic features. It works well when the features are human-made (e.g., roads, administrative boundaries, buildings). There is a number of freely available non-cartographic line simplification algorithms, which this paper will review.

Contrary to line simplification, cartographic generalization does not focus into a single feature class (e.g., rivers), but the whole map. For example, line simplification may change river bends in a way that bridges (and roads to the bridges) become misplaced. While line simplification is limited to a single feature class, cartographic generalization is not. Fully automatic cartographic generalization is not yet a solved problem.

Cartographic line generalization falls in between the two: it does more than line simplification, and less than cartographic generalization. Cartographic line generalization deals with a single feature class, takes into account its geographic properties, but ignores other features. This paper examines Wang–Müller's "Line generalization based on analysis of shape characteristics"[4], a cartographic line generalization algorithm.

2.2.2 Availability of Generalization Algorithms

Lack of robust openly available generalization algorithm implementations poses a problem for map creation with free software: there is no high-quality simplification algorithm to create down-scaled maps, so any cartographic work, which uses line generalization as part of its processing, will be of sub-par quality. We believe that the availability of high-quality open-source tools is an important foundation for future cartographic experimentation and development, thus it benefits the cartographic society as a whole.

Wang–Müller's commercial availability signals something about the value of the algorithm: at least the authors of the commercial software suite deemed it worthwhile to include it. However, not everyone has access to the commercial software suite, access to funds to buy the commercial suite, or access to the operating system required to run the commercial suite. PostGIS, in contrast, is free itself, and runs on free platforms. Therefore, algorithm implementations that run on PostGIS or other free platforms are useful to a wider cartographic society than proprietary ones.

2.2.3 Unfitness of Line Simplification Algorithms

Section 2.1.1 illustrates the current gaps with line simplification algorithms for real rivers. To sum up, we highlight the following cartographic problems from our examples:

Long bends should remain as long bends, instead of becoming fully straight lines.

Many small bends should not be removed. To retain a river's character, the algorithm should retain some small bends, and, when they are too small to be visible, they should be combined or exaggerated.

We are limiting the problem to cartographic line generalization. That is, full cartographic generalization, which takes topology and other feature classes into account, is out of scope.

Figure 7 on page 9 illustrates Wang–Müller algorithm from their original paper. Note how the long bends retain curvy, and how some small bends get exaggerated.

3 Methodology

The original Wang–Müller's algorithm [4] leaves something to be desired for a practical implementation: it is not straightforward to implement the algorithm from the paper alone.

Explanations in this document are meant to expand, rather than substitute, the original description in Wang–Müller. Therefore, familiarity with the original paper is assumed, and, for some sections, having the original close-by is necessary to meaningfully follow this document.

This paper describes Wang–Müller in detail that is more useful for anyone who wishes to follow the algorithm implementation more closely: each section is expanded with additional commentary, and illustrations added for non-obvious steps. Corner cases are discussed, too.

3.1 Main Geometry Elements Used by Algorithm

This section defines and explains the geometry elements that are used throughout this paper and the implementation. Assume Euclidean geometry throughout this document, unless noted otherwise.

VERTEX is a point on a plane, can be expressed by a pair of (x, y) coordinates.

LINE SEGMENT or SEGMENT joins two vertices by a straight line. A segment can be expressed by two coordinate pairs: (x_1, y_1) and (x_2, y_2). Line segment and segment are used interchangeably.

LINE or LINESTRING represents a single linear feature. For example, a river or a coastline.

Geometrically, a line is a series of connected line segments, or, equivalently, a series of connected vertices. Each vertex connects to two other vertices, with the exception of the vertices at either ends of the line: these two connect to a single other vertex.

MULTILINE or MULTILINESTRING is a collection of linear features. Throughout this implementation, this is used rarely (normally, a river is a single line) but can be valid when, for example, a river has an island.

BEND is a subset of a line that humans perceive as a curve. The geometric definition is complex and is discussed in section 4.4.

BASELINE is a line between the bend's first and last vertices.

SUM OF INNER ANGLES is a measure of how "curved" the bend is. Assume that first and last bend vertices are vectors. Then sum of inner angles will be the angular difference of those two vectors.

ALGORITHMIC COMPLEXITY measured in BIG O NOTATION, is a relative measure that helps explain how long[1] the algorithm will run depending on its input. It is widely used in computing science when discussing the efficiency of a given algorithm.

For example, given n objects and time complexity of $O(log(n))$, the time it takes to execute the algorithm is logarithmic to n. Conversely, if complexity is $O(n^2)$, then the time it takes to execute the algorithm grows quadratically with input. Importantly, if the input size doubles, the time it takes to run the algorithm quadruples.

BIG O NOTATION was first suggested by Bachmann[10] and Landau[11] in late XIX century, and clarified and popularized for computing science by Donald Knuth[12] in the 1970s.

[1]the upper bound, i.e., the worst case.

3.2 Algorithm Implementation Process

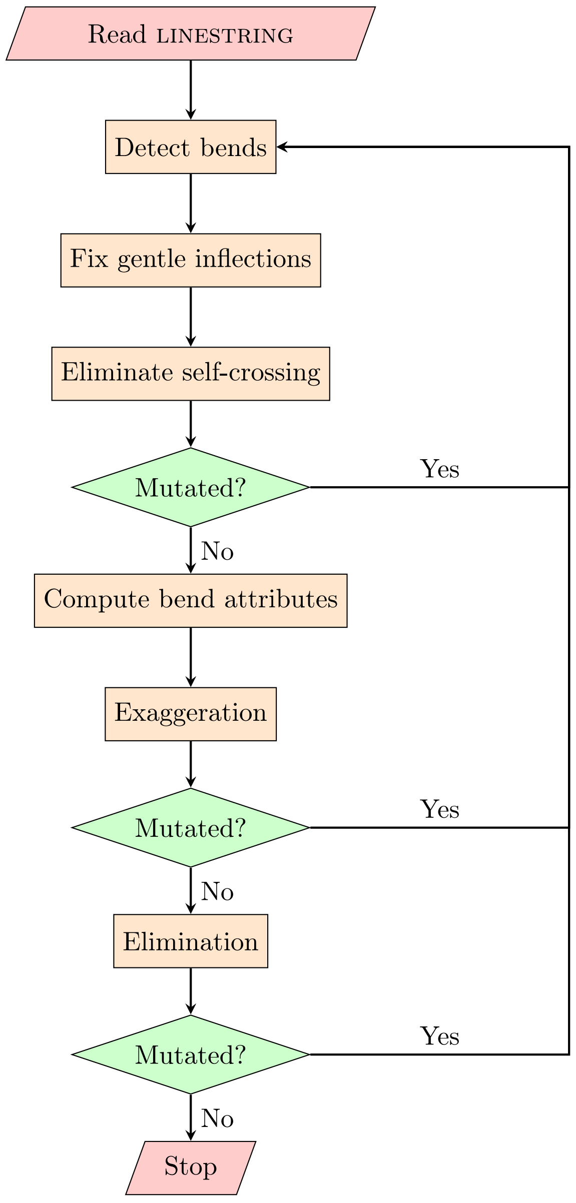


Figure 8: Flow chart of the implementation workflow.

Figure 8 visualizes the algorithm steps for each line. MULTILINESTRING features are split to LINESTRING features and executed in order.

Judging from Wang–Müller prototype flow chart (depicted in figure 11 of the original paper), their approach is iterative for the line: it will process the line in sequence, doing all steps, before moving on to the next step. We will call this approach “streaming”, because it does not require to have the full line to process it.

```
create function ST_SimplifyWM(
  geom geometry,
  dhalfcircle float,
  intersect_patience integer default 10,
  dbgname text default null
) returns geometry
```

Listing 1: Function ST_SIMPLIFYWM.

We have taken a different approach: process each step fully for the line, before moving to the next step. This way provides the following advantages:

- For ELIMINATE SELF-CROSSING stage, when it finds a bend with the right sum of inflection angles, it checks the whole line for self-crossings. This is impossible with streaming because it requires having the full line in memory. It could be optimized by, for example, looking for a fixed number of neighboring bends (say, 10), but that would complicate the implementation.

- FIX GENTLE INFLECTIONS is iterating the same line twice from opposite directions. That could be re-written to streaming fashion, but it complicates the implementation, too.

On the other hand, comparing to the Wang–Müller prototype flow chart, our implementation uses more memory (because it needs to have the full line before processing), and some steps are unnecessarily repeated, like re-computing the bend's attributes during repeated iterations.

3.3 Technical Implementation

Technical algorithm realization was created in *PostGIS 3.1.1*[13]. PostGIS is a PostgreSQL extension for working with spatial data.

PostgreSQL is an open-source relational database, widely used in industry and academia. PostgreSQL can be interfaced from nearly any programming language; therefore, solutions written in PostgreSQL (and their extensions) are usable in many environments. On top of that, PostGIS implements a rich set of functions[14] for working with geometric and geographic objects.

Due to its wide applicability and rich library of spatial functions, PostGIS is the implementation language of the Wang–Müller algorithm. The implementation exposes the entrypoint function ST_SIMPLIFYWM, in listing 1.

This function accepts the following parameters:

GEOM is the input geometry. Either LINESTRING or MULTILINESTRING.

DHALFCIRCLE is the diameter of the half-circle. Explained in section 4.3.

INTERSECT_PATIENCE is an optional parameter to exaggeration operator, explained in section 4.13.

DBGNAME is an optional human-readable name of the figure. Explained in section 4.1.

The function ST_SIMPLIFYWM calls into helper functions, which detect, transform, or remove bends. These helper functions are also defined in the implementation and are part of the algorithm technical realization. All supporting functions use spatial manipulation functions provided by PostGIS.

3.4 Automated Tests

As part of the algorithm realization, an automated test suite has been developed. Shapes to test each function have been hand-crafted, and expected results have been manually calculated. The test suite executes parts of the algorithm against a predefined set of geometries, and asserts that the output matches the resulting hand-calculated geometries.

The full set of test geometries is visualized in Figure 9.

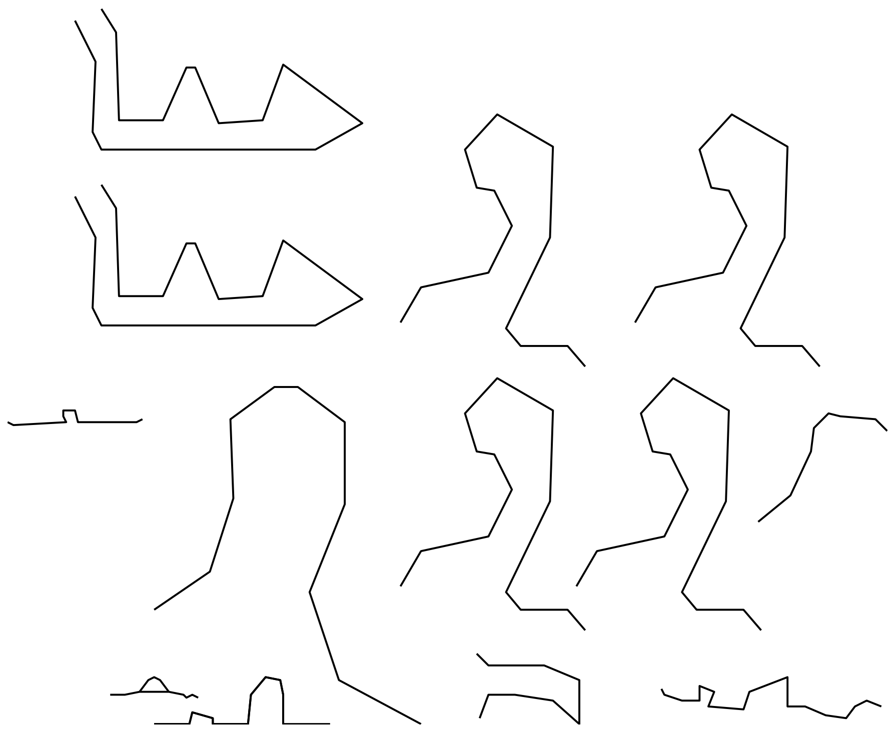


Figure 9: Geometries for automated test cases.

Test suite can be executed with a single command and completes in about a second. Having an easily accessible test suite boosts confidence that no unexpected bugs have snug in while modifying the algorithm.

We will explain two instances when automated tests were very useful during the implementation:

- Created a function WM_EXAGGERATION, which exaggerates bends following the rules. It worked well over simple geometries but, due to a subtle bug, created a self-crossing bend in Visinčia. The offending bend was copied to the automated test suite, which helped fix the bug. Now the test suite contains the same bend (a hook-like bend on the right-hand side of Figure 9) and code to verify that it was correctly exaggerated.

- During algorithm development, automated tests run about once a minute. They quickly find logical and syntax errors. In contrast, running the algorithm with real rivers takes a few minutes, which increases the feedback loop, and takes longer to fix the "simple" errors.

Whenever we find and fix a bug, we aim to create an automated test case for it, so the same bug is not re-introduced by whoever works next on the same piece of code.

Besides testing for specific cases, an automated test suite ensures future stability and longevity of the implementation itself: when new contributors start changing code, they have higher assurance they have not broken an already-working code.

3.5 Reproducibility

It is widely believed that the ability to reproduce the results of a published study is important to the scientific community. In practice, however, it is often hard or impossible: research methodologies, as well as algorithms themselves, are explained in prose, which, due to the nature of the non-machine language, lends itself to inexact interpretations.

This article, besides explaining the algorithm in prose, includes the program of the algorithm in a way that can be executed on reader's workstation. On top of it, all the illustrations in this paper are generated using that algorithm from a predefined list of test geometries (see section 3.4).

This article and accompanying code are accessible on GitHub as of 2021-05-19 [15].

Instructions how to re-generate all the visualizations are in appendix A.1. The visualization code serves as a good example reference for anyone willing to start using the algorithm.

4 Algorithm Implementation

As alluded in section 1, Wang–Müller paper skims over certain details which are important to implement the algorithm. This section goes through each algorithm stage, illustrating the intermediate steps and explaining the author's desiderata for a more detailed description.

Illustrations of the following sections are extracted from the automated test cases which were written during the algorithm implementation (as discussed in section 3.4).

4.1 Debugging

This implementation includes debugging facilities in a form of a table WM_DEBUG. The table's schema is written in listing 2.

When debug mode is active, implementation steps will store their results, which can be useful to manually inspect the results of intermediate actions. Besides manual inspection, most of the figure illustrations in this article are visualized from the WM_DEBUG table. Debugging mode can be activated by passing a non-empty DBGNAME string to the function ST_SIMPLIFYWM (this function was described in section 3.3). By convention, DBGNAME is the name of the geometry that is being simplified, e.g., ŠALČIA. The purpose of each column in WM_DEBUG is described below:

ID is a unique identifier for each feature. Generated automatically by PostgreSQL. Useful when it is necessary to copy one or more features to a separate table for unit tests, as described in section 3.4.

STAGE is the stage of the algorithm. As of writing, there are a few:

AFIGURES at the beginning of the loop.

BBENDS after bends are detected.

CINFLECTIONS after gentle inflections are fixed.

DCROSSINGS after self-crossings are eliminated.

EBENDATTRS after bend attributes are calculated.

GEXAGGERATION after bends have been exaggerated.

HELIMINATION after bends have been eliminated.

Some of these have sub-stages which are encoded by a dash and a sub-stage name, e.g., BBENDS-POLYGON creates polygon geometries after polygons have been detected; this particular example is used to generate colored polygons in Figure 11.

NAME is the name of the geometry, which comes from parameter DBGNAME.

GEN is the top-level iteration number. In other words, the number of times the execution flow passes through DETECT BENDS phase as depicted in Figure 8 on page 13.

NBEND is the bend's index in its LINE.

WAY is the geometry column.

PROPS is a free-form JSON object to store miscellaneous values. For example, EBENDATTRS phase stores a boolean property ISOLATED, which signifies whether the bend is isolated or not (explained in section 4.9).

When debug mode is turned off (that is, DBGNAME is left unspecified), WM_DEBUG is empty and the algorithm runs slightly faster.

4.2 Merging Pieces of a River into One

Example river geometries were sourced from OpenStreetMap[16] and NŽT[17]. Rivers in both data sources are stored in shorter line segments, and multiple segments (usually hundreds or thousands for significant rivers) define one full river. While it is convenient to store and edit, these segments are not explicitly related to each other. This poses a problem for simplification algorithms which manipulate on full linear features at a time: full river geometries, but not their parts.

```
drop table if exists wm_debug;
create table wm_debug(
  id serial,
  stage text not null,
  name text not null,
  gen bigint not null,
  nbend bigint,
  way geometry,
  props jsonb
);
```

Listing 2: WM_DEBUG table definition

Since these rivers do not have an explicit relationship to connect them together, they were connected using heuristics: if two line segments share a name and are within 500 meters from each other, then they form a single river. For all line simplification algorithms, all rivers need to be combined and this way proved to be reasonably effective. Source code for this operation can be found in listing 5 on page 54.

4.3 Bend Scaling And Dimensions

Wang–Müller accepts a single input parameter: the diameter of a half-circle. If the bend's adjusted size (explained in detail in section 4.8) is greater than the area of the half-circle, then the bend will be left untouched. If the bend's adjusted size is smaller than the area of the provided half-circle, the bend will be simplified: either exaggerated, combined, or eliminated.

The extent of line simplification, as well as the half-circle's diameter, depends on the desired target scale. Simplification should be more aggressive for smaller target scales and less aggressive for larger scales. This section goes through the process of finding the correct variable to Wang–Müller algorithm. What is the minimal, but still eligible, figure that should be displayed on the map?

According to *Cartographic Design for Screen Maps*[18], the map is typically held at a distance of 30 cm. Recommended minimum symbol size, given viewing distance of 45 cm (1.5 feet), is 1.5 mm, as analyzed in *Guidelines for minimum size for text and symbols on maps*[19].

In our case, our target is line bend, rather than a symbol. Assume 1.5 mm is a diameter of the bend. A semi-circle of 1.5 mm diameter is depicted in Figure 10. A bend of this size or larger, when adjusted to scale, will not be simplified.

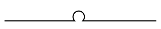


Figure 10: Smallest feature that will be not simplified (to scale).

Wang–Müller algorithm does not have a notion of scale, but it does have a notion of distance: it accepts a single parameter D, the half-circle's diameter. Assuming measurement units in projected coordinate system are meters (for example, *WGS 84/Pseudo-Mercator*[20]), some popular scales are highlighted in table 1.

Scale	$D(m)$
1:10 000	15
1:15 000	22.5
1:25 000	37.5
1:50 000	75
1:250 000	220

Table 1: Wang–Müller half-circle diameter D for popular scales.

4.4 Definition of a Bend

The original article describes a bend as follows:

> A bend can be defined as that part of a line which contains a number of subsequent vertices, with the inflection angles on all vertices included in the bend being either positive or negative and the inflection of the bend's two end vertices being in opposite signs. [4]

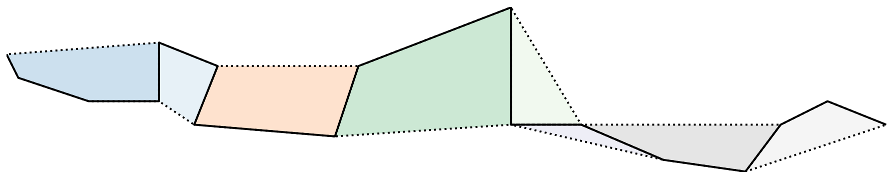


Figure 11: Similar to figure 8 in [4]: detected bends are highlighted.

Here are some non-obvious characteristics that are necessary when writing code to detect the bends:

- End segments of each line should also belong to bends. That way, all segments belong to 1 or 2 bends.
- First and last segments of each bend (except for the two end-line segments) are also the first vertex of the next bend.

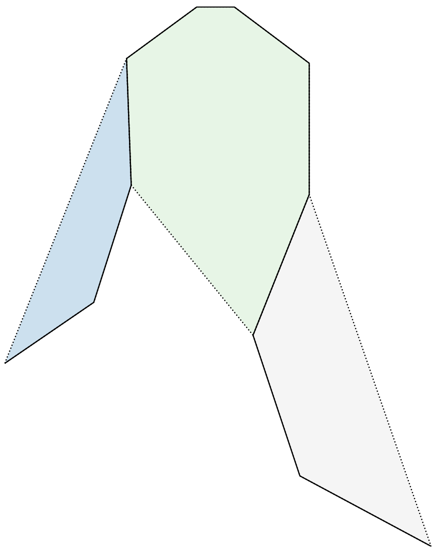
4.5 Gentle Inflection at the End of a Bend

> But if the inflection that marks the end of a bend is quite small, people would not recognize this as the bend point of a bend [4]

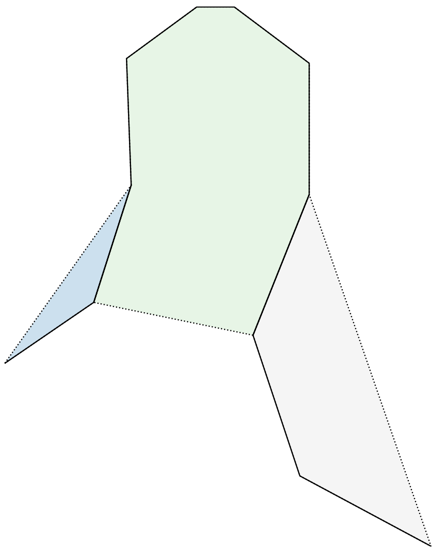
Figure 12 visualizes the original paper's figure 5, when a single vertex is moved outwards the end of the bend.

The illustration for this section was clear but insufficient: it does not specify how many vertices should be included when calculating the end-of-bend inflection. The iterative approach was chosen: as long as the angle is "right" and the baseline is becoming shorter, the algorithm should keep re-assigning vertices to different bends. There is no upper bound on the number of iterations.

To prove that the algorithm implementation is correct for multiple vertices, additional example was created and illustrated in Figure 13: the rule re-assigns two vertices to the next bend.

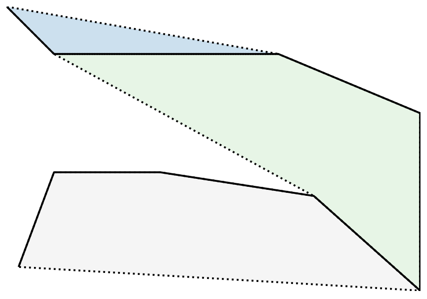


(a) Before applying the inflection rule.

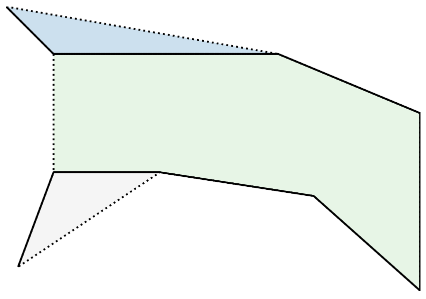


(b) After applying the inflection rule.

Figure 12: Figure 5 in [4]: gentle inflections at the ends of the bend.



(a) Before applying the inflection rule.



(b) After applying the inflection rule.

Figure 13: Gentle inflection at the end of the bend with multiple vertices.

Note that to find and fix the gentle bends' inflections, the algorithm should run twice, both ways. Otherwise, if it is executed only one way, the steps will fail to match some bends that should be adjusted. Current implementation works as follows:

1. Run the algorithm from the beginning to the end.
2. Reverse the line and each bend.
3. Run the algorithm again.
4. Reverse the line and each bend.
5. Return result.

Reversing the line and its bends is straightforward to implement but costly: the two reversal steps cost additional time and memory. The algorithm could be made more optimal with a similar version of the algorithm, but the one which goes backwards. In this case, steps 2 and 4 could be spared, that way saving memory and computation time.

The "quite small angle" was arbitrarily chosen to 45°.

4.6 Self-Line Crossing When Cutting a Bend

When a bend's baseline crosses another bend, it is called self-crossing. Self-crossing is undesirable for the upcoming bend manipulation operators; therefore, should be removed. There are a few rules on when and how they should be removed — this section explains them in higher detail, discusses their time complexity and applied optimizations. Figure 14 is copied from the original article.

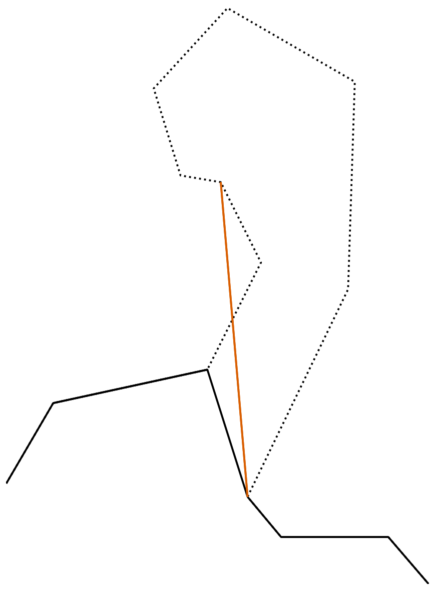


Figure 14: Originally figure 6: the bend's baseline (orange) is crossing a neighboring bend.

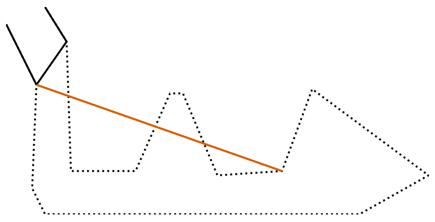


Figure 15: The bend's baseline (orange) is crossing a non-neighboring bend.

Looking at the Wang–Müller paper alone, it may seem like self-crossing may happen only with the neighboring bend. This would mean an efficient $O(n)$ implementation[2]. However, as one can see in Figure 15, it may not be the case: any other bend in the line may be crossing it.

If one translates the requirements to code in a straightforward way, it would be quite computationally expensive: naively implemented, complexity of checking every bend with every bend is $O(n^2)$. In other words, the time it takes to run the algorithm grows quadratically with the number of vertices.

It is possible to optimize this step and skip checking a large number of bends. Only bends, the inner angles' sum of which is larger than 180°, can ever self-cross. That way, only a fraction of bends need to be checked. The worst-case complexity is still $O(n^2)$, when all bends' inner angles are larger than 180°. Having this optimization, the algorithmic complexity (as a result, the time it takes to execute the algorithm) drops by the fraction of bends, the inner angles' sum of which is smaller than 180°.

4.7 Attributes of a Single Bend

COMPACTNESS INDEX is "the ratio of the area of the polygon over the circle whose circumference length is the same as the length of the circumference of the polygon" [4]. Given a bend, its compactness index is calculated as follows:

1. Construct a polygon by joining first and last vertices of the bend.
2. Calculate the area of the polygon A_p.
3. Calculate perimeter P of the polygon. The same value is the circumference of the circle: $C = P$.
4. Given the circle's circumference C, the circle's area A_c is:

$$A_c = \frac{C^2}{4\pi}$$

5. Compactness index c is the area of the polygon A_p divided by the area of the circle A_c:

$$c = \frac{A_p}{A_c} = \frac{A_p}{\frac{C^2}{4\pi}} = \frac{4\pi A_p}{C^2}$$

Once this operation is complete, each bend will have a list of properties which will be used by other modifying operators.

4.8 Shape of a Bend

This section introduces ADJUSTED SIZE A_{adj} which trivially derives from COMPACTNESS INDEX c and "polygonized" bend's area A_p:

$$A_{adj} = \frac{0.75A_p}{c}$$

[2]where n is the number of bends in a line. See explanation of ALGORITHMIC COMPLEXITY in section 3.1.

Adjusted size is necessary later to compare bends with each other, or to decide if the bend is within the simplification threshold.

Sometimes, when working with Wang–Müller, it is useful to convert between half-circle's diameter D and adjusted size A_{adj}. These easily derive from circle's area formula $A = 2\pi\frac{D}{2}^2$:

$$D = 2\sqrt{\frac{2A_{adj}}{\pi}}$$

In reverse, adjusted size A_{adj} from half-circle's diameter:

$$A_{adj} = \frac{\pi D^2}{8}$$

4.9 Isolated Bend

Bend itself and its "isolation" can be described by AVERAGE CURVATURE, which is "geometrically defined as the ratio of inflection over the length of a curve." [4]

Two conditions must be followed to claim that a bend is isolated:

1. AVERAGE CURVATURE of neighboring bends should be larger than the "candidate" bend's curvature. The article did not offer a value; this implementation arbitrarily chose 0.5.

2. Bends on both sides of the "candidate" bend should be longer than a certain value. This implementation does not (yet) define such a constraint and will only follow the average curvature constraint above.

We believe unclear criteria for ISOLATED BEND is one of the main causes for jagged lines in section 5, and is a suggested further area of research in section 7.

4.10 The Context of a Bend: Isolated And Similar Bends

To find out whether two bends are similar, they are compared by 3 components:

1. ADJUSTED SIZE A_{adj}.

2. COMPACTNESS INDEX c.

3. BASELINE LENGTH l.

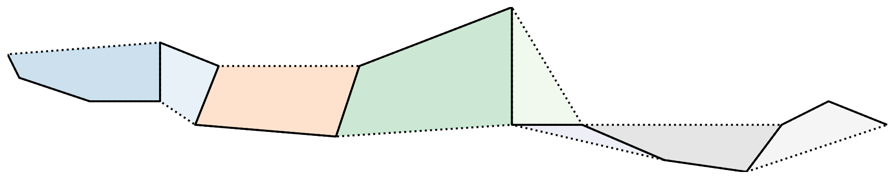
Components 1, 2 and 3 represent a point in a 3-dimensional space, and Euclidean distance $d(p, q)$ between those is calculated to differentiate bends p and q:

$$d(p, q) = \sqrt{(A_{adj(p)} - A_{adj(q)})^2 + (c_p - c_q)^2 + (l_p - l_q)^2}$$

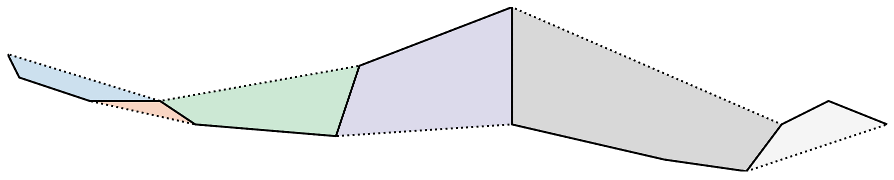
The smaller the distance d, the more similar the bends are.

4.11 Elimination Operator

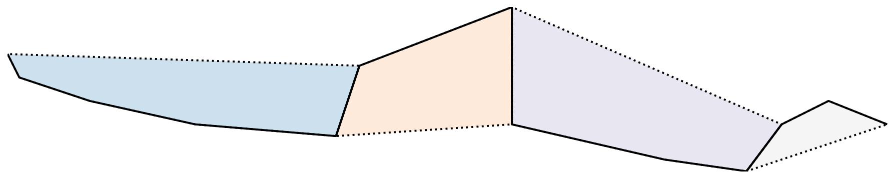
Figure 16 illustrates steps of figure 8 from the original paper. There is not much to add to the original description beyond repeating the elimination steps in an illustrated example.



(a) Original



(b) Iteration 1



(c) Iteration 2 (result)

Figure 16: Originally figure 8: the bend elimination through iterations.

4.12 Combination Operator

Combination operator was not implemented in this version.

4.13 Exaggeration Operator

Exaggeration operator finds bends, of which ADJUSTED SIZE is smaller than the DIAMETER OF THE HALF-CIRCLE. Once a target bend is found, it will be exaggerated in increments until either becomes true:

- ADJUSTED SIZE of the exaggerated bend is larger than the area of the half-circle.
- The exaggerated bend starts intersecting with a neighboring bend. Then exaggeration aborts, and the bend remains as if it were one step before the intersection.

Exaggeration operator uses a hardcoded parameter EXAGGERATION STEP $s \in (1, 2]$. It was arbitrarily picked to 1.2 for this implementation. A single exaggeration increment is done as follows:

1. Find a candidate bend.
2. Find the bend's baseline.
3. Find MIDPOINT, the center of the bend's baseline.
4. Find MIDBEND, the center of the bend. Distance from one baseline vertex to MIDBEND should be the same as from MIDBEND to the other baseline vertex.
5. Mark each bend's vertex with a number between $[1, s]$. The number is derived with elements linearly between the start vertex and MIDBEND, with values somewhat proportional to the azimuth between these lines:
   - MIDBEND and the point.
   - MIDPOINT and the point.

   The other half of the bend, from MIDBEND to the final vertex, is linearly interpolated between $[s, 1]$, using the same rules as for the first half.

   The first version of the algorithm used simple linear interpolation based on the point's position in the line. The current version applies a few coefficients, which were derived empirically, by observing the resulting bend.
6. Each point (except the beginning and end vertices of the bend) will be placed farther away from the baseline. The length of misplacement is the marked value in the previous step.

Figure 17: Example isolated exaggerated bend.

The technical implementation of the algorithm contains two implementations of exaggeration operator: WM_EXAGGERATE_BEND is the original one. It uses simple linear interpolation. It is fast, but simple. It tends to leave jagged bends. WM_EXAGGERATE_BEND2 is a more computationally expensive function, which leaves better-looking exaggerated bends.

Both functions are interchangeable and can be found in listing 4. Figure 17 illustrates an exaggerated bend using WM_EXAGGERATE_BEND2.

5 Results

This section visualizes the results, discusses robustness and issues of the generalization, and suggests specific improvements.

One of our goals is to compare the generalized lines with the official generalized dataset[17]. Therefore, we have selected the target scales that the official sources offer, too: 1:50 000 and 1:250 000. The DHALFCIRCLE values for the subset are as follow:

Scale	$D(m)$
1:50 000	75
1:250 000	220

Our generalized results are viewed from the following angles:

- Compare to the non-simplified originals.
- Compare to the official datasets.
- Compare to Douglas & Peucker and Visvalingam–Whyatt.

5.1 Generalization Results of Analyzed Rivers

5.1.1 Medium-Scale (1:50 000)

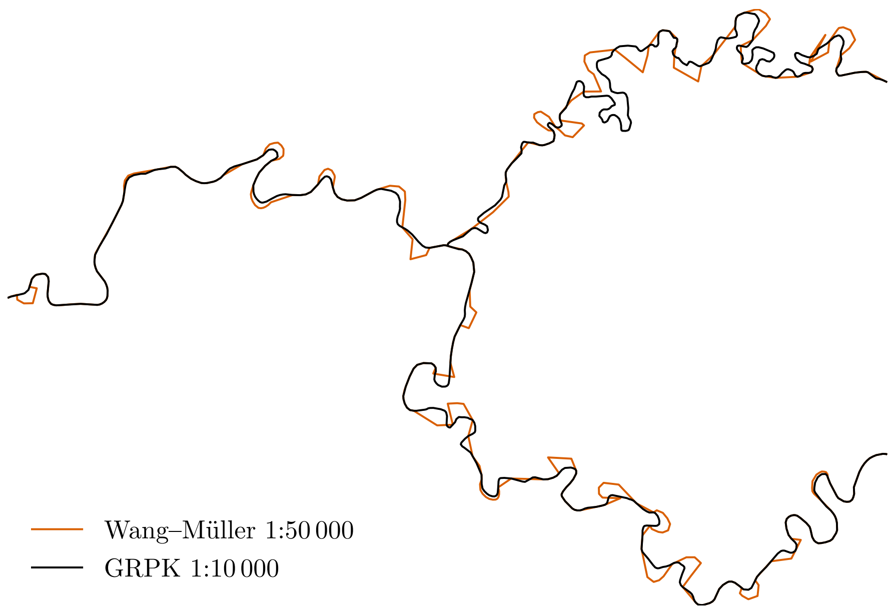


Figure 18: 2x zoomed-in Wang–Müller for 1:50 000.

As one can see in Figure 18, the illustrations deliver what was promised by the algorithm, but with a few caveats. Left side of the figure looks reasonably

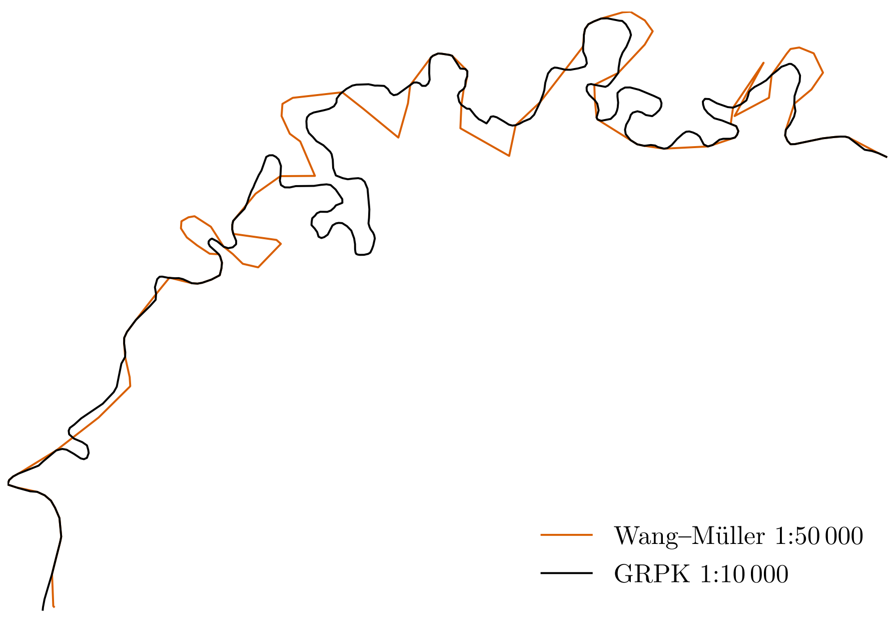


Figure 19: Top–right part of Figure 18.

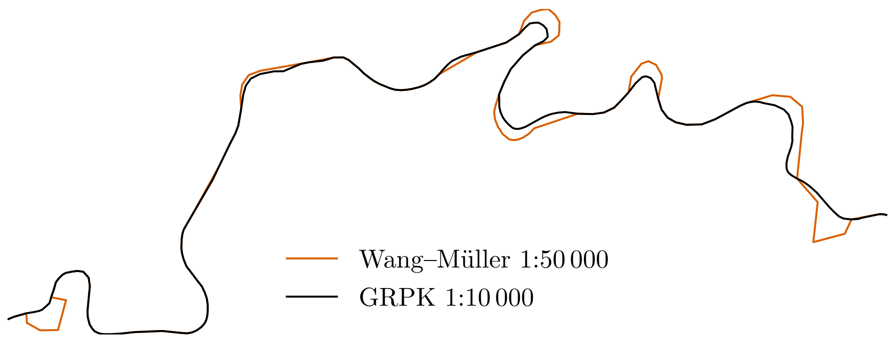


Figure 20: Left part of Figure 18.

well simplified: long bends remain slightly curved, small bends are removed or slightly exaggerated.

Figure's 18 left part is clipped to Figure 20. As one can see, some bends were well exaggerated, and some bends were eliminated.

Top–right side (clipped in Figure 20), some jagged and sharp bends appear. These will become more pronounced in even larger-scale simplification in the next section.

To sum up, mid-scale simplification works well for some geometries, but creates sharp edges for others.

5.1.2 Large-Scale (1:250 000)

As visible in Figure 21, for large-scale map, some of the resulting bends look significantly exaggerated. Why is that? Figure 22 zooms in the large-scale simplification and overlays the original.

(a) Original.

(b) Simplified.

Figure 21: GRPK10 simplified with Wang–Müller for 1:250 000.

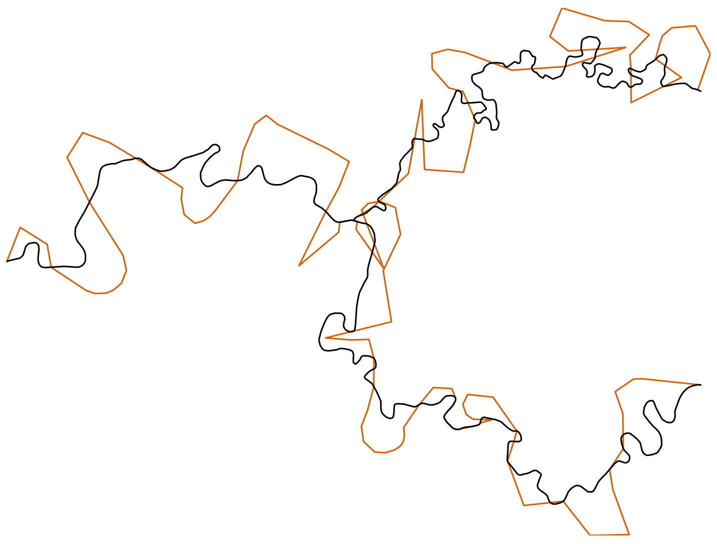


Figure 22: 10x zoomed-in Wang–Müller for 1:250 000.

A conglomeration of bends is visible, especially in top–right side of the illustration. We assume this was caused by two bends significantly exaggerated, leaving no space to exaggerate those between the two.

5.1.3 Discussion

For mid-size scales of 1:50 000, the implemented algorithm works well for certain geometries, and poorly for others. This test surfaced two areas for future research and improvement:

- Exaggeration is sometimes creating sharp edges, especially when the exaggerated bend is quite small. When sharp edges are created, exaggeration could interpolate more points in the bend, and exaggerate using the interpolated points.
- In larger scales, when bends do not have space to exaggerate, they should be combined or eliminated instead.

5.2 Comparison with National Spatial Datasets

There are a few datasets used in this comparison: GRPK10, GRPK50 and GRPK250. They are vector datasets which include rivers. They can be downloaded for free from [17]. Here are the meanings of the codenames:

GRPK10 is a dataset of highest detail. Suited for maps of scale 1:10 000.

GRPK50 is suited for maps of scale 1:50 000.

GRPK250 offers the least detail, and is suited for maps of scale 1:250 000.

During the analysis, we ran Wang–Müller on GRPK10 for 2 destination scales: 1:50 000 and 1:250 000.[3] This section compares the resulting Wang–Müller–generalized rivers to GRPK50 and GRPK250.

5.2.1 Medium-Scale (1:50 000)

For our research location, the national dataset GRPK10 is almost equivalent to GRPK50, with a few nuances. Figure 23 illustrates all three shapes: GRPK50, Wang–Müller–simplified GRPK10, and the original GRPK10.

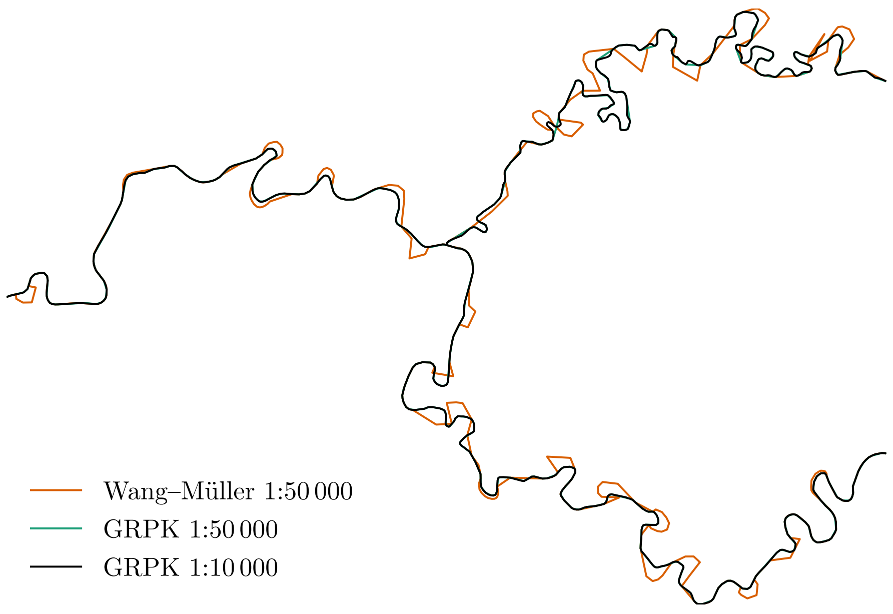


Figure 23: GRPK10, GRPK50 and Wang–Müller for 1:50 000.

Although figures are almost identical, Figure 25 illustrates two small bends that have been removed in GRPK50, but have been exaggerated by our implementation.

[3]parameter calculation is detailed in section 4.3.

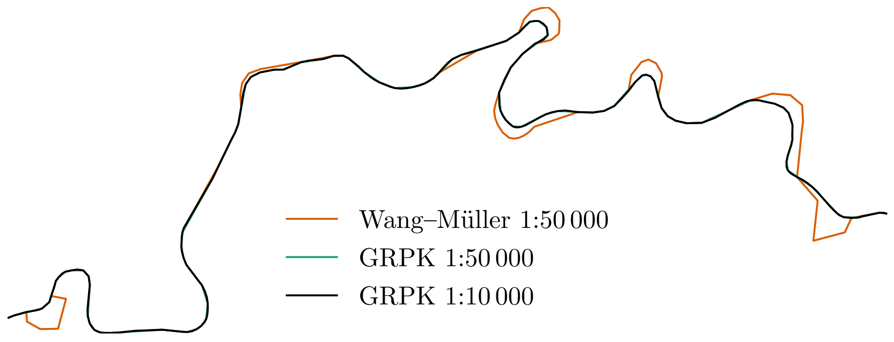


Figure 24: Left side of Figure 23.

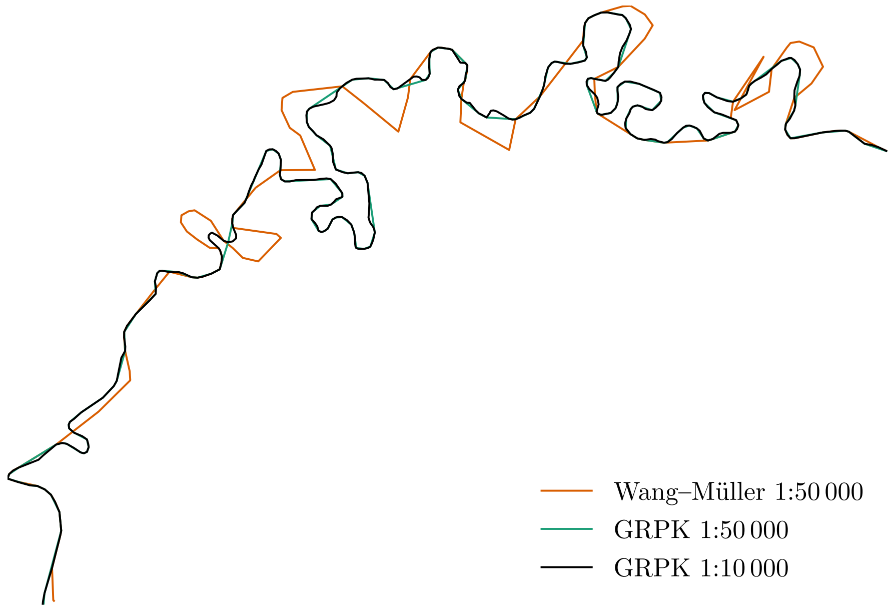
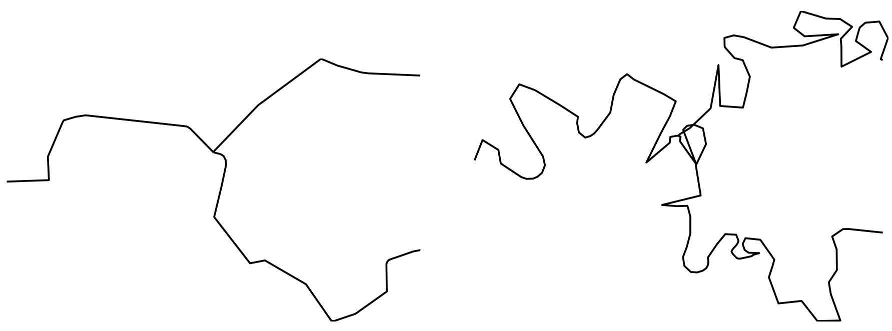


Figure 25: Top–right side of Figure 23.

5.2.2 Large-Scale (1:250 000)

Figure 26 illustrates the original GRPK250 and the Wang–Müller–simplified version. As section 5.1.2 explains, the algorithm tries to exaggerate many bends to a great size. However, GRPK250 takes the opposite approach — only the very basic shapes of the largest bends are retained. Time and customers will tell, which approach is more appropriate, after the current Wang–Müller implementation receives some time and attention, as desired in section 7.



(a) GRPK250. (b) Wang–Müller-simplified GRPK10.

Figure 26: GRPK250 and Wang–Müller–simplified GRPK10.

5.3 Comparison with Douglas & Peucker and Visvalingam–Whyatt

It is time to visually compare our implementation with the classical algorithms: Douglas & Peucker, Visvalingam–Whyatt and Chaikin. Since we have established that more work is needed for small-scale maps (1:250 000), we will limit the comparison in this section to 1:50 000.

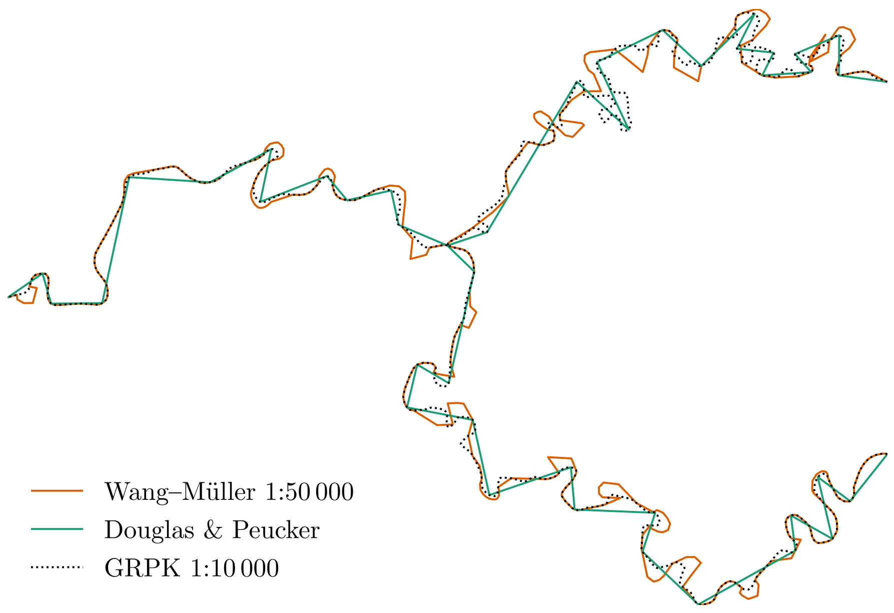


Figure 27

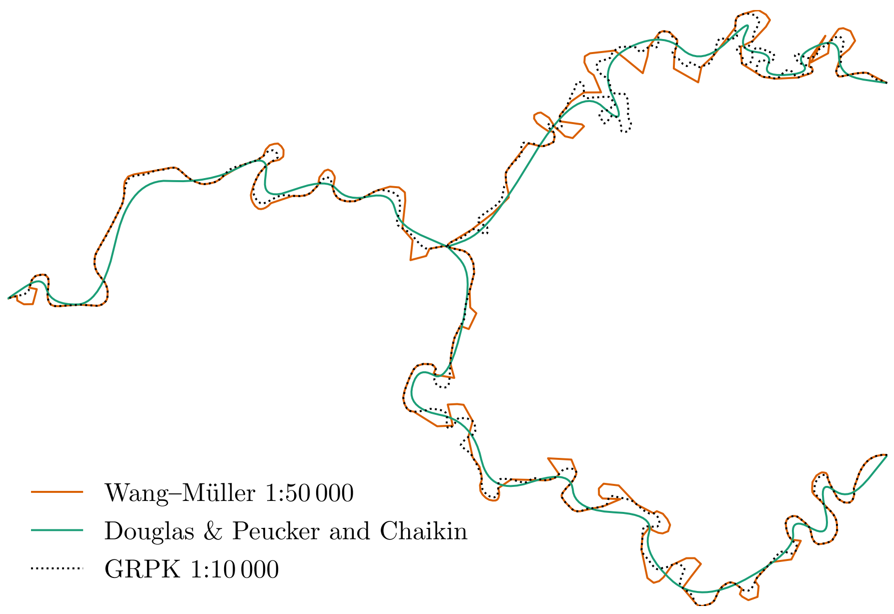


Figure 28

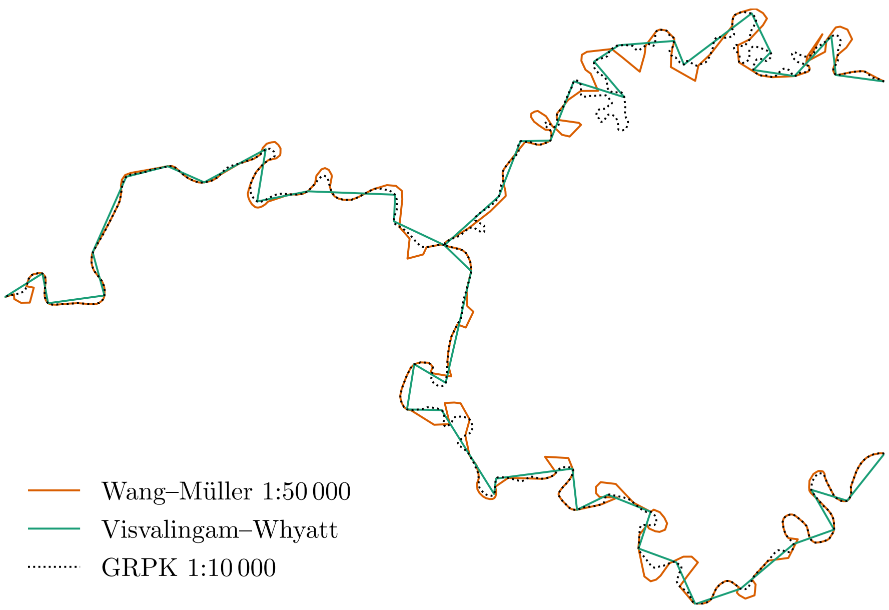


Figure 29

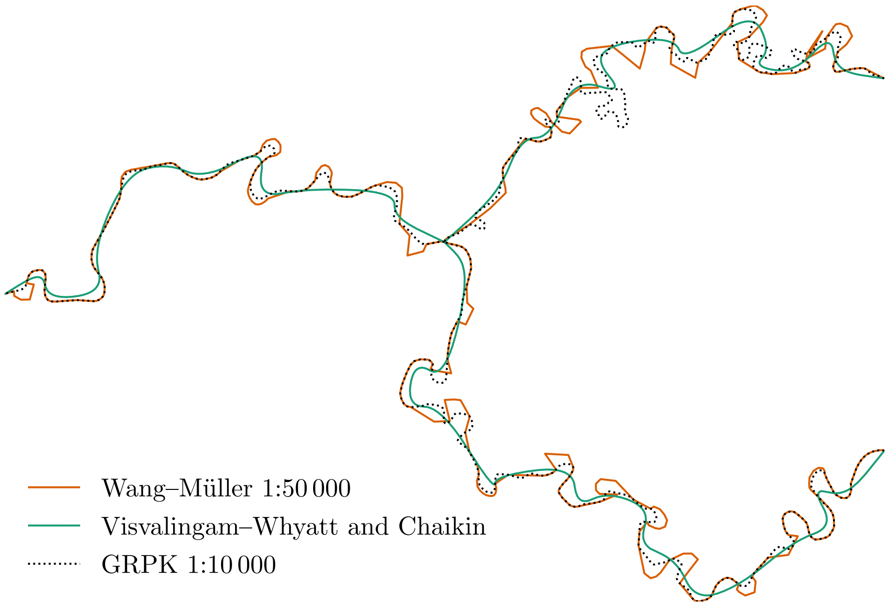


Figure 30

5.4 Testing Results Online

An on-line tool[21] has been developed to test incoming parameters to Wang–Müller algorithm. A user should select a river of interest, enter the DHALFCIRCLE parameter and click "Submit". The simplified line feature will be overlaid on top of the map.

Figure 31 illustrates the end result that looks reasonably well. Figure 32 illustrates that the algorithm produces poorly simplified results for some geometries.

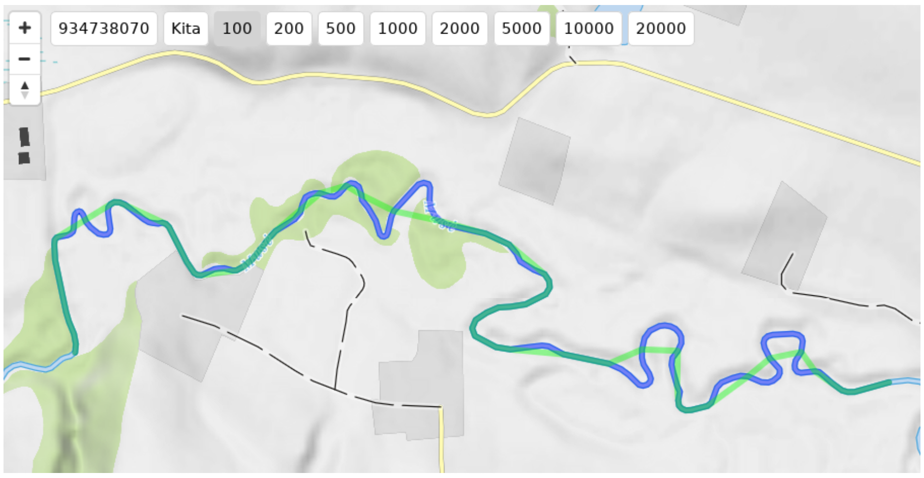


Figure 31: Example on-line test tool for Wang–Müller algorithm.

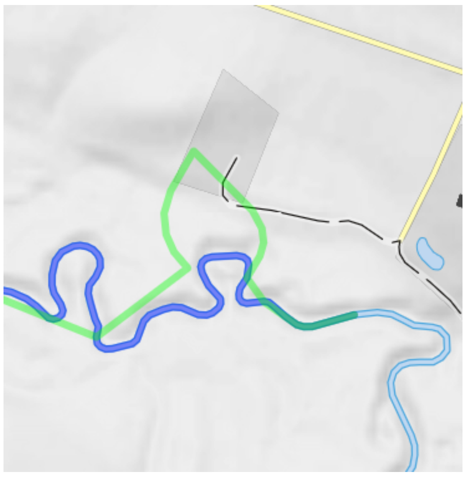


Figure 32: Another example from the on-line test tool.

6 Conclusions

Classical and modern line simplification algorithms were evaluated, main problems with them identified. A method for Wang–Müller technical implementation was defined, and the algorithm implemented. Each geometric transformation was described and visualized. The implemented algorithm was applied for different shapes and compared to national (Lithuanian) datasets.

About 1,000 lines of Procedural SQL were written for the algorithm and tests, and a few hundred lines of supporting scripts in Make, Python, Awk, Bash. With the help of its permissive license and early interest, the algorithm code has already been used to create a prototype on-line service to evaluate the algorithm robustness.

7 Future Suggestions

These are the areas for possible future work with this, published, implementation:

- Implement bend combination operator (section 4.12).
- Fine-tune parameters for bend exaggeration. Section 5.1 contains a exaggerated bends that became sharp and includes some future ideas.
- What are the exaggeration limits when working with large scales? Section 5.2.2 discusses examples that some limits are necessary.
- Research when bends should be marked as ISOLATED. As is seen from examples, the current criteria are not robust enough.
- Once the points above yield a satisfactory result, the efficiency of the algorithm could be improved to work on the lines in "streaming" fashion (more details in section 3.2).

That sums up what could be improved without changing the algorithm in a significant way. Other than that, further area of research is working towards graduating the algorithm from "isolated cartographic generalization" to "full cartographic generalization". The current operators of Wang–Müller algorithm have a few venues to preserve the surrounding topology. This could be further researched and extended.

8 Acknowledgments

I would like to thank my thesis supervisor, Dr. Andrius Balčiūnas, for his help in formulating the requirements and providing early editorial feedback for the thesis.

I am grateful to Tomas Straupis, who handed me the Wang–Müller[4] paper on a warm pre-COVID summer evening. I got intrigued. He was also an early beta-tester of my implementation, and helped me understand where the initial algorithm descriptions were ambiguous.

Many thanks to NŽT[17] for providing the datasets with a very permissive license.

References

[1] David H Douglas and Thomas K Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature.” In: *Cartographica: the international journal for geographic information and geovisualization* 10.2 (1973), pp. 112–122.

[2] Maheswari Visvalingam and James D Whyatt. “Line generalisation by repeated elimination of points.” In: *The cartographic journal* 30.1 (1993), pp. 46–51.

[3] George Merrill Chaikin. “An algorithm for high-speed curve generation.” In: *Computer graphics and image processing* 3.4 (1974), pp. 346–349.

[4] Zeshen Wang and Jean-Claude Müller. “Line generalization based on analysis of shape characteristics.” In: *Cartography and Geographic Information Systems* 25.1 (1998), pp. 3–15.

[5] Bin Jiang and Byron Nakos. “Line simplification using self-organizing maps.” In: *Proceedings of the ISPRS Workshop on Spatial Analysis and Decision Making, Hong Kong, China.* 2003, pp. 3–5.

[6] Christopher Dyken, Morten Dæhlen, and Thomas Sevaldrud. “Simultaneous curve simplification.” In: *Journal of geographical systems* 11.3 (2009), pp. 273–289.

[7] Nabil Mustafa, Shankar Krishnan, Gokul Varadhan, and Suresh Venkatasubramanian. “Dynamic simplification and visualization of large maps.” In: *International Journal of Geographical Information Science* 20.3 (2006), pp. 273–302.

[8] Martin Nöllenburg, Damian Merrick, Alexander Wolff, and Marc Benkert. “Morphing polylines: A step towards continuous generalization.” In: *Computers, Environment and Urban Systems* 32.4 (2008), pp. 248–260.

[9] Türkay Gökgöz, Alper Sen, Abdulkadir Memduhoğlu, and Müslüm Hacar. “A New Algorithm for Cartographic Simplification of Streams and Lakes Using Deviation Angles and Error Bands.” In: *ISPRS International Journal of Geo-Information* 4 (2015-10), pp. 2185–2204. DOI: `10.3390/ijgi4042185`.

[10] Paul Bachmann. *Die analytische zahlentheorie.* Vol. 2. Teubner, 1894.

[11] “Handbuch der Lehre von der Verteilung der Primzahlen.” In: *Monatshefte für Mathematik und Physik* 22.1 (1911-12), A26–A26. ISSN: 1436-5081. DOI: `10.1007/BF01742852`.

[12] Donald E Knuth. “Big omicron and big omega and big theta.” In: *ACM Sigact News* 8.2 (1976), pp. 18–24.

[13] PostGIS Team. *PostGIS 3.1.1.* URL: `https://postgis.net/2021/01/28/postgis-3.1.1/` (visited on 2021-05-12).

[14] PostGIS Team. *PostGIS Reference.* URL: `https://postgis.net/docs/reference.html` (visited on 2021-05-12).

[15] Motiejus Jakštys. *Wang–Müller implementation in PostGIS.* URL: `https://github.com/motiejus/wm` (visited on 2021-05-19).

[16] OpenStreetMap contributors. *Project that creates and distributes free world’s geographic data.* URL: `https://www.openstreetmap.org` (visited on 2021-05-15).

[17] Nacionalinė Žemės Tarnyba Prie Žemės Ūkio Ministerijos. *Atviri Duomenys.* URL: http://nzt.lt/go.php/lit/Atviri-duomenys (visited on 2021-05-15).

[18] CartouCHe. *Cartographic Design for Screen Maps. Minimum Dimensions.* 2012-01-26. URL: http://www.e-cartouche.ch/content_reg/cartouche/cartdesign/en/html/GenRules_learningObject3.html (visited on 2021-05-03).

[19] Aileen Buckley. *Guidelines for minimum size for text and symbols on maps.* Esri. 2008-01-16. URL: https://www.esri.com/arcgis-blog/products/product/mapping/guidelines-for-minimum-size-for-text-and-symbols-on-maps/ (visited on 2021-05-03).

[20] MapTiler Team. *WGS 84/Pseudo-Mercator.* URL: https://epsg.io/3857 (visited on 2021-05-03).

[21] Tomas Straupis. *Test harness for Wang–Müller algorithm.* URL: https://dev.openmap.lt/webgl/wm.html (visited on 2021-05-15).

Appendices

Appendix A Code Listings

This section contains a subset of files for the Wang–Müller algorithm. As a reminder, full listings, including supporting programs, can be found on GitHub[15].

A.1 Re-Generating This Paper

As explained in section 3.5, illustrations in this paper are generated from a small list of sample geometries. To observe the source geometries or regenerate this paper, run this script (assuming the name of this document is MJ-MSC-FULL.PDF).

Listing 3 will extract the source files from the MJ-MSC-FULL.PDF to a temporary directory, run the top-level MAKE command, and display the generated document. Source code for the algorithm, as well as other supporting files, can be found in the temporary directory.

```bash
#!/bin/bash -eu
s=${1:-mj-msc-full.pdf}
d=$(mktemp -d)
f=mj-msc.pdf
l="$d/make.log"
echo "Extracting $s to workdir $d/"; pdfdetach -saveall -o "$d" "$s"
echo "Logs in $l ..."; make -j "$(nproc)" -C "$d" "$f" &> "$l" || {
    echo "Failed to generate. $l extract:"; tail -20 "$l"; exit 1
}
echo "Opening $d/$f ..."; xdg-open "$d/$f"
echo "$d/$f was closed. Removing $d"; rm -r "$d"
```

Listing 3: EXTRACT-AND-GENERATE

A.2 Function ST_SIMPLIFYWM

```
\set ON_ERROR_STOP on
SET plpgsql.extra_errors TO 'all';

-- wm_detect_bends detects bends using the inflection angles. No corrections.
drop function if exists wm_detect_bends;
create function wm_detect_bends(
  line geometry,
  dbgname text default null,
  dbggen integer default null,
  OUT bends geometry[]
) as $$
declare
  p geometry;
  p1 geometry;
  p2 geometry;
  p3 geometry;
  bend geometry;
  prev_sign int4;
  cur_sign int4;
  l_type text;
  dbgpolygon geometry;
begin
  l_type = st_geometrytype(line);
  if l_type != 'ST_LineString' then
    raise 'This function works with ST_LineString, got %', l_type;
  end if;

  -- The last vertex is iterated over twice, because the algorithm uses 3
  -- vertices to calculate the angle between them.
  --
  -- Given 3 vertices p1, p2, p3:
  --
  --            p1___ ...
  --             /
  -- ... _____/
  --      p3   p2
  --
  -- When looping over the line, p1 will be head (lead) vertex, p2 will be the
  -- measured angle, and p3 will be trailing. The line that will be added to
  -- the bend will always be [p3,p2].
  -- So once the p1 becomes the last vertex, the loop terminates, and the
  -- [p2,p1] line will not have a chance to be added. So the loop adds the last
  -- vertex twice, so it has a chance to become p2, and be added to the bend.
  for p in
      (select geom from st_dumppoints(line) order by path[1] asc)
      union all
      (select geom from st_dumppoints(line) order by path[1] desc limit 1)
     loop
    p3 = p2;
    p2 = p1;
    p1 = p;
    continue when p3 is null;
    cur_sign = sign(pi() - st_angle(p1, p2, p2, p3));

    if bend is null then
      bend = st_makeline(p3, p2);
    else
      bend = st_linemerge(st_union(bend, st_makeline(p3, p2)));
    end if;

    if prev_sign + cur_sign = 0 then
      if bend is not null then
        bends = bends || bend;
      end if;
      bend = st_makeline(p3, p2);
    end if;
    prev_sign = cur_sign;
  end loop;

  -- the last line may be lost if there is no "final" inflection angle. Add it.
  if (select count(1) >= 2 from st_dumppoints(bend)) then
    bends = bends || bend;
  end if;

  if dbgname is not null then
    for i in 1..array_length(bends, 1) loop
      insert into wm_debug(stage, name, gen, nbend, way) values(
        'bbends', dbgname, dbggen, i, bends[i]);

      dbgpolygon = null;
      if st_npoints(bends[i]) >= 3 then
          dbgpolygon = st_makepolygon(
            st_addpoint(bends[i], st_startpoint(bends[i]))
          );
      end if;
      insert into wm_debug(stage, name, gen, nbend, way) values(
        'bbends-polygon', dbgname, dbggen, i, dbgpolygon);
    end loop;
  end if;
end $$ language plpgsql;

-- wm_fix_gentle_inflections moves bend endpoints following "Gentle Inflection
-- at End of a Bend" section.
--
-- The text does not specify how many vertices can be "adjusted"; it can
-- equally be one or many. This function is adjusting many, as long as the
-- cumulative inflection angle is small (see variable below).
--
-- The implementation could be significantly optimized to avoid `st_reverse`
-- and array reversals, trading for complexity in wm_fix_gentle_inflections1.
drop function if exists wm_fix_gentle_inflections;
create function wm_fix_gentle_inflections(
  INOUT bends geometry[],
  dbgname text default null,
  dbggen integer default null
) as $$
declare
  len int4;
  bends1 geometry[];
  dbgpolygon geometry;
begin
  len = array_length(bends, 1);

  bends = wm_fix_gentle_inflections1(bends);
  for i in 1..len loop
    bends1[i] = st_reverse(bends[len-i+1]);
  end loop;
  bends1 = wm_fix_gentle_inflections1(bends1);

  for i in 1..len loop
    bends[i] = st_reverse(bends1[len-i+1]);
  end loop;

  if dbgname is not null then
    for i in 1..array_length(bends, 1) loop
      insert into wm_debug(stage, name, gen, nbend, way) values(
        'cinflections', dbgname, dbggen, i, bends[i]);

      dbgpolygon = null;
      if st_npoints(bends[i]) >= 3 then
        dbgpolygon = st_makepolygon(
          st_addpoint(bends[i],
            st_startpoint(bends[i]))
        );
      end if;

      insert into wm_debug(stage, name, gen, nbend, way) values(
        'cinflections-polygon', dbgname, dbggen, i, dbgpolygon);
    end loop;
  end if;
end $$ language plpgsql;

-- wm_fix_gentle_inflections1 fixes gentle inflections of an array of lines in
-- one direction. An implementation detail of wm_fix_gentle_inflections.
drop function if exists wm_fix_gentle_inflections1;
create function wm_fix_gentle_inflections1(INOUT bends geometry[]) as $$
declare
  -- the threshold when the angle is still "small", so gentle inflections can
  -- be joined
  small_angle constant real default radians(45);
  ptail geometry; -- tail point of tail bend
  phead geometry[]; -- 3 tail points of head bend
  i int4; -- bends[i] is the current head
begin
  for i in 2..array_length(bends, 1) loop
    -- Predicate: two bends will always share an edge. Assuming (A,B,C,D,E,F)
    -- is a bend:
    --           C________D
    --           /         \
    -- \________/          \_______/
    -- A       B           E       F
    --
    -- Then edges (A,B) and (E,F) are shared with the neighboring bends.
    --
    --
    -- Assume this curve (figure `inflection-1`), going clockwise from A:
    --
    --    \______B
    --    A      `-------. C
    --                   /
    --    G___ F         /
    --    /   `-----.____+ D
    --              E
    --
    -- After processing the curve following the definition of a bend, the bend
    -- [A-E] would be detected. Assuming inflection point E and F are "small",
    -- the bend needs to be extended by two edges to [A,G].
    select geom from st_dumppoints(bends[i-1])
      order by path[1] asc limit 1 into ptail;

    while true loop
      -- copy last 3 points of bends[i-1] (tail) to ptail
      select array(
        select geom from st_dumppoints(bends[i]) order by path[1] asc limit 3
      ) into phead;

      -- if the bend got too short, stop processing it
      exit when array_length(phead, 1) < 3;

      -- inflection angle between ptail[1:3] is "large", stop processing
      exit when abs(st_angle(phead[1], phead[2], phead[3]) - pi()) > small_angle;

      -- distance from head's 1st vertex should be larger than from 2nd vertex
      exit when st_distance(ptail, phead[2]) < st_distance(ptail, phead[3]);

      -- Between two bends, bend with smaller baseline wins when two
      -- neighboring bends can have gentle inflections. This is a heuristic
      -- that can be safely removed, but in practice has shown to avoid
      -- creating some very bendy lines.
      exit when st_distance(st_pointn(bends[i], 1), st_pointn(bends[i], -1)) <
        st_distance(st_pointn(bends[i-1], 1), st_pointn(bends[i-1], -1));

      -- Detected a gentle inflection.
      -- Move head of the tail to the tail of head
      bends[i] = st_removepoint(bends[i], 0);
      bends[i-1] = st_addpoint(bends[i-1], phead[3]);
    end loop;

  end loop;
end $$ language plpgsql;

-- wm_if_selfcross returns whether baseline of bendi crosses bendj.
-- If it doesn't, returns a null geometry.
-- Otherwise, it will return the baseline split into a few parts where it
-- crosses bendj.
drop function if exists wm_if_selfcross;
create function wm_if_selfcross(
  bendi geometry,
  bendj geometry
) returns geometry as $$
declare
  a geometry;
  b geometry;
  multi geometry;
begin
  a = st_pointn(bendi, 1);
  b = st_pointn(bendi, -1);
  multi = st_split(bendj, st_makeline(a, b));

  if st_numgeometries(multi) = 1 then
    return null;
  end if;

  if st_numgeometries(multi) = 2 and
    (st_contains(bendj, a) or st_contains(bendj, b)) then
    return null;
  end if;

  return multi;
end $$ language plpgsql;

-- wm_self_crossing eliminates self-crossing from the bends, following the
-- article's section "Self-line Crossing When Cutting a Bend".
drop function if exists wm_self_crossing;
create function wm_self_crossing(
  INOUT bends geometry[],
  dbgname text default null,
  dbggen integer default null,
  OUT mutated boolean
) as $$
declare
  i int4;
  j int4;
  multi geometry;
begin
  mutated = false;
  <<bendloop>>
  for i in 1..array_length(bends, 1) loop
    continue when abs(wm_inflection_angle(bends[i])) <= pi();
    -- sum of inflection angles for this bend is >180, so it may be
    -- self-crossing. Now try to find another bend in this line that
    -- crosses an imaginary line of end-vertices

    -- Go through each bend in the given line, and see if has a potential to
    -- cross bends[i]. The line-cut process is different when i<j and i>j;
    -- therefore there are two loops, one for each case.
    for j in 1..i-1 loop
```

```
      multi = wm_if_selfcross(bends[i], bends[j]);
      continue when multi is null;
      mutated = true;

      -- remove first vertex of the following bend, because the last
      -- segment is always duplicated with the i'th bend.
      bends[i+1] = st_removepoint(bends[i+1], 0);
      bends[j] = st_geometryn(multi, 1);
      bends[j] = st_setpoint(
        bends[j],
        st_npoints(bends[j])-1,
        st_pointn(bends[i], st_npoints(bends[i]))
      );
      bends = bends[1:j] || bends[i+1:];
      continue bendloop;
    end loop;

    for j in reverse array_length(bends, 1)..i+1 loop
      multi = wm_if_selfcross(bends[i], bends[j]);
      continue when multi is null;
      mutated = true;

      -- remove last vertex of the previous bend, because the last
      -- segment is duplicated with the i'th bend.
      bends[i-1] = st_removepoint(bends[i-1], st_npoints(bends[i-1])-1);
      bends[i] = st_makeline(
        st_pointn(bends[i], 1),
        st_removepoint(st_geometryn(multi, st_numgeometries(multi)), 0)
      );
      bends = bends[1:i] || bends[j+1:];
      continue bendloop;
    end loop;
  end loop;

  if dbgname is not null then
    insert into wm_debug(stage, name, gen, nbend, way) values(
      'dcrossings', dbgname, dbggen, generate_subscripts(bends, 1),
      unnest(bends)
    );
  end if;
end $$ language plpgsql;

drop function if exists wm_inflection_angle;
create function wm_inflection_angle (IN bend geometry, OUT angle real) as $$
declare
  p0 geometry;
  p1 geometry;
  p2 geometry;
  p3 geometry;
begin
  angle = 0;
  for p0 in select geom from st_dumppoints(bend) order by path[1] asc loop
    p3 = p2;
    p2 = p1;
    p1 = p0;
    continue when p3 is null;
    angle = angle + abs(pi() - st_angle(p1, p2, p3));
  end loop;
end $$ language plpgsql;

drop function if exists wm_bend_attrs;
drop function if exists wm_elimination;
drop function if exists wm_exaggeration;
drop type if exists wm_t_attrs;
create type wm_t_attrs as (
  adjsize real,
  baselinelength real,
  curvature real,
  isolated boolean
);
create function wm_bend_attrs(
  bends geometry[],
  dbgname text default null,
  dbggen integer default null
) returns wm_t_attrs[] as $$
declare
  isolation_threshold constant real default 0.5;
  attrs wm_t_attrs[];
  attr wm_t_attrs;
  bend geometry;
  i int4;
  needs_curvature real;
  skip_next boolean;
  dbglastid integer;
begin
  for i in 1..array_length(bends, 1) loop
    bend = bends[i];
    attr.adjsize = 0;
    attr.baselinelength = st_distance(st_startpoint(bend), st_endpoint(bend));
    attr.curvature = wm_inflection_angle(bend) / st_length(bend);
    attr.isolated = false;
    if st_numpoints(bend) >= 3 then
      attr.adjsize = wm_adjsize(bend);
    end if;
    attrs[i] = attr;
  end loop;

  for i in 1..array_length(attrs, 1) loop
    if dbgname is not null then
      insert into wm_debug (stage, name, gen, nbend, way, props) values(
        'ebendattrs', dbgname, dbggen, i, bends[i],
        jsonb_build_object(
          'adjsize', attrs[i].adjsize,
          'baselinelength', attrs[i].baselinelength,
          'curvature', attrs[i].curvature,
          'isolated', false
        )
      ) returning id into dbglastid;
    end if;

    -- first and last bends can never be isolated by definition
    if skip_next or i = 1 or i = array_length(attrs, 1) then
      -- invariant: two bends that touch cannot be isolated.
      if st_npoints(bends[i]) > 3 then
        skip_next = false;
      end if;
      continue;
    end if;

    needs_curvature = attrs[i].curvature * isolation_threshold;
    if attrs[i-1].curvature < needs_curvature and
       attrs[i+1].curvature < needs_curvature then
      attr = attrs[i];
      attr.isolated = true;
      attrs[i] = attr;
      skip_next = true;

      if dbgname is not null then
        update wm_debug
        set props=props || jsonb_build_object('isolated', true)
        where id=dbglastid;
      end if;
    end if;
  end loop;

  return attrs;
end $$ language plpgsql;

-- sm_st_split a line by a point in a more robust way than st_split.
-- See https://trac.osgeo.org/postgis/ticket/2192
drop function if exists wm_st_split;
create function wm_st_split(
  input geometry,
  blade geometry
) returns geometry as $$
declare
  type1 text;
  type2 text;
begin
  type1 = st_geometrytype(input);
  type2 = st_geometrytype(blade);
  if not (type1 = 'ST_LineString' and
          type2 = 'ST_Point') then
    raise 'Arguments must be LineString and Point, got: % and %', type1, type2;
  end if;
  return st_split(st_snap(input, blade, 0.00000001), blade);
end $$ language plpgsql;

-- wm_exaggerate_bend2 is the second version of bend exaggeration. Uses
-- non-linear interpolation by point azimuth. Slower, but produces nicer
-- exaggerated geometries.
drop function if exists wm_exaggerate_bend2;
```

```
create function wm_exaggerate_bend2(
  INOUT bend geometry,
  size float,
  desired_size float
) as $$
declare
  scale2 constant float default 1.2; -- exaggeration enthusiasm
  midpoint geometry; -- midpoint of the baseline
  points geometry[];
  startazimuth float;
  azimuth float;
  diffazimuth float;
  point geometry;
  sss float;
  protect int = 10;
begin
  if size = 0 then
    raise 'invalid input: zero-area bend';
  end if;
  midpoint = st_lineinterpolatepoint(st_makeline(
      st_pointn(bend, 1),
      st_pointn(bend, -1)
    ), .5);
  startazimuth = st_azimuth(midpoint, st_pointn(bend, 1));

  while (size < desired_size) and (protect > 0) loop
    protect = protect - 1;
    for i in 2..st_npoints(bend)-1 loop
      point = st_pointn(bend, i);
      azimuth = st_azimuth(midpoint, point);
      diffazimuth = degrees(azimuth - startazimuth);
      if diffazimuth > 180 then
        diffazimuth = diffazimuth - 360;
      elseif diffazimuth < -180 then
        diffazimuth = diffazimuth + 360;
      end if;
      diffazimuth = abs(diffazimuth);
      if diffazimuth > 90 then
        diffazimuth = 180 - diffazimuth;
      end if;
      sss = ((scale2-1) * (diffazimuth / 90)^0.5);
      point = st_transform(
        st_project(
          st_transform(point, 4326)::geography,
          st_distance(midpoint, point) * sss, azimuth)::geometry,
        st_srid(midpoint)
      );
      bend = st_setpoint(bend, i-1, point);
    end loop;
    size = wm_adjsize(bend);
  end loop;
end $$ language plpgsql;

-- wm_exaggerate_bend exaggerates a given bend. Uses naive linear
-- interpolation. Faster than wm_exaggerate_bend2, but result visually looks
-- worse.
drop function if exists wm_exaggerate_bend;
create function wm_exaggerate_bend(
  INOUT bend geometry,
  size float,
  desired_size float
) as $$
declare
  scale constant float default 1.2; -- exaggeration enthusiasm
  midpoint geometry; -- midpoint of the baseline
  splitbend geometry; -- bend split across its half
  bendm geometry; -- bend with coefficients to prolong the lines
  points geometry[];
begin
  if size = 0 then
    raise 'invalid input: zero-area bend';
  end if;
  midpoint = st_lineinterpolatepoint(st_makeline(
      st_pointn(bend, 1),
      st_pointn(bend, -1)
    ), .5);

  while size < desired_size loop
    splitbend = wm_st_split(bend, st_lineinterpolatepoint(bend, .5));
    -- Convert bend to LINESTRINGM, where M is the fraction by how
    -- much the point will be prolonged:
    -- 1. draw a line between midpoint and the point on the bend.
    -- 2. multiply the line length by M. Midpoint stays intact.
    -- 3. the new set of lines form a new bend.
    -- Uses linear interpolation; can be updated to gaussian or similar;
    -- then interpolate manually instead of relying on st_addmeasure.
    bendm = st_collect(
      st_addmeasure(st_geometryn(splitbend, 1), 1, scale),
      st_addmeasure(st_geometryn(splitbend, 2), scale, 1)
    );

    points = array((
        select st_scale(
          st_makepoint(st_x(geom), st_y(geom)),
          st_makepoint(st_m(geom), st_m(geom)),
          midpoint
        )
        from st_dumppoints(bendm)
        order by path[1], path[2]
      ));

    bend = st_setsrid(st_makeline(points), st_srid(bend));
    size = wm_adjsize(bend);
  end loop;
end $$ language plpgsql;


-- wm_adjsize calculates adjusted size for a polygon. Can return 0.
```

```
drop function if exists wm_adjsize;
create function wm_adjsize(bend geometry, OUT adjsize float) as $$
declare
  polygon geometry;
  area float;
  cmp float;
begin
  adjsize = 0;
  polygon = st_makepolygon(st_addpoint(bend, st_startpoint(bend)));
  -- Compactness Index (cmp) is defined as "the ratio of the area of the
  -- polygon over the circle whose circumference length is the same as the
  -- length of the circumference of the polygon". I assume they meant the
  -- area of the circle. So here goes:
  -- 1. get polygon area P.
  -- 2. get polygon perimeter = u. Pretend it's our circle's circumference.
  -- 3. get A (area) of the circle from u: A = u^2/(4pi)
  -- 4. divide P by A: cmp = P/A = P/(u^2/(4pi)) = 4pi*P/u^2
  area = st_area(polygon);
  cmp = 4*pi()*area/(st_perimeter(polygon)^2);
  if cmp > 0 then
    adjsize = (area*(0.75/cmp));
  end if;
end $$ language plpgsql;

-- wm_exaggeration is the Exaggeration Operator described in the WM paper.
create function wm_exaggeration(
  INOUT bends geometry[],
  attrs wm_t_attrs[],
  dhalfcircle float,
  intersect_patience integer,
  dbgname text default null,
  dbggen integer default null,
  OUT mutated boolean
) as $$
declare
  desired_size constant float default pi()*(dhalfcircle^2)/8;
  bend geometry;
  tmpint geometry;
  i integer;
  n integer;
  last_id integer;
begin
  mutated = false;
  <<bendloop>>
  for i in 1..array_length(attrs, 1) loop
    if attrs[i].isolated and attrs[i].adjsize < desired_size then
      bend = wm_exaggerate_bend2(bends[i], attrs[i].adjsize, desired_size);
      -- Does bend intersect with the previous or next
      -- intersect_patience bends? If they do, abort exaggeration for this one.

      -- Do close-by bends intersect with this one? Special
      -- handling first, because 2 vertices need to be removed before checking.
      n = st_npoints(bends[i-1]);
      if n > 3 then
        continue when st_intersects(bend,
          st_removepoint(st_removepoint(bends[i-1], n-1), n-2));
      end if;
      if n > 2 then
        tmpint = st_intersection(bend, st_removepoint(bends[i-1], n-1));
        continue when st_npoints(tmpint) > 1;
      end if;

      n = st_npoints(bends[i+1]);
      if n > 3 then
        continue when st_intersects(bend,
          st_removepoint(st_removepoint(bends[i+1], 0), 0));
      end if;
      if n > 2 then
        tmpint = st_intersection(bend, st_removepoint(bends[i+1], 0));
        continue when st_npoints(tmpint) > 1;
      end if;

      for n in -intersect_patience+1..intersect_patience-1 loop
        continue when n in (-1, 0, 1);
        continue when i+n < 1;
        continue when i+n > array_length(attrs, 1);

        -- More special handling: if the neigbhoring bend has 3 vertices, the
        -- neighbor's neighbor may just touch the tmpbendattr.bend; in this
        -- case, the nearest vertex should be removed before comparing.
        tmpint = bends[i+n];
        if st_npoints(tmpint) > 2 then
          if n = -2 and st_npoints(bends[i+n+1]) = 3 then
            tmpint = st_removepoint(tmpint, st_npoints(tmpint)-1);
          elsif n = 2 and st_npoints(bends[i+n-1]) = 3 then
            tmpint = st_removepoint(tmpint, 0);
          end if;
        end if;

        continue bendloop when st_intersects(bend, tmpint);
      end loop;

      -- No intersections within intersect_patience, mutate bend!
      mutated = true;
      bends[i] = bend;

      -- remove last vertex of the previous bend and first vertex of the next
      -- bend, because bends always share a line segment together this is
      -- duplicated in a few places, because PostGIS does not allow (?)
      -- mutating an array when passed to a function.
      bends[i-1] = st_addpoint(
        st_removepoint(bends[i-1], st_npoints(bends[i-1])-1),
        st_pointn(bends[i], 1),
        -1
      );

      bends[i+1] = st_addpoint(
        st_removepoint(bends[i+1], 0),
        st_pointn(bends[i], st_npoints(bends[i])-1),
        0
      );
      if dbgname is not null then
        insert into wm_debug (stage, name, gen, nbend, way) values(
          'gexaggeration', dbgname, dbggen, i, bends[i]);
      end if;
    end if;
  end loop;
end $$ language plpgsql;

create function wm_elimination(
  INOUT bends geometry[],
  attrs wm_t_attrs[],
  dhalfcircle float,
  dbgname text default null,
  dbggen integer default null,
  OUT mutated boolean
) as $$
declare
  desired_size constant float default pi()*(dhalfcircle^2)/8;
  leftsize float;
  rightsize float;
  i int4;
begin
  mutated = false;

  i = 1;
  while i < array_length(attrs, 1)-1 loop
    i = i + 1;
    continue when attrs[i].adjsize = 0;
    continue when attrs[i].adjsize > desired_size;

    if i = 2 then
      leftsize = attrs[i].adjsize + 1;
    else
      leftsize = attrs[i-1].adjsize;
    end if;

    if i = array_length(attrs, 1)-1 then
      rightsize = attrs[i].adjsize + 1;
    else
      rightsize = attrs[i+1].adjsize;
    end if;

    continue when attrs[i].adjsize >= leftsize;
    continue when attrs[i].adjsize >= rightsize;

    -- Local minimum. Elminate bend!
    mutated = true;
    bends[i] = st_makeline(st_pointn(bends[i], 1), st_pointn(bends[i], -1));

    -- remove last vertex of the previous bend and
    -- first vertex of the next bend, because bends always
    -- share a line segment together
    bends[i-1] = st_addpoint(
      st_removepoint(bends[i-1], st_npoints(bends[i-1])-1),
      st_pointn(bends[i], 1),
      -1
    );

    bends[i+1] = st_addpoint(
      st_removepoint(bends[i+1], 0),
      st_pointn(bends[i], st_npoints(bends[i])-1),
      0
    );
    -- the next bend's adjsize is now messed up; it should not be taken
    -- into consideration for other local minimas. Skip over 2.
    i = i + 2;
  end loop;

  if dbgname is not null then
    insert into wm_debug(stage, name, gen, nbend, way) values(
      'helimination',
      dbgname,
      dbggen,
      generate_subscripts(bends, 1),
      unnest(bends)
    );
  end if;
end $$ language plpgsql;


drop function if exists ST_SimplifyWM_Estimate;
create function ST_SimplifyWM_Estimate(
  geom geometry,
  OUT npoints bigint,
  OUT secs bigint
) as $$
declare
  lines geometry[];
  l_type text;
begin
  l_type = st_geometrytype(geom);
  if l_type = 'ST_LineString' then
    lines = array[geom];
  elseif l_type = 'ST_MultiLineString' then
    lines = array((select a.geom from st_dump(geom) a order by path[1] asc));
  else
    raise 'Unknown geometry type %', l_type;
  end if;

  npoints = 0;
  for i in 1..array_length(lines, 1) loop
    npoints = npoints + st_numpoints(lines[i]);
  end loop;
  secs = npoints / 33;
end $$ language plpgsql;
```

```
-- ST_SimplifyWM simplifies a given geometry using Wang & Müller's
-- "Line Generalization Based on Analysis of Shape Characteristics" algorithm,
-- 1998.
-- Input parameters:
-- - geom: ST_LineString or ST_MultiLineString: the geometry to be simplified
-- - dhalfcircle: the diameter of a half-circle, whose area is an approximate
--   threshold for small bend elimination. If bend's area is larger than that,
--   the bend will be left alone.
drop function if exists ST_SimplifyWM;
create function ST_SimplifyWM(
  geom geometry,
  dhalfcircle float,
  intersect_patience integer default 10,
  dbgname text default null
) returns geometry as $$
declare
  gen integer;
  i integer;
  j integer;
  line geometry;
  lines geometry[];
  bends geometry[];
  attrs wm_t_attrs[];
  mutated boolean;
  l_type text;
begin
  if intersect_patience is null then
    intersect_patience = 10;
  end if;
  l_type = st_geometrytype(geom);
  if l_type = 'ST_LineString' then
    lines = array[geom];
  elseif l_type = 'ST_MultiLineString' then
    lines = array((select a.geom from st_dump(geom) a order by path[1] asc));
  else
    raise 'Unknown geometry type %', l_type;
  end if;

  <<lineloop>>
  for i in 1..array_length(lines, 1) loop
    mutated = true;
    gen = 1;

    while mutated loop

      if dbgname is not null then
        insert into wm_debug (stage, name, gen, nbend, way) values(
          'afigures', dbgname, gen, i, lines[i]);
      end if;

      bends = wm_detect_bends(lines[i], dbgname, gen);
      bends = wm_fix_gentle_inflections(bends, dbgname, gen);
      select * from wm_self_crossing(bends, dbgname, gen) into bends, mutated;
      if not mutated then
        attrs = wm_bend_attrs(bends, dbgname, gen);

        select * from wm_exaggeration(bends, attrs,
          dhalfcircle, intersect_patience, dbgname, gen) into bends, mutated;
      end if;

      -- TODO: wm_combination

      if not mutated then
        select * from wm_elimination(bends, attrs,
          dhalfcircle, dbgname, gen) into bends, mutated;
      end if;

      if mutated then
        lines[i] = st_linemerge(st_union(bends));

        if st_geometrytype(lines[i]) != 'ST_LineString' then
          -- For manual debugging:
          --insert into wm_manual(name, way)
          --select 'non-linestring-' || a.path[1], a.geom
          --from st_dump(lines[i]) a
          --order by a.path[1];
          raise '[%] Got % (in %) instead of ST_LineString. '
          'Does the exaggerated bend intersect with the line? '
          'If so, try increasing intersect_patience.',
          gen, st_geometrytype(lines[i]), dbgname;
          --exit lineloop;
        end if;
        gen = gen + 1;
        continue;
      end if;
    end loop;
  end loop;

  if l_type = 'ST_LineString' then
    return st_linemerge(st_union(lines));
  elseif l_type = 'ST_MultiLineString' then
    return st_union(lines);
  end if;
end $$ language plpgsql;
```

Listing 4: WM.SQL

A.3 Function AGGREGATE_RIVERS

```
/* Aggregates rivers by name and proximity. */
drop function if exists aggregate_rivers;
create function aggregate_rivers() returns table(
  id integer,
  name text,
  way geometry
) as $$
declare
  c record;
  cc record;
  changed boolean;
begin
  while (select count(1) from wm_rivers_tmp) > 0 loop
    select * from wm_rivers_tmp limit 1 into c;
    delete from wm_rivers_tmp a where a.id = c.id;
    changed = true;
    while changed loop
      changed = false;
      for cc in (
        select * from wm_rivers_tmp a where
          a.name = c.name and
          st_dwithin(a.way, c.way, 500)
        ) loop
        c.way = st_linemerge(st_union(c.way, cc.way));
        delete from wm_rivers_tmp a where a.id = cc.id;
        changed = true;
      end loop;
    end loop; -- while changed
    return query select c.id, c.name, c.way;
  end loop; -- count(1) from wm_rivers_tmp > 0
  return;
end
$$ language plpgsql;

drop index if exists wm_rivers_tmp_id;
drop index if exists wm_rivers_tmp_gix;
drop table if exists wm_rivers_tmp;
create temporary table wm_rivers_tmp (id serial, name text, way geometry);
create index wm_rivers_tmp_id on wm_rivers_tmp(id);
create index wm_rivers_tmp_gix on wm_rivers_tmp using gist(way) include(name);

insert into wm_rivers_tmp (name, way)
  select p.vardas as name, p.shape as way from :srctable p;

drop table if exists :dsttable;
create table :dsttable as (
  select * from aggregate_rivers() where st_length(way) >= 50000
);
drop table wm_rivers_tmp;
```

Listing 5: AGGREGATE-RIVERS.SQL