UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

VALIDACIÓN Y EXTENSIÓN DE SOFTWARE ABIERTO PARA EL RECONOCIMIENTO DE REDES DE DRENAJE EN MODELOS DE TERRENO / *VALIDATION AND IMPROVEMENT OF AN OPEN-SOURCE SOFTWARE FOR DRAINAGE NETWORK EXTRACTION ON TERRAIN MODELS*

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DIEGO ANDRÉS GAJARDO JIMÉNEZ

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER
PROFESORA GUÍA 2:
LUISA PINTO LINCOÑIR

MIEMBROS DE LA COMISIÓN:
PATRICIO INOSTROZA FAJARDIN
PABLO BARCELÓ BAEZA
GILBERTO GUTIÉRREZ RETAMAL

Este trabajo ha sido parcialmente financiado por el proyecto Enlace Fondecyt ENL009/15, Vicerrectoría de Extensión, Universidad de Chile, 2015.

SANTIAGO DE CHILE
2016

RESUMEN DE LA TESIS PARA OPTAR
AL TÍTULO DE: Ingeniero Civil en Computación
Y GRADO DE: Magíster en Ciencias mención Computación
POR: Diego Andrés Gajardo Jiménez
FECHA: 23 de agosto, 2016
PROFESORAS GUÍA: Nancy Hitschfeld Kahler, Luisa Pinto Lincoñir

En el campo de geomorfología fluvial, para el estudio de un terreno resulta de suma importancia el reconocimiento de sus redes de drenaje, conjunto de cauces y surcos que conducen el flujo hídrico a través de él. Como herramienta de apoyo a la investigación en terreno, así como para la enseñanza en aula, es deseable—y un área de investigación activa—el poder determinar automáticamente potenciales redes de drenaje en modelos digitales de la región de estudio. Para ello existen programas como *Runnel*, software abierto desarrollado en la Universidad de Chile y con el cual se busca sustituir el uso del software comercial *RiverTools* en la enseñanza de este campo.

Runnel permite la lectura y visualización tridimensional de modelos de elevación digital de terrenos, así como el reconocimiento de redes de drenaje en éstos en base a algoritmos clásicos del área. Sin embargo, en comparación preliminar con los resultados provistos por la herramienta comercial *RiverTools* que se busca sustituir, se observa que es muy difícil establecer una correspondencia visual entre los resultados ambos programas. El propósito central de esta tesis consiste en la extensión y validación de *Runnel*, con la finalidad de determinar la satisfactoriedad de los algoritmos implementados con respecto a *RiverTools*, y de añadir otras funcionalidades de análisis necesarias para el estudio de terrenos. En materia de extensión, se agrega un algoritmo de publicación reciente para la detección de redes, junto con mecanismos de preprocesamiento del terreno para eliminar zonas planas y depresiones aisladas que dificultan el reconocimiento de redes de drenaje, y una herramienta simple para apoyar al investigador en la identificación de terrazas geológicas.

Para fines de validación, se proponen dos casos de estudio en base a terrenos reales, sobre los cuales se realiza una comparación visual de las redes de drenaje construidas por *Runnel* contra los resultados entregados por *RiverTools*. El primer caso de estudio involucra un modelo de terreno de baja resolución—generado por el proyecto *Shuttle Radar Topography Mission*—que cubre la zona de Chile central entre Valparaíso y Pichilemu. El segundo caso de estudio consiste en un modelo de mayor resolución, construido en el Departamento de Geología, que cubre un sector de la provincia de Petorca en la región de Valparaíso. Estas zonas son escogidas por tratarse de ubicaciones estudiadas con anterioridad en el Departamento de Geología, permitiendo a futuro determinar si los resultados entregados por el *software* permiten identificar elementos conocidos del terreno en dichas áreas.

El principal resultado de este trabajo muestra que el nuevo algoritmo implementado, en conjunto con los mecanismos de preprocesamiento del terreno, provee resultados visualmente similares a los de *RiverTools*. El aporte central de esta tesis es el de generar, mediante el trabajo interdisciplinario, un software abierto que resulta promisorio como sustituto de una herramienta comercial establecida.

Abstract

In this project we seek to validate and improve an open-source software tool for fluvial geomorphology, named Runnel. This tool is being developed by the Departments of Geology and Computer Science at University of Chile, with the goal of replacing the usage of commercial software RiverTools for instructional purposes in this field. The development of this tool began as an undergraduate dissertation in 2014, offering functionality for drainage network extraction on terrain elevation models.

The proposal described in this work involves visual comparison of drainage networks extracted by Runnel with those extracted by RiverTools, with the purpose of determining the efficacy of the implemented algorithms. We also seek to extend the existing software, adding a modern drainage network extraction algorithm, together with preprocessing algorithms to resolve flat zones and isolated depressions that hamper drainage recognition, and a simple tool to assist the researcher in geological terrace detection.

For this purpose, we suggest studying two scenarios based on real terrain. The first scenario involves a low resolution terrain elevation model–generated by the Shuttle Radar Topography Mission project–which covers the zones of central Chile between the cities of Valparaiso and Pichilemu. The second scenario relies on a higher resolution elevation model, generated by the Department of Geology, which covers a section of the Petorca province in the Valparaiso region. These two zones are chosen due to having been previously studied by the Department of Geology, making it possible to determine if the results provided by Runnel allow for the recognition of previously known landforms in said regions of study.

Among the main findings of our work, we highlight that the classical algorithms implemented in the base version of Runnel are unsatisfactory for the purpose of replacing RiverTools. On the other hand, we show that the modern drainage algorithm RWFlood implemented in our work, together with the usage of the Garbrecht-Martz preprocessing algorithms, produces results that are visually similar to those provided by RiverTools, and which are consistent with reference maps of rivers in central Chile, as well as the main streams in the Petorca area.

As future work, in terms of software we highlight the necessity of implementing postprocessing algorithms to construct a full drainage tree from the nodes identified by the drainage recognition algorithm, as well as implementing external memory support for operating on high resolution terrain models. With respect to validation, after implementing a postprocessing linking algorithm we suggest defining a similarity metric between trees, taking into account the Strahler order of network streams by way of a weighting scheme, as well as the geographic data associated to each node, to allow for a formal, data-based comparison between the results of both software tools.

Acknowledgments

My deepest gratitude goes out to my advisors Nancy Hitschfeld and Luisa Pinto, for guiding me attentively and providing ample feedback at all stages of this work; to the thesis committee, for their helpful input and suggestions to improve this thesis, and to my family and friends, for their invaluable support throughout this journey.

Thank you∼ ♪

Contents

1 Introduction **1**

1.1 Context 1

1.1.1 Runnel software 2

1.1.2 Problem of study 2

1.2 Goals and objectives 3

1.2.1 Main goal 3

1.2.2 Specific objectives 3

1.3 Methodology 3

1.4 Chapter contents 4

2 Antecedents and background **5**

2.1 Geomorphology 5

2.2 Digital terrain models 9

2.2.1 Data acquisition for DTMs 9

2.2.2 Types of DTMs 9

2.3 Rectangular grid model analysis 10

2.4 Drainage network detection algorithms 11

2.5 Large DEM processing 12

2.5.1 Drainage pattern recognition algorithms 14

2.5.2 Water flow path calculation algorithms 15

2.5.3 Strahler order calculation 15

2.5.4 Terrain preprocessing algorithms 16

2.6 Geographical information systems 18

3 Software analysis **19**

3.1 Base version of Runnel 19

3.1.1 Data Input subsystem . . . 19

3.1.2 Model Analysis subsystem . . . 20

3.1.3 Model Rendering subsystem . . . 23

3.2 Detected issues . . . 25

3.2.1 Drainage network extraction . . . 25

3.2.2 Drainage pattern classification . . . 28

3.2.3 Water flow path calculation . . . 28

3.2.4 Data input . . . 31

3.3 Other desirable techniques to be implemented . . . 31

3.4 Proposed solution . . . 32

4 Design and implementation **33**

4.1 Algorithm descriptions . . . 33

4.1.1 Peucker algorithm . . . 33

4.1.2 RWFlood algorithm . . . 35

4.1.3 Gleyzer algorithm . . . 38

4.1.4 Garbrecht-Martz flat resolution algorithm . . . 40

4.1.5 Garbrecht-Martz pit removal algorithm . . . 46

4.1.6 Normal vector similarity algorithm for terrace detection . . . 51

4.2 Implementation details . . . 51

4.2.1 RWFlood algorithm . . . 51

4.2.2 Gleyzer algorithm . . . 52

4.2.3 Garbrecht-Martz flat resolution algorithm . . . 53

4.2.4 Garbrecht-Martz pit removal algorithm . . . 53

4.2.5 Peucker algorithm fix . . . 55

4.2.6 Normal vector similarity algorithm for terrace detection . . . 56

5 Results and discussion **57**

5.1 Drainage network extraction on raw terrain . . . 57

5.2 Preprocessing techniques . . . 62

5.3 Drainage network extraction on fully preprocessed terrain . . . 65

5.4 Strahler order calculation . . . 72

5.5 Terrace visualization . . . 75

5.6 High resolution DEM . . . 77

5.7 Execution time . . . 80

6 Conclusions **83**

7 Future work **84**

8 Appendix A: Code listings **86**

8.1 RWFlood algorithm . . . 86

8.2 Gleyzer algorithm . . . 89

8.3 Garbrecht-Martz flat resolution algorithm . . . 93

8.4 Garbrecht-Martz pit removal algorithm . . . 99

8.5 Peucker algorithm fix . . . 106

8.6 Normal vector similarity algorithm for terrace detection . . . 107

9 Appendix B: Full results **111**

9.1 Drainage network extraction on raw terrain . . . 111

9.2 Preprocessing techniques . . . 111

9.3 Drainage network extraction on preprocessed terrain . . . 111

9.4 Strahler order calculation . . . 126

9.5 Terrace visualization . . . 129

9.6 High resolution DEM . . . 130

10 Bibliography **135**

List of Figures

1 Strahler order example 6

2 Drainage patterns in river networks 8

3 Spurious Strahler order increase in braided stream 16

4 Runnel software subsystems 19

5 Data Input subsystem modules 20

6 Model Analysis subsystem modules 21

7 Drainage Network Extraction module class diagram 21

8 Drainage Pattern Classification module class diagram 22

9 Water Flow Path Calculation module class diagram 23

10 Model Rendering subsystem modules 23

11 GLSL Shaders module class diagram 24

12 Terrain Renderer module class diagram 25

13 Comparison of RiverTools and Runnel Peucker algorithm results 26

14 Comparison of RiverTools and Runnel O'Callaghan algorithm results 27

15 Strahler order ≥ 3 calculation on Peucker-based network in Runnel 29

16 Runnel drainage pattern classification results 30

17 Peucker algorithm example diagram 34

18 RWFlood algorithm example diagram, flow direction assignment 37

19 RWFlood algorithm example diagram, water accumulation 38

20 Garbrecht-Martz imposed gradients example, first gradient 44

21 Garbrecht-Martz imposed gradients example, second gradient 45

22 Garbrecht-Martz imposed gradients example, combined gradient 46

23 Window size occurrence plot 54

24 Absence of drainage calculation in flat areas by Peucker algorithm 57

25 Drainage cycles generated by Peucker algorithm 58

26 RiverTools vs. RWFlood on raw terrain, higher areas 59

27 RiverTools vs. RWFlood on raw terrain, lower areas 60

28 RWFlood parallel flow example, central flats 61

29 RWFlood parallel flow example, southwestern flats 61

30 Peucker algorithm results on raw and flat-preprocessed flat zone 62

31 O'Callaghan algorithm results on raw and flat-preprocessed flat zone 63

32 RWFlood algorithm results on raw and flat-preprocessed flat zone 63

33 RWFlood algorithm results on raw and flat-preprocessed flat zone (2) 64

34 Example of depression breaching differences, Peucker algorithm 65

35 Example of depression breaching differences, O'Callaghan algorithm 65

36 Peucker algorithm results on raw and fully preprocessed DEM 66

37 O'Callaghan algorithm results on raw and fully preprocessed DEM 67

38 RiverTools and Runnel Peucker algorithm results on fully preprocessed terrain . . . 68

39 RiverTools and Runnel O'Callaghan algorithm results on fully preprocessed terrain 69

40 RWFlood algorithm results on raw and fully preprocessed DEM 70

41 RiverTools and Runnel RWFlood algorithm results on fully preprocessed terrain . . 71

42 IGM map of main rivers in central Chile 72

43 Strahler order calculation on preprocessed Peucker network before and after implementing the Gleyzer algorithm 73

44 Strahler order calculation on preprocessed RWFlood network with the Gleyzer algorithm 74

45 Gleyzer algorithm failure, local minimum 74

46 Gleyzer algorithm with depression breaching 75

47 Terrace visualization with 1 degree threshold on raw terrain 76

48 Terrace visualization with 1 degree threshold on fully preprocessed terrain 76

49 RiverTools and Peucker algorithm results on 4030x3080 DEM 77

50 RiverTools and O'Callaghan algorithm results on 4030x3080 DEM 78

51 RiverTools and RWFlood algorithm results on 4030x3080 DEM 79

52 IGM map of major streams in Petorca province section 80

53 Execution time plot for drainage algorithms 81

54 Execution time plot for preprocessing algorithms 82

55 Comparison of Peucker algorithm results before and after fixes 112

56 Comparison of RiverTools and Runnel RWFlood algorithm results 113

57 Comparison of Peucker algorithm results on raw and flat-preprocessed terrain 114

58 Comparison of Callaghan algorithm results on raw and flat-preprocessed terrain . . 115

59 Comparison of RWFlood algorithm results on raw and flat-preprocessed terrain . . . 116

60 Comparison of Peucker algorithm results on raw and depression-preprocessed terrain 117

61 Comparison of O'Callaghan algorithm results on raw and depression-preprocessed terrain 118

62 Comparison of RWFlood algorithm results on raw and depression-preprocessed terrain 119

63 Comparison of Peucker algorithm results on raw and fully preprocessed terrain . . . 120

64 Comparison of O'Callaghan algorithm results on raw and fully preprocessed terrain 121

65 Comparison of RWFlood algorithm results on raw and fully preprocessed terrain . . 122

66 Comparison of RiverTools and Runnel Peucker algorithm results on fully preprocessed terrain (full size) 123

67 Comparison of RiverTools and Runnel O'Callaghan algorithm results on fully preprocessed terrain (full size) 124

68 Comparison of RiverTools and Runnel RWFlood algorithm results on fully preprocessed terrain (full size) 125

69 IGM map of main rivers in central Chile (full size) 126

70 Comparison of Strahler order calculation on preprocessed Peucker network before and after implementing the Gleyzer algorithm (full size) 127

71 Strahler order calculation on preprocessed RWFlood network with the Gleyzer algorithm (full size) 128

72 Terrace visualization with 1 degree threshold on raw terrain 129

73 Terrace visualization with 1 degree threshold on fully preprocessed terrain 129

74 Comparison of RiverTools and Peucker algorithm results on 4030x3080 DEM 131
75 Comparison of RiverTools and O'Callaghan algorithm results on 4030x3080 DEM . 132
76 Comparison of RiverTools and RWFlood algorithm results on 4030x3080 DEM . . . 133
77 IGM map of major streams in Petorca province section 134

List of Algorithms

1 Peucker algorithm for drainage network extraction 33
2 RWFlood algorithm for flow direction assignment 35
3 RWFlood algorithm for flow accumulation calculation 36
4 MakeDictionaries function for Gleyzer algorithm 39
5 StreamOrdering recursive function for Gleyzer algorithm 40
6 Function to calculate the gradient towards lower terrain for Garbrecht-Martz imposed gradients algorithm 41
7 Function to calculate the gradient away from higher terrain for Garbrecht-Martz imposed gradients algorithm 42
8 Function to calculate the final imposed gradient for Garbrecht-Martz imposed gradients algorithm 43
9 Garbrecht-Martz depression breaching & filling algorithm 48
10 ComputeContributingArea function for Garbrecht-Martz depression breaching & filling algorithm 49
11 ComputePotentialOutlets function for Garbrecht-Martz depression breaching & filling algorithm 49
12 ComputePotentialBreachingSites function for Garbrecht-Martz depression breaching & filling algorithm 49
13 ChebyshevDistance function for Garbrecht-Martz depression breaching & filling algorithm 50
14 PerformFilling function for Garbrecht-Martz depression breaching & filling algorithm 50
15 SelectBreachingSite function for Garbrecht-Martz depression breaching & filling algorithm 50

16 PerformBreaching function for Garbrecht-Martz depression breaching & filling algorithm . . . 50

17 Normal vector similarity algorithm . . . 51

Listings

1 RWFlood algorithm class header file . . . 86
2 flood function in RWFloodAlgorithm.cpp . . . 87
3 calculateWaterAccumulation function in RWFloodAlgorithm.cpp . . . 88
4 getNumberStrahlerHorton function in arbol.cpp . . . 89
5 computeNetworkStrahlerOrdering function in arbol.cpp . . . 90
6 makeInflowingEdgeMap function in arbol.cpp . . . 90
7 makeUpstreamNodePerEdgeMap function in arbol.cpp . . . 91
8 streamOrdering function in arbol.cpp . . . 91
9 extractFlatIds function in garbrechtmartz.cpp . . . 93
10 run function in garbrechtmartz.cpp . . . 94
11 gradientTowardsLowerTerrain function in garbrechtmartz.cpp . . . 94
12 gradientAwayFromHigherTerrain function in garbrechtmartz.cpp . . . 96
13 combineGradients function in garbrechtmartz.cpp . . . 97
14 findHalfIncrementIds function in garbrechtmartz.cpp . . . 98
15 applyIncrements function in garbrechtmartz.cpp . . . 99
16 run function in gmpitremover.cpp . . . 99
17 computeContributingArea function in gmpitremover.cpp . . . 101
18 findPotentialOutlets function in gmpitremover.cpp . . . 102
19 findPotentialBreachingSites function in gmpitremover.cpp . . . 103
20 performBreaching function in gmpitremover.cpp . . . 104
21 nodeDistance function in gmpitremover.cpp . . . 105
22 calculateGrid function in peuckerdrainagenetwork.cpp . . . 106
23 Software defect fix for calculateGrid function in peuckerdrainagenetwork.cpp . . . 107
24 Header file for TerraceDetectionAlgorithm virtual class . . . 107
25 Header file for NormalVectorSimilarityAlgorithm class . . . 107
26 Implementation of NormalVectorSimilarityAlgorithm class . . . 108

1 Introduction

In this chapter we present the context of our work, our problem of study and our goals, objectives and methodology.

1.1 Context

In geology, fluvial geomorphology is the subfield devoted to the study of rivers and their impact on terrain relief. Fluvial geomorphology is chiefly concerned with the recognition and analysis of rivers and the whole network of smaller streams, permanent or intermittent, that feed them within a geographical region.

The analysis of drainage network structure, and its correlation to different kinds of geological characteristics and past events within the region of study, provides insight into the geological processes that contribute to the shaping of terrain relief throughout time, allowing one to achieve a better geological understanding of a given area.

The broadening availability of high-resolution elevation data for much of the planetary surface in the last decades has made it possible to build detailed digital models for a wide variety of fields of application. Hence, it is also useful for fluvial geomorphologists to have access to software tools that may assist them in performing scientific analyses of digital terrain models representing the region of interest. Through their use, researchers are able, for example, to determine whether a given area appears to have elements of interest that may warrant further on-site study, allowing them to focus their efforts by identifying geographical areas of interest more efficiently.

The Department of Geology at the University of Chile requires an educational and scientific tool—a geographical information system (GIS)—oriented towards fluvial geomorphology. While there are open source tools such as GRASS GIS that can be used for fluvial geomorphology, these tools are not specifically aimed at said field, but are rather general GIS's, which the Department considers a drawback for teaching due to an overabundance of irrelevant functions. On the other hand, RiverTools [7] is a fluvial geomorphology-specific tool and is currently used for educational purposes in the Department, but it is a commercial GIS whose cost per license amounts to 1,000 USD.

Thus, the Department of Geology seeks to build a fluvial geomorphology-specific open source GIS that may replace the instructional role RiverTools currently covers. In this context, as part of a software engineering dissertation, an early version of this replacement software named *Runnel* [30] is developed in late 2014.

The Runnel software will serve as a basis for our work in this thesis. In broad terms, we seek to evaluate, improve and extend the Runnel software.

We proceed to briefly describe the tools offered by Runnel in its first version.

1.1.1 Runnel software

Technologies Runnel is developed in the C++ programming language and built on the *Qt* development framework, due to the need for multi-platform support and the tools Qt offers for quick development of user interfaces. Runnel's visualization mechanisms are based on OpenGL 3.0, relying for development purposes on the external library GLEW.

Functionality The functionality of Runnel in its first version focuses mainly on three points: 1) 3D visualization of a terrain mesh built from an input DEM; 2) drainage network extraction and visualization over the terrain mesh, and 3) drainage pattern recognition and visualization on extracted drainage networks. It also includes basic functions for water flow path modeling.

Regarding visualization, Runnel allows for displaying a 3D triangle mesh built from the terrain model used as input. This mesh may be displayed either as a wireframe (i.e. with triangle faces being transparent) or with a solid color texture.

For drainage network extraction, Runnel implements the Peucker [29] and O'Callaghan [28] algorithms. Runnel also implements an extraction algorithm for triangulated irregular networks named *dihedral angle algorithm*, which recognizes streams depending on the angle formed between the containing planes of adjacent triangles in the generated mesh.

As for drainage pattern recognition, Runnel implements the Zhang-Guilbert algorithm [41] for classifying drainage patterns according to geometrical parameters computed over the extracted drainage networks.

1.1.2 Problem of study

In this first version of Runnel, we determined the existence of multiple issues related to drainage network extraction. Particularly, when the results given by Runnel are compared to results offered by RiverTools over the same DEM, we observe little visual similarity between the drainage networks built by each. This issue is regarded as critical for the future usefulness of Runnel, given that a proper and relatively accurate extraction of drainage networks from a DEM is an absolutely crucial step for any further kind of geomorphological analysis one may seek to perform, such as drainage area calculation, basin delimitation and drainage pattern recognition.

Given the above, the main purpose of this work resides in evaluating and improving the drainage network extraction tools implemented in Runnel.

Also, we find that the Strahler ordering scheme implementation is deficient and results in unusable order data, which motivates us to reimplement it correctly based on the Gleyzer algorithm. In addition, understanding Runnel to be an incremental project, we seek to implement other useful tools and techniques for DEM handling and analysis, such as preprocessing techniques for flat zone resolution and pit removal, and terrace recognition over the region of study. Finally, we shall suggest distinct lines of future work along which Runnel may be developed further.

1.2 Goals and objectives

1.2.1 Main goal

Evaluation, improvement and validation of drainage network extraction mechanisms, and extension of implemented functionalities in Runnel.

1.2.2 Specific objectives

- Software defect correction on base version of the Runnel software.
- Implementation and evaluation of the RWFlood algorithm for drainage network extraction.[23]
- Implementation and evaluation of the Garbrecht-Martz preprocessing algorithm for flat zone resolution. [13]
- Implementation and evaluation of the Garbrecht-Martz preprocessing algorithm for pit removal. [25]
- Implementation and evaluation of the normal vector similarity algorithm for terrace recognition.
- Implementation and evaluation of the Gleyzer algorithm for Strahler order calculation over the drainage network. [14]
- Proposal for future development of the Runnel software.

1.3 Methodology

On the subject of evaluation and validation, we propose two case studies based on two dataset. As our first dataset, we employ a 1074x856 DEM obtained from SRTM [8] data, based on WGS84 coordinates and with 90 meter horizontal resolution, covering the area of central Chile ranging from $32°51'15"$S to $34°38'15"$S.

Second, we also employ a 4030x3080 section of the high resolution 42576x42093 DEM produced by González [16], covering a portion of Chilean territory located between ($71°29'5"$W, $32°20'0"$S) and ($71°16'0"$W, $32°30'0"$S), to study the efficacy of the Runnel software on higher quality data involving higher memory requirements. We select this section of the DEM due to having the least elevation error, as reported by González, with respect to field measurements.

These datasets are chosen due to said areas being regions of study previously analyzed by Rodríguez [31] and González [16] at the Department of Geology, allowing for future validation involving the detection of known basins and landforms within these areas.

To assess the validity of our results on a given dataset, we obtain images of the drainage networks extracted from the same dataset through RiverTools. These images serve as a baseline to evaluate the efficacy of the drainage algorithms implemented in Runnel. RiverTools is chosen over other GIS's as it is the currently used software the Department of Geology seeks to substitute.

We perform a visual inspection to determine whether our modified version of Runnel provides results similar to those of RiverTools. The results provided by the base version of Runnel for the same region of study will be used as a control to assess our improvements.

Regarding terrain preprocessing for flat zone resolution and pit removal based on the Garbrecht-Martz algorithms, we compare the results given by any particular drainage network extraction algorithm both with and without preprocessing, determining if this procedure offers any clearly visible benefits. The central Chile dataset chosen for this study contains both flat areas and isolated depressions that allow us to evaluate this aspect of our proposal.

1.4 Chapter contents

In Chapter 2 (*Antecedents and background*) we outline the necessary background in fluvial geomorphology, digital terrain modeling, digital terrain processing and fluvial geomorphology-specific algorithms that are relevant to our work. Chapter 3 (*Software analysis*) presents a detailed account of the existing software, Runnel, describing its design, structure and function, along with the problems detected in its base version, and a definition of our proposed solution. Chapter 4 (*Design and implementation*) covers the implementation phase of our work, detailing the software defects corrected and the algorithms implemented as part of our proposal. In Chapter 5 (*Results and discussion*) we present cropped images of relevant results obtained through our modified version of the Runnel software, in comparison with results obtained through RiverTools, and we assess whether our implemented proposal induces improvements on the involved software tools. Chapter 6 (*Conclusions*) briefly summarizes the main findings of our work and addresses the completion of our listed objectives. Finally, in Chapter 7 (*Future work*) we describe further ways in which the Runnel software can be improved.

Appendix A contains relevant source code listings mentioned throughout Chapter 4, and Appendix B contains uncropped full resolution images of results produced in our work.

2 Antecedents and background

In this chapter we introduce the necessary background in fluvial geomorphology, digital terrain modeling, digital terrain processing and fluvial geomorphology-specific algorithms that are relevant to our work.

2.1 Geomorphology

Geomorphology is a subfield of geology devoted to the study of landforms and of the formative and erosive processes that configure them (which are termed *geomorphological processes*) over a range of timescales [17]. Over brief timescales, geomorphology focuses on the impact of terrain relief on human activity and viceversa, whereas over longer timescales it seeks to unravel the geological history of landforms and their evolution.

There are several kinds of geomorphological processes. *Tectonic processes* are linked to seismic plate activity and their effects on terrain relief, such as their role in terrain uplift and rock faulting. *Aeolian processes* are related to the activity of winds and their ability to erode and transport soil materials, especially in arid areas. *Glacial processes* involve the development of glaciers and their important effects on the landscape. There are also *fluvial processes*, involving the flow of surface water and its powerful modelling effect on terrain.

Fluvial geomorphology is the subfield of geomorphology that focuses on fluvial processes, studying both formation and structure of rivers and the geographical regions they cross. Water flow has a significant effect on the terrain relief of an area, both due to its erosional effect on soil, as well as its ability to transport the eroded material and deposit it at other regions along its route. Over periods ranging from decades to centuries, fluvial geomorphology is directly relevant to human activity, by virtue of the importance of freshwater as a resource and the necessity of its proper management in benefit of human communities and settlements. Over longer timescales, fluvial geomorphology seeks to understand the development of fluvial landforms and their relation to both permanent and intermittent waterbodies.

We define *hydrological basin* as the geographical extent covered by the set of channels, aquifers and underground flows that feed a given river. We also define *hydrographic basin* as a hydrological basin excluding aquifers and underground flow as objects of study, that is, considering only those landforms visible on the surface (which we collectively label as a *drainage network*). A basin is a basic unit of study in fluvial geomorphology.

Basins are delimited by *drainage divides*, chains of connected points of maximum local elevation on whose flanks surface water flows towards different geographical regions. We define *basin outlet* as the point of convergence of all streams within the drainage network of a basin. Basins can be considered recursive structures, given that each basin may be composed of multiple sub-basins, each with their own drainage networks and basin outlets. Thus, basin outlets connect basins to other bodies of water, either still (e.g. lakes or oceans) or flowing (e.g. streams belonging to a larger basin.)

Within a drainage network, a stream may be labeled as a *tributary* if it flows into another stream. Streams that have no tributaries are termed *sources*. Streams may also be classified according to their relative importance within the basin, based on the numbering scheme known as *Strahler order*. Under this scheme, source streams within a drainage network are assigned a Strahler order of 1, and whenever two or more streams of equal Strahler order converge, the Strahler order of the resulting stream increases by 1 (Figure 1). In this way, streams that are fed by a larger proportion of network streams will have a greater Strahler order, reflecting their relative importance within the basin. A greater Strahler order is therefore correlated with a larger drainage area, higher water discharge and greater channel size [33].

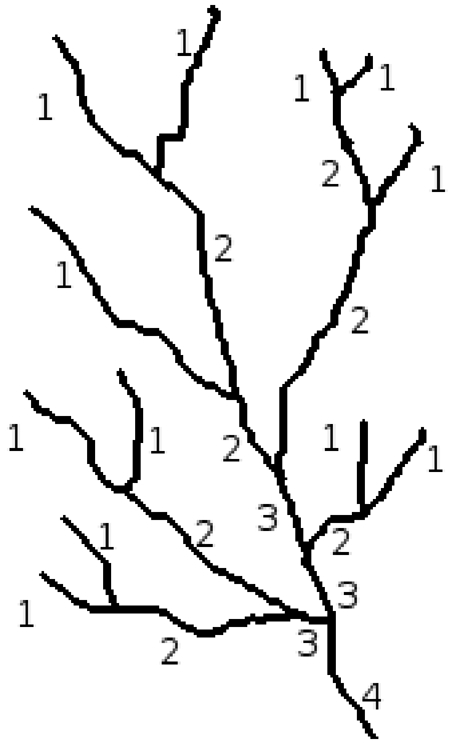


Figure 1: Example of the Strahler ordering scheme applied to a simple basin. Source streams have a Strahler order of 1, and when two streams of the same order merge, the order of the resulting stream increases by 1.

Different kinds of *drainage patterns* can be detected within a drainage network. A drainage pattern is a specific geometrical distribution of the streams that form a network. Drainage patterns are differentiated mainly by the angles of incidence between a main stream and the tributaries that feed it. The presence of certain kinds of drainage patterns is correlated with different types of geological or tectonic characteristics and events, allowing one to infer geomorphological details about the region of study [40, 36].

Some of the most common drainage patterns are as follows (Figure 2):

- **Dendritic:** The dendritic pattern is named such due to its visual similarity to the branches of a tree. This type of pattern lacks any directional preferences and may have broadly varying angles of incidence, and is observed predominantly in zones where subsoil rock offers directionally-uniform resistance to water flow.

- **Parallel:** This type of pattern is characterized by the presence of streams that run approximately parallel to each other, and tributaries with acute incidence angles. This type of pattern may be correlated, for example, with a pronounced terrain slope in the region of study, or the presence of parallel structural elements in the underlying rock.

- **Rectangular:** This type of pattern is induced by fractures and faulting in subsoil rock. The rectangular pattern is characterized by the presence of tributaries that form approximately straight angles of incidence with their main stream, as well as by approximately straight turns along the main stream itself.

- **Trellis:** The trellis pattern is similar to the rectangular pattern, but with its main characteristic being the presence of elongated tributaries that flow parallel to their main stream, joining into it through secondary tributaries that form approximately straight angles with the main stream. This pattern is often correlated to a high degree of folding in sedimentary rock.

- **Radial:** This type of pattern represents a concentric distribution of streams that flow outward from a central region of higher elevation. This pattern is commonly seen around volcanoes due to their conical shape.

- **Centripetal:** This pattern is analogous to the radial pattern, but with an opposite direction of flow, with multiple streams from different geographical areas converging into a central depression.

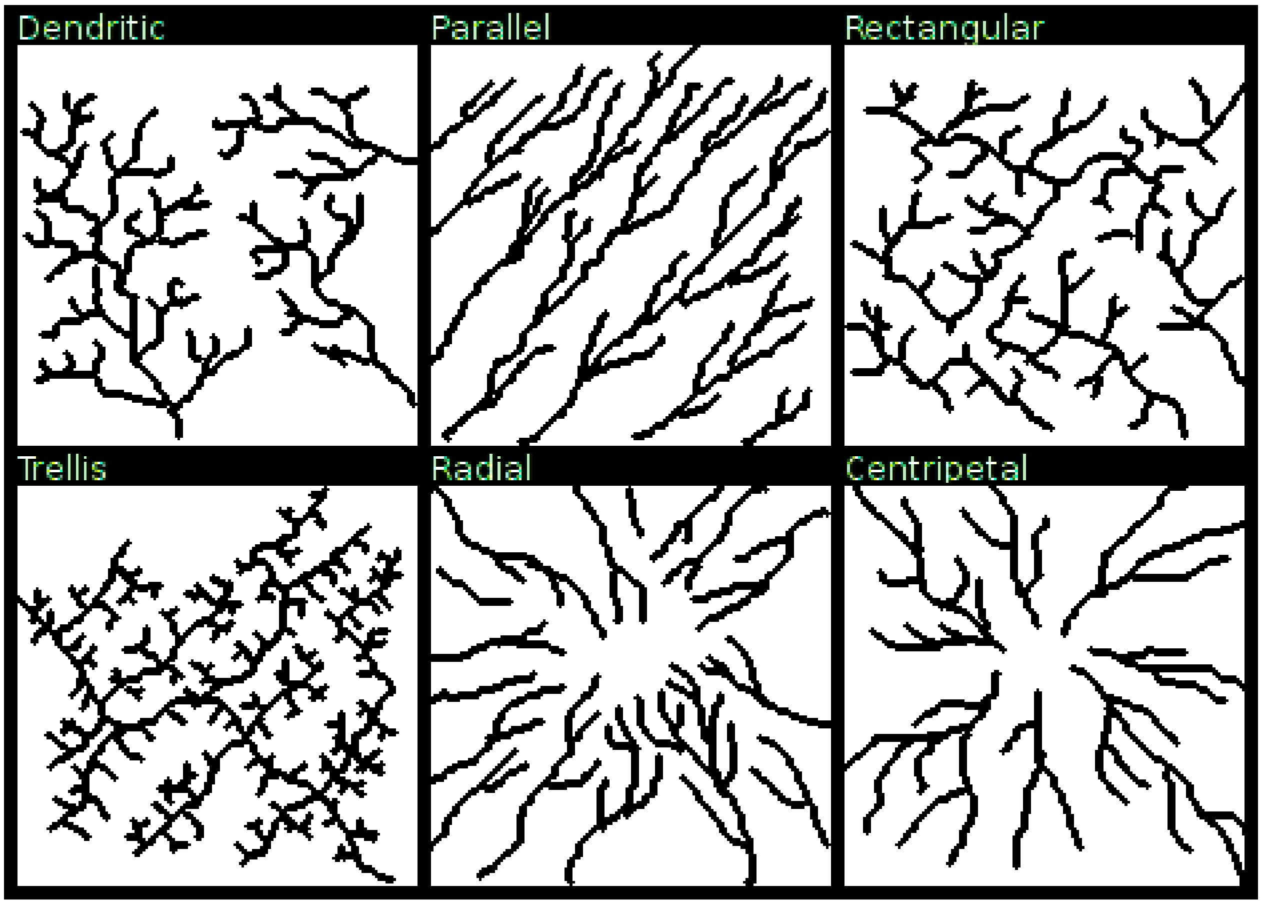


Figure 2: Common drainage patterns that may be detected in river networks.

A *geological terrace* is a landform consisting of a relatively flat surface with gentle slope, bounded by steeper slopes, usually formed by erosion through different types of geomorphological processes (e.g. fluvial or marine processes).

A *fluvial terrace* is a terrace built by fluvial processes. There are two kinds of fluvial terraces: *alluvial terraces* and *bedrock terraces*. Alluvial terraces tend to form around rivers in alluvium-floored valleys as they change their elevation over time, downcutting further into the soil due to the variable erosional effect of rivers on their own bed. Due to this, the alluvium deposits that previously formed the valley floor are exposed to the surface. As this process repeats, a staircase of alluvial terraces may become visible in the valley. Bedrock or strath terraces, on the other hand, occur analogously in rock-floored valleys as rivers downcut into the underlying bedrock.

Fluvial terraces therefore provide a history of elevation changes in a river, which may be linked to different phenomena that may have induced variations in discharge or erosional power over time. Thus, it is of interest to fluvial geomorphologists to determine the presence of fluvial terraces in the region of study.

2.2 Digital terrain models

Digital terrain models (DTM) are digital representations of the surface of a terrain for purposes of storage, processing and computer-based analysis. These models can be mathematical-analytical (e.g. polynomials or Fourier series) or graphical (e.g. contour lines or meshes) [11, 21]. Generation and analysis of DTMs is useful for many fields of study, including geology, ecology, hydraulic engineering, civil engineering and military engineering.

DTMs may include not only information regarding the elevation of each point within a geographical area, but also additional layers of information such as soil type, distribution of fauna and flora, land zoning or human settlements, depending on the model's purpose. In the fluvial geomorphology context presented in this work, only terrain elevation data will be required, with no other additional data layers; we shall refer to these models as *digital elevation models* (DEM).

2.2.1 Data acquisition for DTMs

Geographical data acquisition is the first step in building a DTM, with aerial photography and satellite data-based photogrammetry being the main technique employed for this purpose [21, 39]. Different data resolution may be required depending on the purpose for which said data is being collected. Aerial photography-based photogrammetry tends to produce high-resolution data over smaller geographical zones, whereas satellite-based photogrammetry allows for mass data acquisition over larger regions more efficiently. There are also other data acquisition techniques, such as the digitalization of topographical maps and contour lines drawn from on-site measurements.

There are multiple active projects aimed at acquiring and releasing geographical elevation data for most of the globe, such as *Shuttle Radar Topography Mission* (SRTM) [8], *Advanced Spaceborne Thermal Emission and Reflection Radiometer* [3], and *Landsat Program* [6]. In this work we shall rely on publicly available data from SRTM and from the work of González [16], without delving further into the topic of data acquisition.

2.2.2 Types of DTMs

As previously mentioned, there are different kinds of digital terrain models, which may be classified broadly into mathematical-analytical models, and graphical models. Mathematical-analytical models characterize the spatial distribution of geographic features of a terrain through equations and numerical conditions on its attributes. Graphical models usually characterize 3D geographical data through a 2D representation that allows for its digital visualization. In this work, we shall only address graphical models and issues related to them.

Graphical models are classified into three subtypes [27, 37]:

1. **Contour lines.** This type of model defines a set of elevation values for a given terrain, and for each of these values a contour line is stored as a sequence of x-y coordinates, thus subdividing the region of study into a set of irregular polygons bounded by said lines. Elevation data

for points located between contour lines is computed through interpolation methods between the lines that bound them.

Contour line models pose an advantage in that they can be readily digitalized from topographical maps and on-site elevation surveys. However, from a computational standpoint, contour line models require more storage space than other types of graphical models, and provide no computational advantage for analysis techniques. Therefore, contour line models are rarely used for digital terrain analysis [27], although they are often used for visualization purposes [37].

2. **Rectangular grids.** Rectangular grid models define a rectangular zone of axis-wise equidistant points over the region of study, together with an elevation value for each point in the grid. Elevation values for terrain points located inside each rectangle within the grid may be obtained through interpolation methods between the nodes of the rectangle.

   Grid models are advantageous in terms of ease of implementation, require less storage space and allow for the employment of efficient terrain attribute analysis techniques. However, being a regular subdivision of the region of study, a rectangular grid model may have an insufficient level of detail for mountainous areas with sharp changes in elevation, and simultaneously have an excessive level of detail for flatter areas, depending on the grid spacing parameters used [27].

3. **Triangulated irregular networks.** Triangulated irregular network models store an arbitrary set of terrain points with their respective elevation values, over which a planar triangulation is constructed. Elevation data for points contained within the resulting triangles may be obtained through linear interpolation between the nodes of the triangle.

   This type of model offers an advantage in dealing with terrains with variable degrees of elevation uniformity, allowing for the storage of a higher density of points in zones with sharp changes in elevation, and lower density for flatter zones, therefore representing the region of study with higher fidelity than grid models for an equal number of stored points [11]. However, triangulated irregular network models are both harder to implement and to analyze, with the development of analysis algorithms over TINs being an active area of research [27, 37].

In this work, we shall mostly deal with rectangular grid models and related algorithms.

2.3 Rectangular grid model analysis

There is a multitude of topographical and morphological attributes one might be interested in obtaining from a DEM, such as the distribution of peaks, plateaus, valleys and slopes, and numerical attributes describing them. In fluvial geomorphology there are also other rather complex terrain characteristics that are of interest, such as the partition of a terrain into hydrographical basins, the drainage networks contained within each of them, and specific attributes such as total drainage areas or basin dimensions [27].

In this section we shall mostly describe algorithms based on the rectangular grid terrain model, and focused mainly on drainage network detection.

2.4 Drainage network detection algorithms

As previously established, drainage network recognition in the region of study is a crucial aspect of fluvial geomorphology. Therefore, in this thesis, the detection of drainage networks over DEMs is of utmost importance. Conceptually, given a rectangular grid model, we seek to construct a directed graph that contains all of the grid nodes and whose edges reflect water flow, if existent, between adjacent nodes. For this purpose we use information such as the respective elevations of points within the grid, and the height difference between a point and its neighborhood in the grid. We shall consider water flow within a basin to favor a direction of vertical descent due to gravity.

It must be taken into account that the drainage network extracted from a digital terrain model does not necessarily correspond to streams observed in real terrain, due to multiple factors that include: 1) the accuracy of geographical data used to build the DEM; 2) the resolution of the particular DEM used for drainage network analysis, and 3) geological factors (e.g. soil types) that are not taken into account when building a model purely based on elevation data. Therefore, drainage networks obtained from DEMs should be validated through comparison with georeferenced aerial or satellite imagery of the region of study, contrasting the constructed network and the topographical and geomorphological features observed.

Below we describe some drainage network recognition algorithms for rectangular grid models. It must be noted that all of the following algorithms are $O(N)$ in time complexity; space usage may vary, but it is typically small compared to the space required to load the DEM itself.

The Peucker algorithm [29], one of the earliest efforts in this matter, analyzes every 2x2 point square within the grid independently, and in each square the highest point is flagged (if there are several highest points of equal elevation, all of them are flagged). After repeating this procedure for every square, the drainage network is constructed from all non-flagged points in the grid. While this algorithm is simple to implement, it poses a significant drawback in that it cannot directly identify flow direction between adjacent nodes, given that it only determines whether a point should be included in the constructed drainage network or not. Therefore, to build the directed network graph, additional constraints must be imposed, e.g. forcing a steepest descent flow direction between a node and its immediate neighborhood.

The O'Callaghan algorithm [28] is another landmark algorithm in this field, which introduces the concept of *flow accumulation*. Flow accumulation is a positive integer value that represents the amount of water transported along each point in the grid. In this particular algorithm, all points within the grid are initially assigned a flow accumulation value of 1. Then, processing all of the grid nodes from higher to lower elevation, a steepest descent flow direction is determined between each node and its immediate neighborhood, increasing the destination node's flow accumulation value by the source node's own accumulation. In this way, nodes that are located at lower elevations in the grid will tend to have a higher flow accumulation value, which is consistent with the notion of water descent due to gravity.

Unlike the Peucker algorithm, the O'Callaghan algorithm does compute a flow direction for each node in the grid, allowing one to construct the directed graph that represents the drainage network. A drawback of this algorithm, however, is that it only considers the immediate neighborhood of a point, with no regard for the broader structure of the area where the point is located; therefore, there may be considerable differences between results obtained at different resolutions of the same terrain, and the presence of data artifacts such as spurious depressions or peaks may have a significant effect.

The Ehlschlaeger-Metz algorithm [26], also known as *r.watershed* as per its implementation in GRASS GIS, employs a different approach based on a least cost path search technique. First, a set of potential basin outlets is defined; in the particular case of rectangular grid models, this set consists of all points on the grid boundary, along with all points adjacent to an oceanic point (identified by a special elevation value that represents the absence of data). Each potential basin outlet is then assigned a cost equal to its elevation. Afterwards, the directed graph is built incrementally beginning with the lowest cost point among all potential basin outlets, following a gentlest ascent path between adjacent neighbors. In each step of the construction process, adjacent neighbors are assigned their corresponding flow direction, determining as well in which order they should be later processed when computing flow accumulation.

The RWFlood algorithm [23] is another algorithm that constructs the drainage network in ascending elevation order from basin outlets. Conceptually, this algorithm floods the terrain beginning with the lowest point in the grid boundary, up towards the highest point in the grid. In each processing step, flooding elevation is increased, and for each new point that becomes flooded, flow direction is assigned towards the neighbor that flooded it. While this idea turns out to be similar to the Ehlschlaeger-Metz algorithm, the authors of RWFlood show that their algorithm outpaces the Ehlschlaeger-Metz algorithm in terms of running time.

2.5 Large DEM processing

In digital terrain modeling, given the growing availability of geographical datasets possessing both ample territorial coverage and high resolution, it is necessary to consider how to handle and process DEMs that may be too large to be loaded completely into *internal memory* (i.e. RAM). In essence, this subject can be approached in two ways (possibly combined): 1) aiming to reduce the total memory required to load a terrain model, and 2) expanding the total memory available by resorting to *external memory* (i.e. hard drives and other slower storage devices).

Reducing memory requirements The first approach involves the utilization of carefully designed data structures that exploit the inherent redundancy present in elevation data. This redundancy stems from the natural sloping patterns found in real terrains; as elevation values tend to be similar to other values in their neighborhood, blocks of values may be described exactly, or approximated, by space-wise smaller representations. We describe these representations as *compressed*.

Depending on whether there is any loss of information (i.e. whether we can reconstruct the original dataset from its compressed form or not), data compression techniques may be classified into lossless or lossy compression. Lossless compression allows one to recover the original data from its compressed form, while also reducing memory requirements, being particularly suited to applications where data accuracy is critical. Lossy compression, on the other hand, approximates the original data and allows for a partial loss of information in exchange for much better compression ratios than what may be achieved by lossless techniques, being desirable in applications where minimizing memory usage is of higher priority than maintaining fidelity with respect to the original dataset [32].

In the context of fluvial geomorphology, data accuracy plays a very important role in obtaining useful results from analysis techniques, with smoothed or low resolution DEMs generally being of limited use beyond providing a general view of an area of interest. Therefore, lossless compression techniques are more suitable for these purposes. However, it is also highly desirable for analysis algorithms to operate quickly, and compression algorithms introduce a trade-off in this regard. Whereas memory usage may be lowered, execution time may be increased significantly. This is due to analysis algorithms being unable to operate directly on the compressed representation, and therefore requiring decompression prior to their usage. Thus, for this field of application it is desirable to reach a compromise between reducing memory usage in a lossless manner and keeping execution times low.

Succinct data structures are particularly suited to fulfill this necessity. A succinct data structure is a space-optimized data representation that retains the ability to directly query and access the structure without prior decompression [18], allowing us to save memory without hampering performance significantly.

One approach to succinct representations of raster datasets is that of De Bernardo et al. [12], who propose two different modifications of the k^2-tree [10], as well as an n-dimensional generalization applied to three dimensions for spatial data, achieving a memory usage comparable to that of compressed GeoTIFF DEMs, yet still allowing for fast single node queries, and also providing faster range queries than can be achieved on an uncompressed GeoTIFF DEM. Ladra et al. [20] note that these structures perform well on datasets where the total amount of possible elevation values is relatively low or limited (e.g. 8-bit integer data or mostly flat, low relief terrain), but scale poorly in execution time as the number of different values grows (e.g. in 16-bit integer data, real values and/or terrains of high relief), and propose a different structure named k^2*-raster* which is shown to outperform the previous proposals in said scenario.

Expanding total memory The usage of external memory, on the other hand, entails the storage of partial processing results in hard drives and other mass storage devices. Due to the structural differences and the inherent processing speeds of internal and external memory, external memory usage for DEM processing requires specially-designed algorithms that make proper use of the particular characteristics and limitations of external memory in order to minimize costly disk accesses [38].

Most of the previously described algorithms for drainage network recognition belong to the internal memory category, given that they do not consider, in their design, the block size of the file system, nor the proper compartmentalization of grid processing into sectors whose partial results may be merged to construct the drainage network for the whole terrain.

Adaptation of internal memory algorithms for external memory usage depends crucially on how much information must be shared between different zones of the grid during the processing stage in order to construct the final result properly. For instance, in the case of the Peucker algorithm, adapting it to external memory is relatively straightforward, given that each 2x2 point square zone in the grid can be processed independently, with no reliance or influence whatsoever on the results produced in other squares; therefore, it is feasible to subdivide grid processing into smaller areas that can fit into internal memory, storing their results in separate files and merging them at a later stage. On the other hand, in the case of the RWFlood algorithm, it is not immediately clear to determine how to minimize disk access, given that during the flooding process one does not know beforehand which points of the grid must be processed next, and thus it is not straightforward to subdivide the grid processing optimally.

For building drainage networks in external memory, there are algorithms such as *TerraFlow* [9, 34], *r.watershed* [26], and an external memory adaptation of RWFlood named *EMFlow* [15].

In this work we shall only employ internal memory algorithms without compression.

2.5.1 Drainage pattern recognition algorithms

There are few algorithms described in the literature for digital drainage pattern recognition, given that the very definition of a drainage pattern is qualitative and lacks any precise formal specifications. Algorithms for this purpose specify and measure geometrical parameters in order to classify drainage networks according to different drainage patterns. This approach is taken by Zhang and Guilbert [41], for instance, who characterize four relevant geometrical indicators in drainage networks:

- **Angle of incidence between tributaries and main streams:** Angles formed between a stream and the tributaries that feed it are an important factor in drainage network classification, as they allow for distinguishing between dendritic or parallel patterns, and patterns with approximately straight angles such as the trellis and rectangular patterns. Thus, the average angle of incidence of stream tributaries may be considered as a classification metric.
- **Tributary stream sinuosity:** The sinuosity value of a stream is a quantification of flow direction changes along the stream, and its calculation on tributary streams is another relevant factor to distinguish between drainage patterns. For example, parallel patterns tend to manifest elongated tributary streams with low sinuosity, whereas trellis or rectangular patterns tend to have sharp turns in tributaries or main streams.
- **Length ratio between tributaries and main streams:** The length ratio between a river and its tributaries is another useful metric for drainage pattern classification. In the case of parallel and trellis patterns, tributary streams tend to be relatively longer, when compared to the main stream, than those in dendritic and rectangular patterns, and therefore the length ratio may also contribute towards pattern differentiation.
- **Basin elongation or depth-amplitude ratio:** The elongation of a drainage network is defined as the ratio between the depth and amplitude of its minimum bounding rectangle, where amplitude is measured in the flow direction of the main stream. A network is more elongated if its depth-amplitude ratio is higher. This metric allows for distinguishing parallel and dendritic patterns, for example, with the latter usually being less elongated.

The authors establish empirical, quantitative thresholds over the previously described metrics in order to perform drainage network pattern classification. An important limitation to consider is that their work only allows for distinguishing between dendritic, parallel, trellis and rectangular patterns. Certain patterns such as the radial and centripetal patterns are not made distinct by the geometrical properties of the streams themselves, but rather due to the spatial distribution of different drainage networks over the terrain, and thus are not covered by the proposed categorization.

2.5.2 Water flow path calculation algorithms

In order to differentiate basins within the region of study, it is of interest for researchers to have the capability of computing water flow paths from arbitrary points within the terrain. This may be understood as a reduced version of drainage network extraction algorithms, given that one only needs to compute the network path originating in a specific point.

The O'Callaghan algorithm may be readily adapted for this purpose: given a point in the grid, we determine which of its neighbors is located at the lowest elevation, and an edge is formed towards it, repeating said procedure along the constructed path until a locally minimal elevation is reached at a basin outlet. This greedy algorithm, however, has a drawback in that it requires preprocessing the DEM to remove isolated depressions, given that, being local minima, they may end up trapping water flow and halting algorithm execution without building a full path towards an actual basin outlet.

For triangulated irregular networks, there is also the gradient algorithm for this purpose. Initially, the gradient vector is calculated for every triangle within the mesh. Then, given an arbitrary initial point in the mesh (not necessarily a triangle node), the underlying triangle is determined, and an intercept point between its gradient vector and another triangle in the mesh is calculated. With this information, an edge can be drawn between the initial point and the intercept point, and the procedure may be repeated along this path. However, this algorithm is also vulnerable to isolated depressions like the previously mentioned grid algorithm, and thus also requires a preprocessing step on the DEM.

2.5.3 Strahler order calculation

Calculating the Strahler order of all streams within a drainage network is not trivial. In its simplest definition, the Strahler order of a stream t, $O(t)$, is a positive integer number computed as follows:

- If t is a source, $O(t) = 1$.
- Otherwise, given $m = max(O(t' \in Tributaries(t)))$:
  - If $|\{t' \in Tributaries(t) : O(t') = m\}| > 1$, then $O(t) = m + 1$.
  - Otherwise, $O(t) = m$.

However, this definition turns out to be problematic in drainage networks that contain braided streams, that is, streams that split into two or more substreams that rejoin downstream. In these cases, given that Strahler order of streams never decreases in the downstream direction, and considering that the resulting substreams have the same Strahler order as the original stream, a spurious Strahler order increase will occur when said substreams merge together (Figure 3).

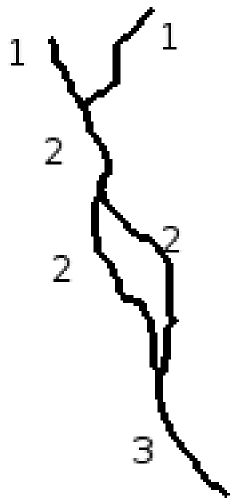


Figure 3: Example of incorrect Strahler order calculation in a braided stream. The order 2 stream splits into two substreams, which later merge and erroneously produce an order 3 stream according to the described algorithm. Braided streams in nature tend to split and merge many times throughout their path, which would induce many of these spurious Strahler order increments.

To address this problem, Gleyzer et al. [14] propose a modification to this definition. The essential idea behind their proposal is to augment the data structure storing Strahler order values with an additional piece of information that references the specific stream in the network where each value originated. Thus, when two streams of equal Strahler order merge, the Strahler order of the resulting stream is only incremented if their order origins are different. Through this approach, when a stream splits, all of its substreams will be linked to the same Strahler order origin, and therefore the Strahler order of the stream resulting from their merging will be the same as in the original stream before the split.

2.5.4 Terrain preprocessing algorithms

There are certain conditions in DEMs that may hamper the proper functioning of drainage analysis algorithms, for which a preprocessing step may be required. The most self-evident case is when there is an absence of elevation data for specific points or zones of the region of study, which may occur due to errors or technical limitations in the data acquisition process from which the DEM is built. In SRTM data, these *voids* usually appear in areas located at very high elevations, such as in the highest zones of the Andes mountain range. The absence of elevation data may impede algorithms from obtaining correct results in surrounding areas. To overcome this type of problem, the DEM may be preprocessed employing interpolation techniques to fill terrain voids.

Another problematic condition involves the presence of isolated depressions or *pits*. By isolated depressions, we refer to a set of contiguous points in the DEM with a lower elevation value than their containing boundary, having no outlet for flow routing. The presence of pits is problematic in that it leads to the failure of drainage analysis algorithms that rely on a greedy steepest descent strategy (e.g. O'Callaghan algorithm). It may be necessary, therefore, to perform preprocessing procedures to eliminate these pits in order to compute the drainage network more accurately.

A third kind of problematic condition refers to the existence of *flats*, contiguous regions of equal elevation. These flat surfaces hinder attempts at accurate drainage network extraction from the DEM, given that it is not trivial to compute water flow directions correctly over them. Thus, some preprocessing procedures may attempt to modify the elevation data in these regions to allow for proper flow routing.

Pits and flats often arise as artifacts during the terrain digitalization process, either due to insufficient horizontal or vertical resolution of the data acquisition technique employed, or introduced by the sampling process performed on the original dataset during DEM construction. Eliminating these elements from a DEM is not straightforward, as it is not possible to distinguish with certainty whether a given element is an artifact that must be removed, or an actual topographical feature that exists in the region of study [22]. Therefore, there are different approaches for dealing with pits and flats, with advantages and drawbacks that may depend on the particular topographic attributes of the terrain on which they are applied.

Pit removal The first attempts aimed at depression removal in the literature are based on the *elevation smoothing* approach introduced by O'Callaghan and Mark [28]. The elevation smoothing approach involves one or multiple passes of an operator that modifies the elevation data of all points on the DEM, based on a weighted sum of their own values and those of their eight immediate neighbors. Besides decreasing the correspondence between the terrain model elevation data and the actual topography of the region of study, this approach has an important drawback in that it will tend to produce very large flat areas when applied to DEMs of low relief [35]. This method also fails to remove deeper depressions that may require too large an amount of operator passes to resolve [19].

Most other early efforts to resolve pits are based on a *sink filling* approach [19]. Conceptually, the sink filling technique involves detecting isolated depressions and *filling* them by raising the elevation of their nodes up to the level of the depression boundary, resulting in a flat area across which water flow may be subsequently routed through flat resolution algorithms. The efficacy of this method, therefore, depends directly on the efficacy of the flat resolution algorithms employed to deal with the flats newly introduced.

Garbrecht and Martz [25] present a more complete approach based on the concept of *depression breaching*. In their work, they observe that isolated depressions in DEMs are not only generated by underestimation of elevation data (i.e. points whose elevation is erroneously lower than in reality), as the sink filling method implicitly assumes, but that they may also be introduced by *overestimation*, creating spurious dam-like features that may block actual flow paths. Hence, their breaching method seeks, as a first step before filling a pit, if a pit outlet could instead be created by decreasing the elevation of a small number of nodes in the pit boundary.

Flat resolution With respect to flat area resolution, the method of Jenson and Domingue [19] involves multiple passes over the terrain, iteratively assigning flow directions towards lower terrain, beginning at flat boundaries and progressing inwards. A drawback of this method, as presented by Tribe [35], is that many unrealistic parallel streams tend to result across flat areas, when in reality one might expect these streams to be merged.

The improvement to the Jenson-Domingue method proposed by Tribe [35] involves defining a main flow path in flat areas, drawn across the shortest path from the inflow node to the closest outlet, and forcing other streams in the flat region to converge towards it. A problem with this approach, identified by Garbrecht and Martz [24], is that the greedy strategy employed to define main convergence paths may unrealistically force streams to flow upwards to cross areas of higher elevation.

The *imposed gradients* algorithm developed by Garbrecht and Martz [13] incorporates the notion that surface water in real terrain tends to flow both towards lower elevation, and *away* from higher elevation; the previously mentioned approaches only consider the first observation. The Garbrecht-Martz approach involves applying these observations to introduce small elevation increments to cells within flat areas. For this purpose, two separate elevation gradients are calculated for a flat. Each gradient conceptually represents a very small slope to be added on top of the flat area, descending towards lower elevation and descending away from higher elevation respectively. The linear addition of these two slopes is then computed to determine how the elevation data in the flat area should be incremented for each node. The final result of this process is the superimposition of a 'microrelief' on the flat area, allowing for unambiguous calculation of flow direction for every node.

2.6 Geographical information systems

We define *geographical information system* (GIS) as a software system designed for handling geographical data and digital terrain models. There are multiple GIS's that are relevant to fluvial geomorphology, such as the following:

- **ArcGIS**: ArcGIS [2] is a commercial license, general purpose GIS. ArcGIS offers functionalities for working with multiple data layers on DTMs, as well as an array of analysis tools for different fields of application.
- **AquaVeo WMS**: AquaVeo Watershed Modeling System [1] is a commercial license GIS focused specifically on fluvial geomorphology over brief timescales, aimed at professionals working in the hydraulic engineering field. This software offers tools for digital modeling and simulation of basins and their modification through channels, dams, etc.
- **GRASS GIS**: GRASS GIS [4] is a multi-platform, general purpose, open source GIS. Similar to ArcGIS, GRASS GIS is oriented towards multilayered terrain data models, providing tools for their modification and analysis.
- **RiverTools**: RiverTools [7] is a commercial license GIS focused specifically on fluvial geomorphology. RiverTools offers a diversity of tools for basin analysis, such as drainage network extraction, computation of various metrics and geomorphological data plotting.

3 Software analysis

In this chapter we present the software structure of the base version of Runnel, the flaws spotted in its functionality, and our proposed solution to address them.

3.1 Base version of Runnel

In broad terms, the software is divided into four major subsystems (Figure 4): the User Interface system, responsible for handling user input and providing access to the other systems; the Data Input system, responsible for loading terrain elevation data, transforming terrain data between coordinate systems, and constructing the digital terrain model in internal memory; the Model Analysis system, responsible for executing analysis algorithms on the digital terrain model and reporting analysis results; and the Model Rendering system, responsible for drawing a 3D mesh representing the digital terrain model and displaying analysis results overlaid on the mesh.

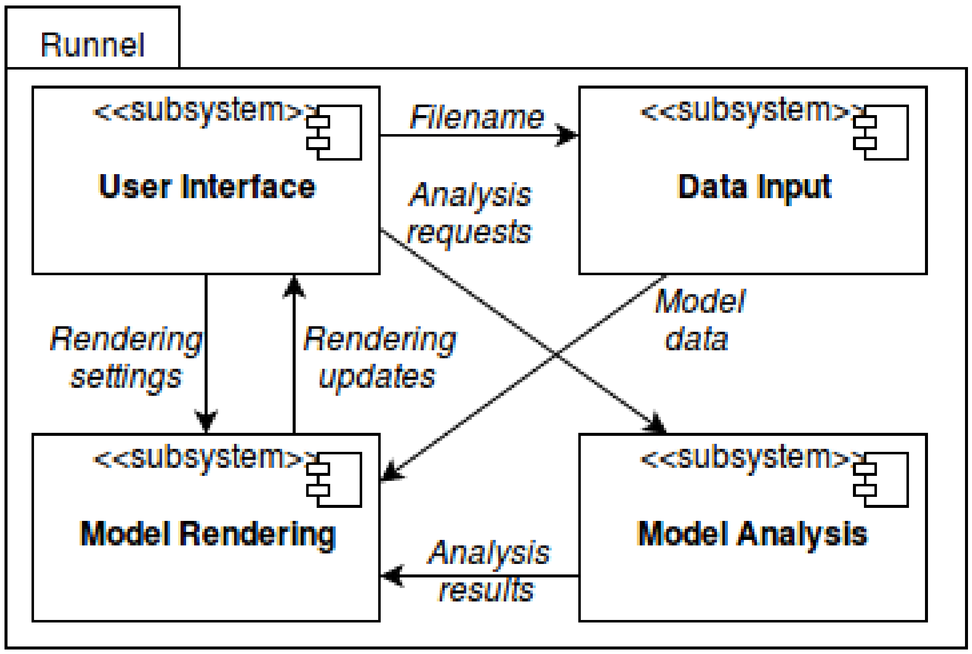


Figure 4: Diagram representing the Runnel software in terms of its subsystems and their interactions.

We proceed to describe the most important elements of the Data Input, Model Analysis and Model Rendering subsystems.

3.1.1 Data Input subsystem

The Data Input system (Figure 5) is composed of three main modules:

- The *GeoTIFF Parser* module is responsible for opening input files in GeoTIFF format and extract the coordinate and elevation data contained within.
- The *.runnel Parser* module is responsible for reading and writing files in .runnel format, a convenience format provided by the application for storing triangulation data constructed from GeoTIFF raster input.
- The *Google Earth Data Extractor* module is in charge of reading and extracting elevation data from the Google Earth plugin accessible from the user interface, allowing the user to obtain low-resolution data for most areas of the globe.

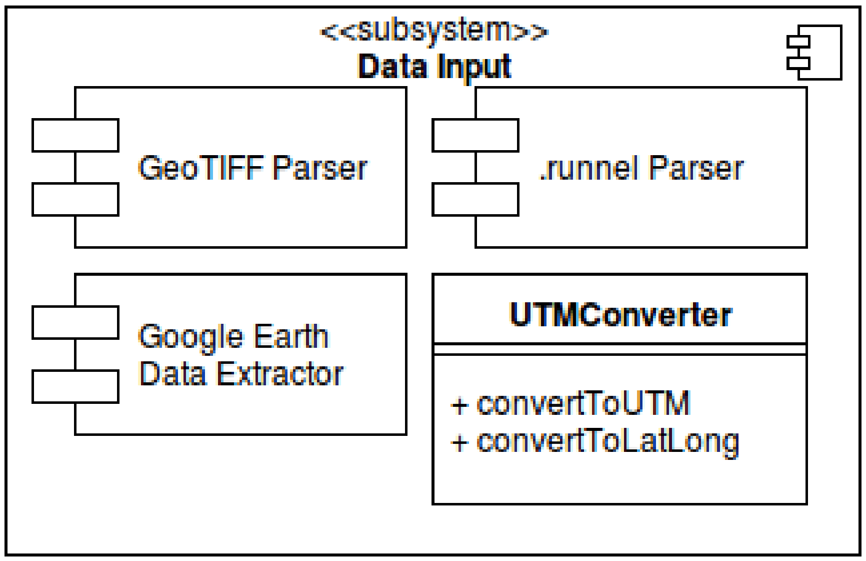
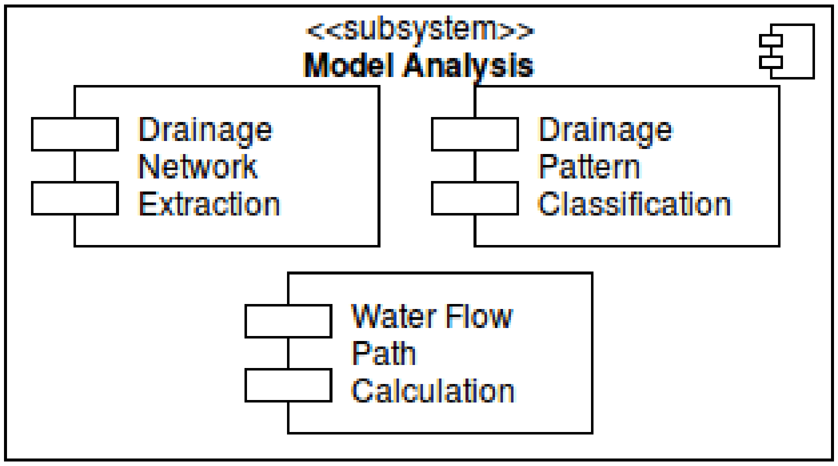


Figure 5: Data Input subsystem and its main modules, along with the utility class UTMConverter.

The subsystem also includes utility classes, of which the most relevant is *UTMConverter*, a class that provides functions to convert between latitude-longitude and Universal Transverse Mercator coordinates.

3.1.2 Model Analysis subsystem

The Model Analysis subsystem (Figure 6) is based on the *Strategy* design pattern. For each type of model analysis, the subsystem provides an abstract base class which defines an interface that particular algorithm implementation classes must conform to. Hence, to add an analysis algorithm, the implementation must inherit from the base class and implement its interface.



<<subsystem>>
Model Analysis
Drainage Network Extraction
Drainage Pattern Classification
Water Flow Path Calculation

Figure 6: Main modules that compose the Model Analysis subsystem, according to their function.

The Model Analysis subsystem is divided into three modules, according to the type of analysis performed:

- The *Drainage Network Extraction* module (Figure 7) is in charge of applying drainage network algorithms, such as the Peucker and O'Callaghan algorithms previously described, on the model data.

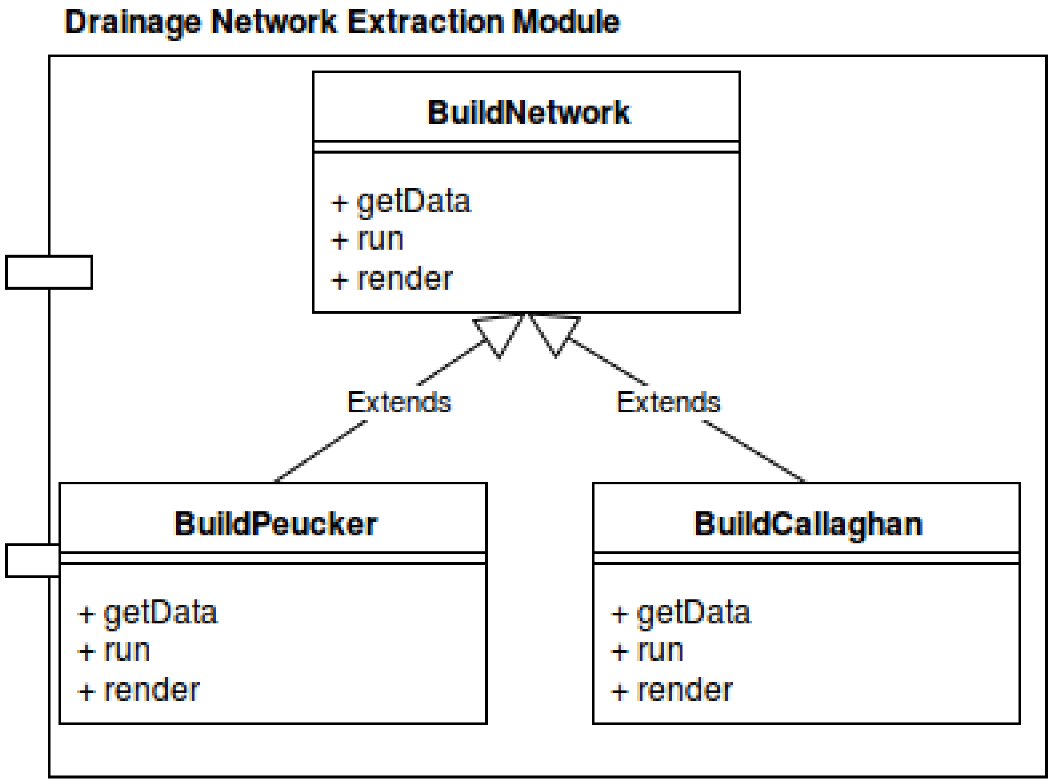


Figure 7: Diagram of class structure within the Drainage Network Extraction module. Drainage network algorithms inherit from the BuildNetwork class and implement its virtual functions *getData*, *run* and *render*.

- The *Drainage Pattern Classification* module (Figure 8) is responsible for executing drainage pattern classification algorithms (currently only the Zhang-Guilbert algorithm) on the drainage networks extracted through the previous module.

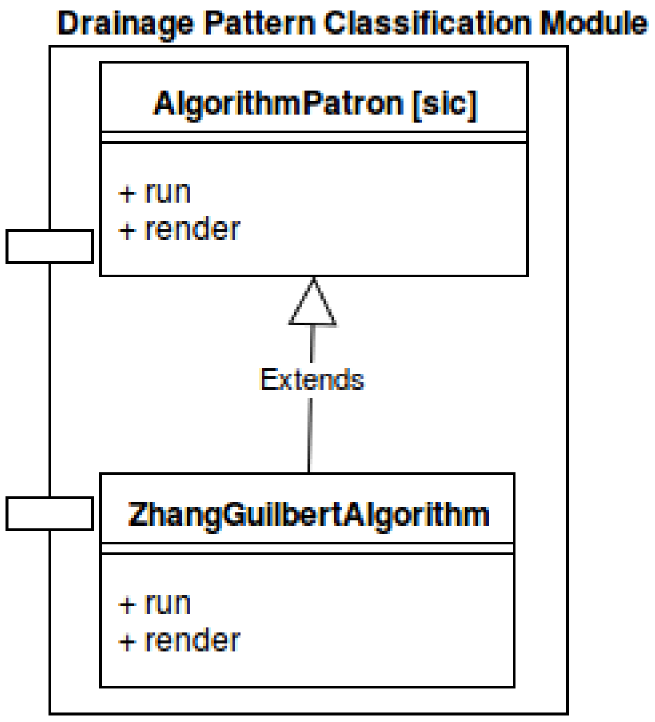


Figure 8: Class diagram for the Drainage Pattern Classification module. Pattern recognition algorithms must inherit from the AlgorithmPatron class and implement its virtual functions *run* and *render*.

- The *Water Flow Path Calculation* module (Figure 9) is in charge of determining water flow paths from mesh points specified by the user.

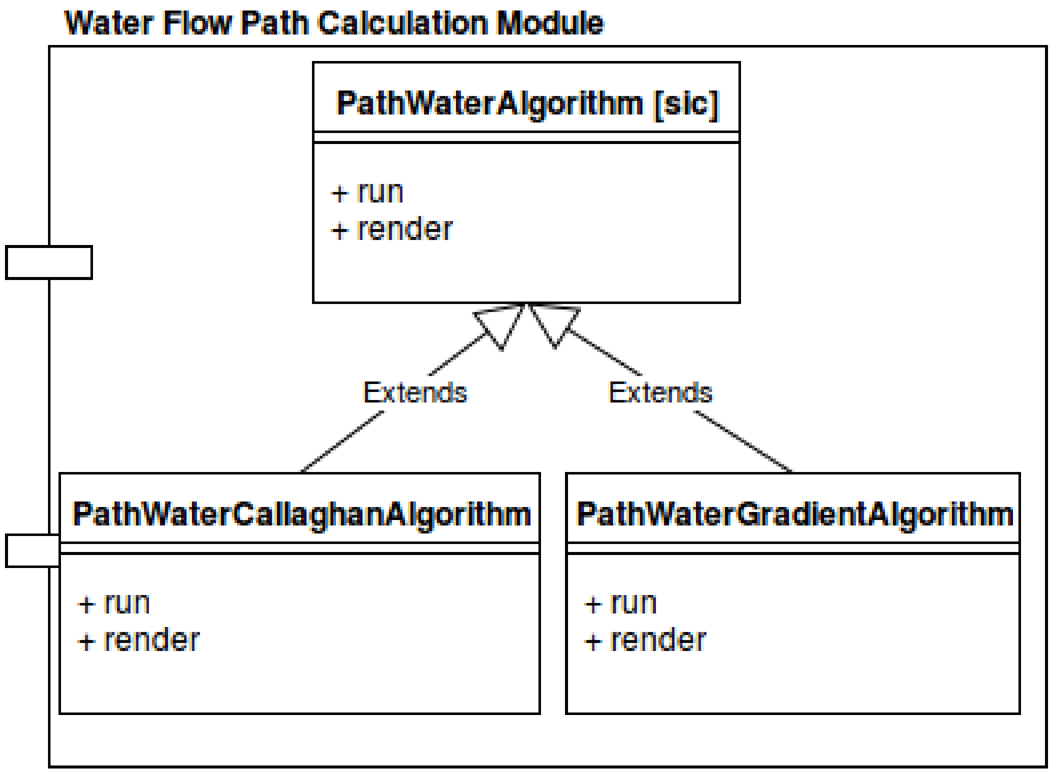

Water Flow Path Calculation Module
PathWaterAlgorithm [sic]
+ run
+ render
Extends
Extends
PathWaterCallaghanAlgorithm
+ run
+ render
PathWaterGradientAlgorithm
+ run
+ render

Figure 9: Water Flow Path Calculation module class diagram. Algorithm classes must inherit from the PathWaterAlgorithm class and implement its virtual functions *run* and *render*.

3.1.3 Model Rendering subsystem

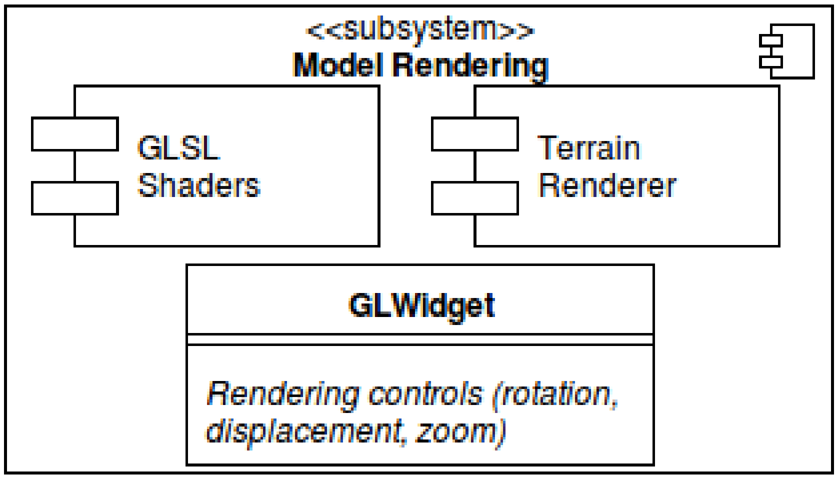


Figure 10: Modules composing the Model Rendering subsystem, along with the GLWidget class that interacts with the User Interface subsystem (not depicted).

The Model Rendering subsystem (Figure 10) is composed of the following elements:

- The GLSL Shaders module is responsible for reading and initializing the actual shaders written in the OpenGL Shading Language, and passing the necessary data for their operation (Figure 11). As in the Model Analysis subsystem, this module is based on the Strategy pattern, with an abstract base class *ShaderUtils* from which a subclass is derived for each specific type of shader required.

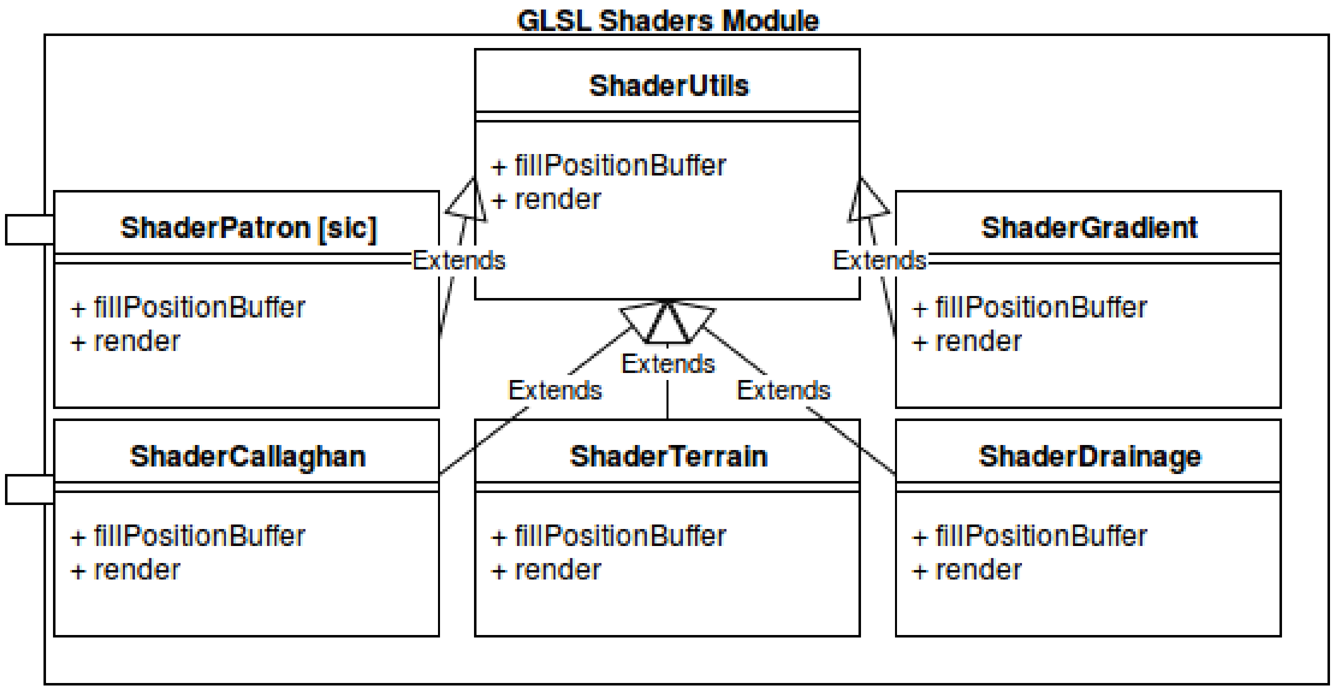


Figure 11: GLSL Shaders module class diagram. Shader classes must inherit from ShaderUtils and implement its virtual functions *fillPositionBuffer* and *render*.

- The Terrain Renderer module (Figure 12) is in charge of instantiating shader classes, and performing the function calls required to render the terrain model, along with any pertinent analysis results that may be currently active. It is composed of a single class *PainterTerrain*.

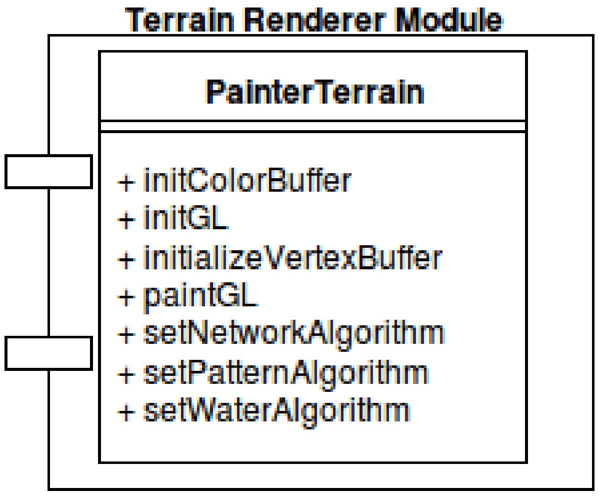


Figure 12: Terrain Renderer module, composed of a single class PainterTerrain in charge of performing the necessary invocations to render both the model and any analysis results to be overlaid on the terrain.

- The GLWidget class, which provides a rendering canvas to be integrated into the user interface. It receives rendering control input from the User Interface subsystem and performs the necessary adjustments in order to rotate, pan or zoom into the rendered mesh.

3.2 Detected issues

Through a preliminary review of the Runnel software, we identified the following major issues.

3.2.1 Drainage network extraction

Poor efficacy in comparison to RiverTools When comparing results given by the Peucker and O'Callaghan drainage algorithms with those provided by RiverTools over the same DEM, we observe significantly different drainage networks. We are unable to establish a clear visual correspondence between the networks identified by RiverTools and the results provided by Runnel.

In the case of the Peucker algorithm depicted in Figure 13, the streams detected in the highest areas of the terrain (right side), where slopes are more pronounced, tend to coincide to some degree (although the Runnel results do not perform order pruning and are therefore visually contaminated with order 1 streams). However, as one approaches the regions of lower relief (middle to left side), results become poor and cluttered with disconnected points, showing little correspondence with the results provided by RiverTools.

With respect to the O'Callaghan algorithm results shown in Figure 14, we observe a very similar situation, although the disconnected points occasionally tend to follow the direction of streams present in the RiverTools results.

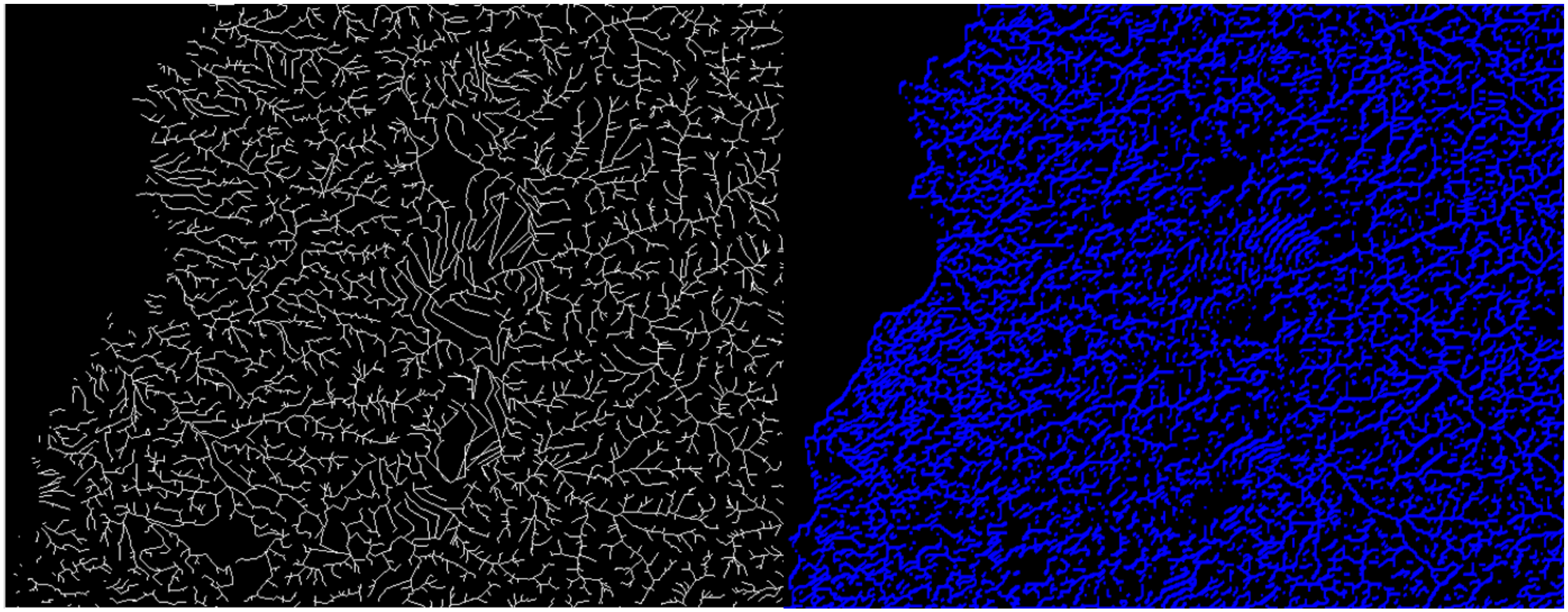


Figure 13: Comparison of results obtained through RiverTools (left), and the Peucker algorithm available in the base version of Runnel (right).

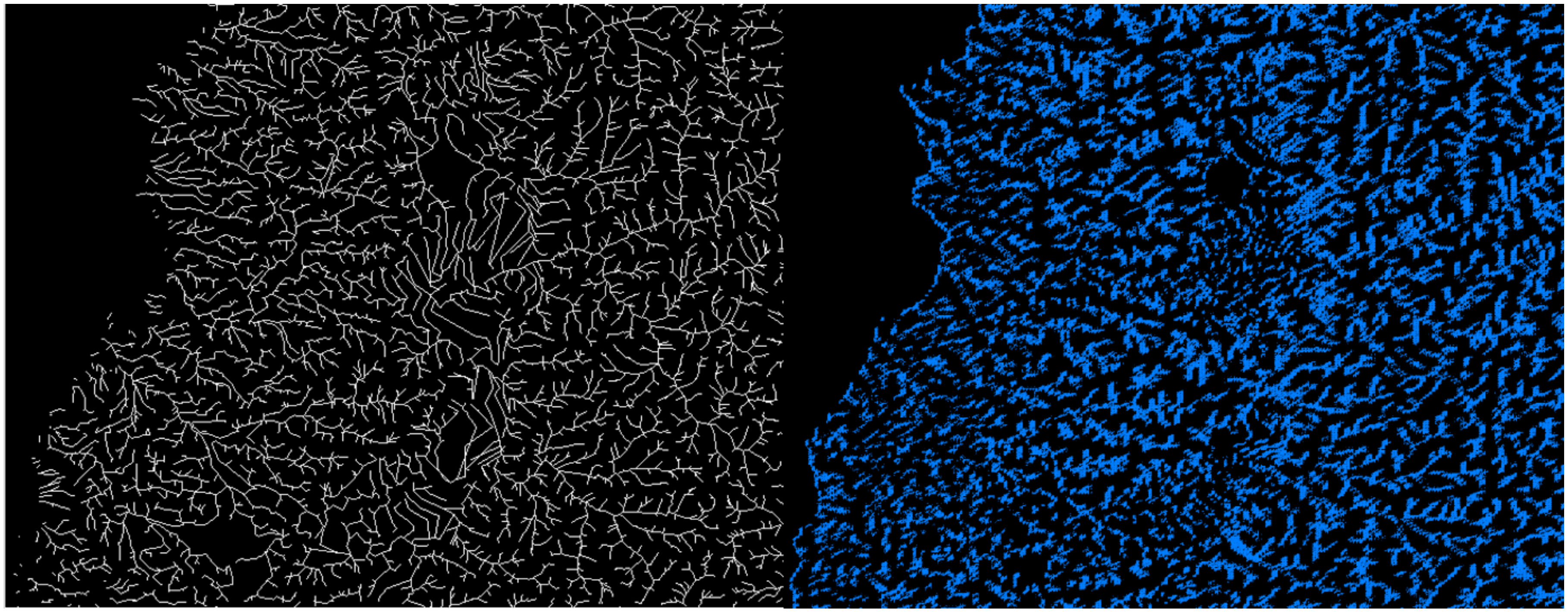


Figure 14: Comparison of results obtained through RiverTools (left), and the O'Callaghan algorithm available in the base version of Runnel (right).

We consider RiverTools, as a established commercial tool and as the software sought to be replaced with Runnel, to provide relatively accurate results that will serve as our baseline in this work. Hence, we believe Runnel to be significantly flawed in this regard.

Strahler order calculation defect The software does not correctly compute Strahler order numbers for extracted drainage networks. Through comparison with RiverTools results (Figure 15), we find ourselves unable to establish a visual correspondence between major streams according to Strahler order. In results provided by Runnel, stream orders rarely appear to have a hierarchical relation consistent with the downstream direction, and we observe high order streams that do not clearly flow towards an outlet, but rather traverse most of the terrain with no apparent reason. We consider these results to be erroneous and of little practical use.

Absence of preprocessing techniques The software does not provide any preprocessing techniques to resolve pits or flats in the input DEM. As described in Chapter 2, the presence of these unresolved elements can severely hamper the proper functioning of drainage analysis algorithms, especially those that employ steepest descent strategies (affected by pits) or local neighborhood-based flow routing (affected by flats). The availability of tools for resolving these elements is very important to obtain accurate analysis results.

Peucker algorithm implementation defect A software defect was detected in the Peucker algorithm implementation. According to the Peucker algorithm, for every 2x2 square in the grid, we determine the highest elevation value and flag all of the points with that elevation, with the remaining non-flagged points becoming part of the drainage network. In the Runnel implementation, however, we find that only one point is flagged in each square, with the flagged point being arbitrarily chosen as the first point in processing order. This causes the resulting drainage network to have additional spurious points that may alter the results of subsequent drainage analysis.

3.2.2 Drainage pattern classification

Poor network coverage When executing the Zhang-Guilbert algorithm over an extracted drainage network, we observe that most of the network remains unclassified, with positive results only for very specific areas (Figure 16). This poses the following question: is this coverage problem mostly attributable to an incorrect implementation of the Zhang-Guilbert algorithm, to the input drainage network being too flawed to allow for adequate pattern classification, or to limitations of the Zhang-Guilbert algorithm itself?

3.2.3 Water flow path calculation

Flow ends at local minima The water flow path algorithm implemented in Runnel is a greedy algorithm based on a steepest descent flow assignment strategy. Given that the software does not provide preprocessing techniques to resolve pits and flats in the input DEM, we find that this

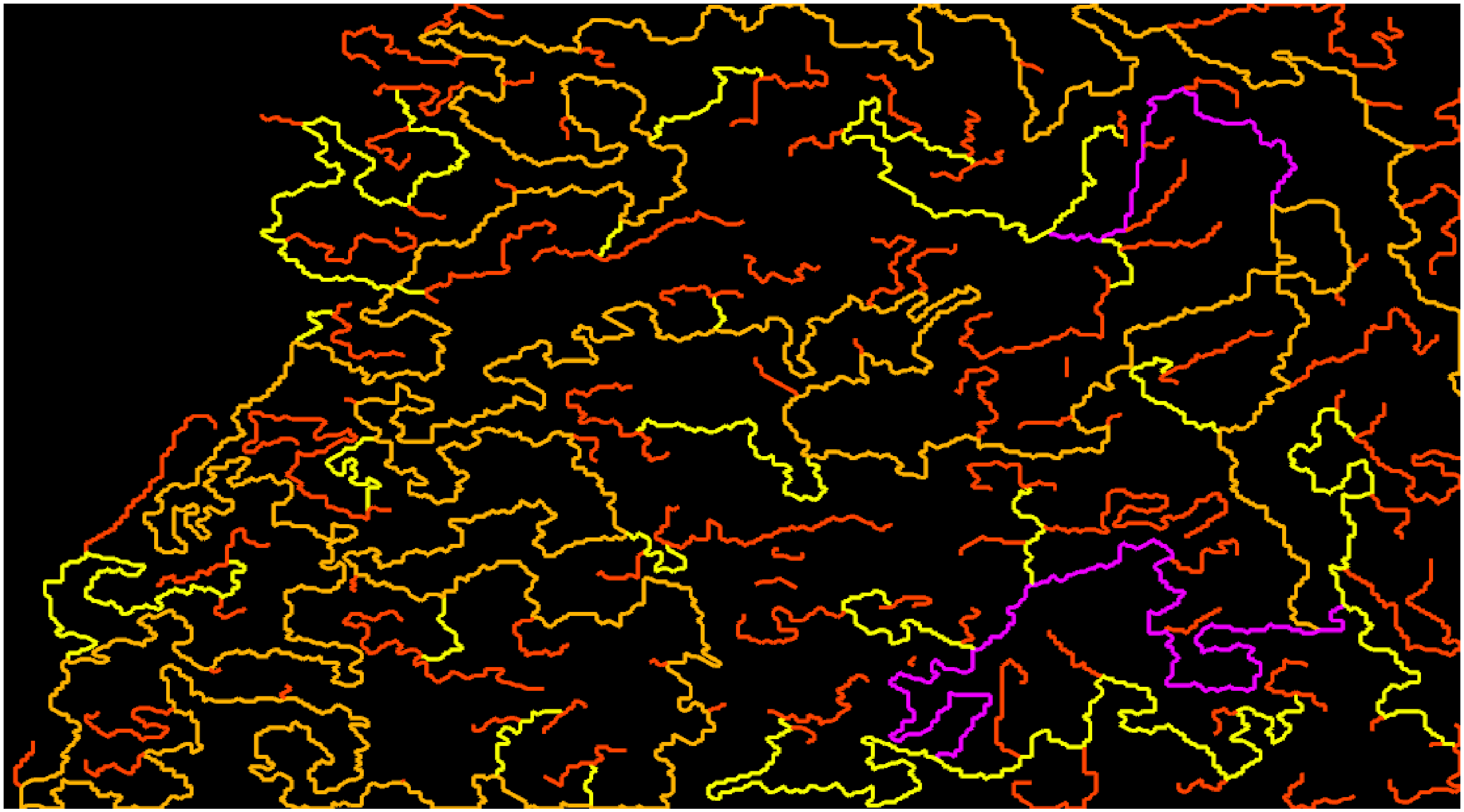


Figure 15: Results of Strahler order calculation performed in Runnel on a Peucker-based drainage network. This image depicts all network streams with Strahler order 3 or higher. Strahler orders are color-coded: red for order 3, yellow for order 4, orange for order 5 and purple for order 6.

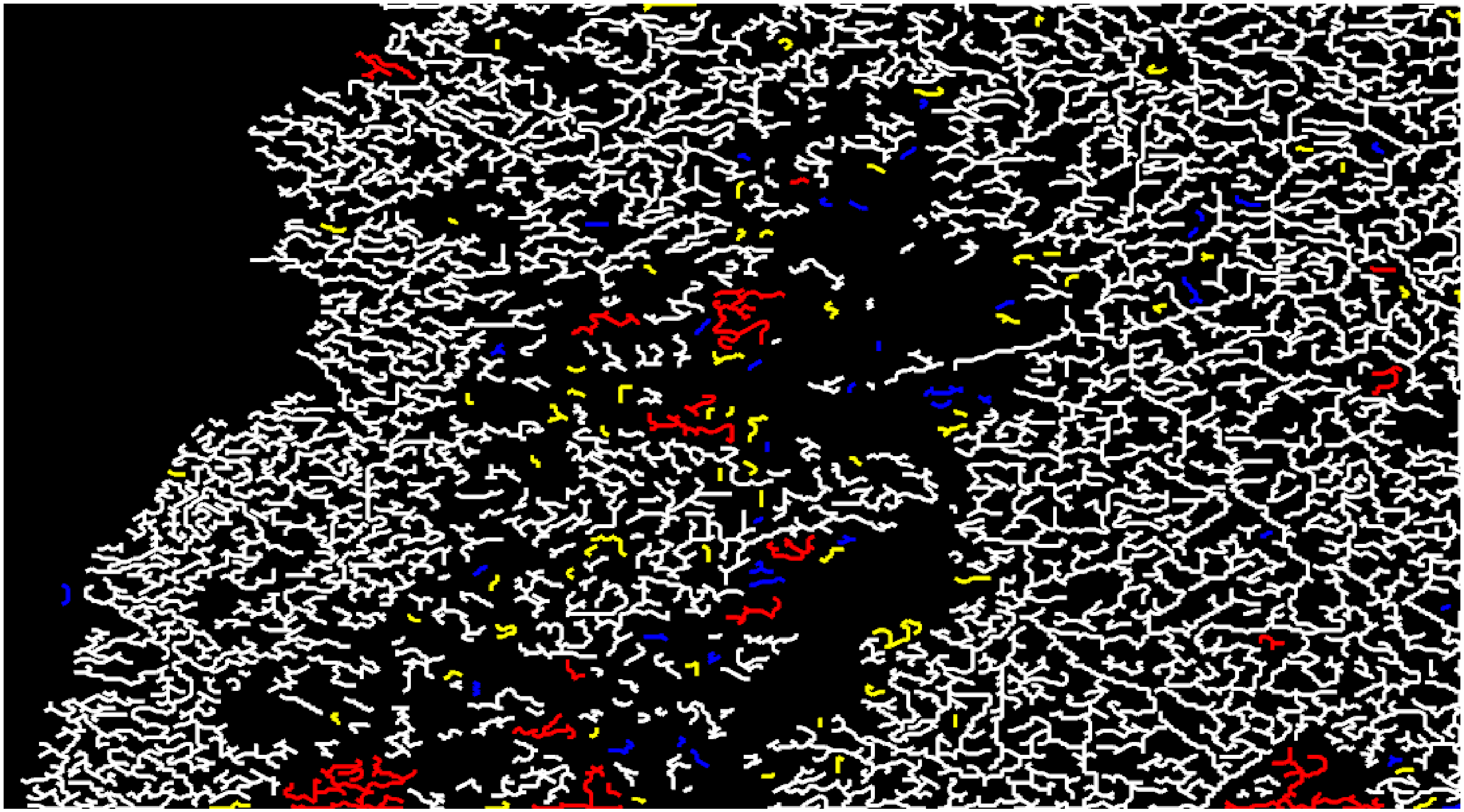


Figure 16: Drainage pattern classification results obtained from the Zhang-Guilbert algorithm in Runnel. Subnetworks are color-coded according to the identified pattern: yellow for dendritic, blue for parallel, green for trellis and red for rectangular, with white for subnetworks that remain unclassified.

algorithm prematurely halts execution at local minima, impeding the calculation of a complete flow path towards a basin outlet. Hence, the results provided by this tool are very limited in their usefulness.

3.2.4 Data input

Elevation data limited to unsigned byte type Runnel supports DEM files based on the GeoTIFF format under WGS84 coordinates. Datasets with these properties are readily available from the SRTM project for the region of study described in the Methodology section. The elevation data provided by the SRTM datasets is stored in a 16-bit data type, which is sufficiently large to allow for expressing elevation values in meters. However, the Runnel software only supports input elevation data provided in the 8-bit unsigned byte type (i.e. constrained to the 0-255 range), forcing one to convert the input 16-bit data into this type, resulting in a significant loss of vertical data precision.

3.3 Other desirable techniques to be implemented

Support for Strahler order calculation in networks with braided streams Notwithstanding the defects described concerning Strahler order calculation in Runnel, as per our discussion of braided streams in Chapter 2 we consider it desirable to implement an adequate version of the Strahler order calculation algorithm that may prevent improper order increments where braided streams are present.

Tool for terrace visualization As mentioned in Chapter 2, it is of interest for fluvial geomorphologists to distinguish geological terraces within the region of study. Currently, the Runnel software lacks any tools to assist in this endeavour. Thus, we seek to implement a basic tool that may allow researchers to detect and visualize terraces in the 3D terrain mesh.

3.4 Proposed solution

With respect to drainage network extraction, besides correcting the defect spotted in the Peucker algorithm implementation, we also consider that the Peucker and O'Callaghan algorithms are in a sense outdated, for newer algorithms have also incorporated newer key observations of drainage network analysis in their design. Hence, we propose implementing the RWFlood algorithm in order to compare its results against those generated through RiverTools. We select RWFlood over other more widespread algorithms in virtue of its recency and its reported efficiency in comparison to the mainstream approaches (e.g. Ehlschlaeger-Metz in GRASS GIS).

Regarding Strahler order calculation, we propose fully reimplementing the current scheme according to the Gleyzer algorithm described in Chapter 2, due to its support for networks containing braided streams.

On the subject of preprocessing techniques, we consider that their implementation may have a significant, beneficial effect on the results obtained by the currently implemented water flow path calculation tool, as well as potentially on drainage network extraction results. Hence, we propose implementing the Garbrecht-Martz depression breaching and flat resolving algorithms described in Chapter 2, given their widespread usage in this field, in order to study their improvement on network extraction results. As we shall only utilize voidless DEMs in our work, we do not implement void filling techniques at this point.

As for drainage pattern classification, we decide to avoid addressing this subject further at this stage in our work. Our rationale for this decision is that a functional and accurate pattern classification tool is of little use if the drainage network itself is severely flawed, as none of the resulting patterns may be truly observable in reality, and hence will provide no useful information to the researcher. Obtaining solid results in drainage network extraction is, therefore, a prerequisite that must be fulfilled before we can sensibly approach the subject of pattern recognition. Thus, we shall only consider drainage pattern classification as a potential future avenue of research.

Concerning data input, we propose adapting the input system to handle the 16-bit elevation data type in replacement of the current 8-bit unsigned byte type. To the author's knowledge, no active and publicly accessible mass data acquisition projects release their elevation data in the currently supported data type, and the added precision of handling elevation data directly in meters is more relevant to the usefulness of Runnel than the minor savings in memory usage from utilizing a smaller data type. However, we regard the implementation of effective drainage network extraction and terrain preprocessing algorithms as a task of higher priority, as the flaws in the existing software prevent us from making effective use of a 16-bit data type. Thus, we shall only consider this adaptation as future work.

Finally, for the purpose of terrace visualization, we propose a simple interactive tool that, given a user-selected triangle in the mesh and an angle threshold, will highlight all parts of the terrain whose normal vector has a roughly similar direction within the given threshold.

4 Design and implementation

This chapter is divided into two main sections. In the first section, we present pseudocode for each of the algorithms to be implemented in our work. In the second section, we describe certain details of the actual C++ implementation of these algorithms in Runnel, based on the existing software design. The referenced source code listings are shown in Appendix A.

4.1 Algorithm descriptions

4.1.1 Peucker algorithm

The Peucker algorithm (Algorithm 1) is a drainage network extraction algorithm. Given the matrix of DEM nodes, a window of 2x2 nodes is moved sequentially throughout the terrain. The maximum elevation of all nodes within the window is calculated, and all nodes that share said elevation value are flagged, which indicates they do not belong to the drainage network. After processing the entire DEM, the drainage network is defined as the collection of all unflagged nodes.

Algorithm 1: Peucker algorithm for drainage network extraction

```
Input: DEM node matrix
Result: Nodes not belonging to the drainage network are flagged
foreach matrixNode do matrixNode.peuckerFlag ← false;
foreach 2x2 square region in matrix do
    Define m as the highest elevation among all four nodes;
    foreach squareNode do
        if $squareNode.elevation = m$ then squareNode.peuckerFlag ← true;
```

Example Figure 17 shows a step-by-step diagram of its application to a 5x5 elevation grid.

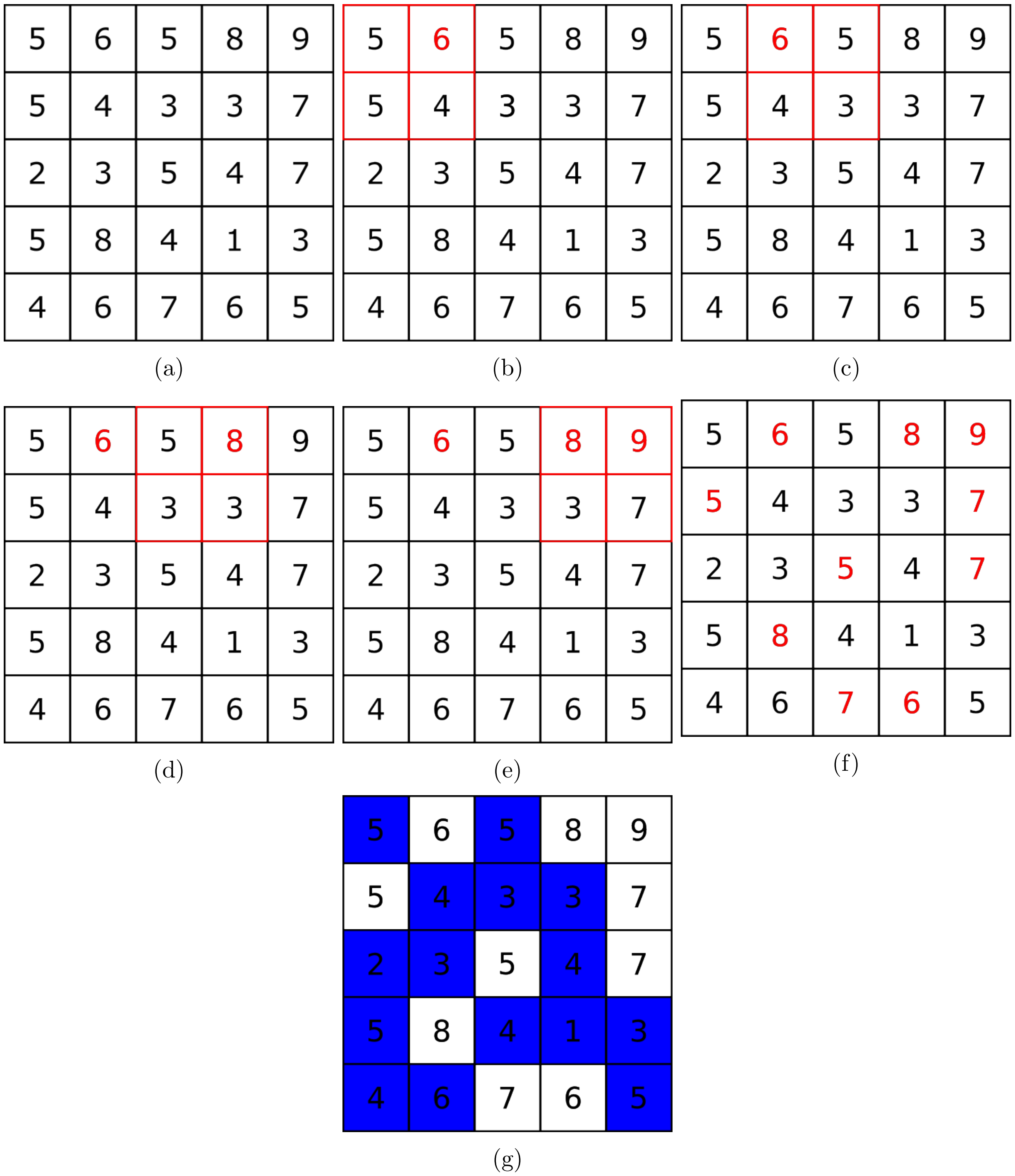


Figure 17: Peucker algorithm example. (a) Base grid with elevation data. (b) The first 2x2 square is processed, flagging the highest elevation (red). (c) The next 2x2 square is processed; the highest point is already flagged. (d) The next square is processed and the 8 is flagged. (e) The next square is processed; the 9 is flagged, even when another node is already flagged within the square. (f) All 2x2 squares have been processed, and flagged nodes are marked in red. (g) The drainage network is composed of all nodes that have not been flagged.

4.1.2 RWFlood algorithm

The RWFlood algorithm is a drainage network extraction algorithm, divided into two subroutines executed sequentially. In simple terms, the RWFlood algorithm constructs a drainage network in a bottom-up manner beginning from outlets in the DEM boundary, as if the terrain were to be progressively flooded by a rising water level.

The first subroutine (Algorithm 2) assigns flow directions to all nodes within the terrain. It receives the matrix of DEM nodes as its input. An array of node queues is created, with one queue for each (integer) elevation value present in the node matrix, and all nodes within the boundary of the matrix are inserted into the queue that corresponds to their respective elevations. Flow directions for all nodes on the terrain boundary are defined to point away from said boundary.

The flood subroutine then iterates over all elevation values in ascending order. For each elevation, the corresponding node queue is processed. Nodes contained within the queue are removed from it, and for each such queue node, all of its neighbors with unassigned flow directions are assigned to flow towards the queue node. If any of these neighbors are also located at a lower elevation than the queue node, their elevation is increased to match the elevation of the queue node. After completing these steps, the neighbor is then inserted into the queue corresponding to its elevation value. The process repeats until the node queue is empty. After a given queue is empty, it is guaranteed that no other node will be inserted into it, as any other insertions will necessarily occur on queues representing higher elevation values. The end result of this subroutine is the assignment of flow directions for all DEM nodes within the matrix.

Algorithm 2: RWFlood algorithm for flow direction assignment

```
Input: DEM node matrix
Output: DEM node matrix with flow directions assigned between nodes
Define Q[minElev, ..., maxElev] as an array of queues;
foreach node in matrix do
    node.flowDirection ← NULL;
foreach node in matrix boundary do
    Q[node.elev].insert(node);
    node.flowDirection ← OutsideTerrain;
for z ← minElev to maxElev do
    while Q[z] is not empty do
        node ← Q[z].remove();
        foreach neighbor of node where neighbor.flowDirection = NULL do
            neighbor.flowDirection ← node;
            if neighbor.elev < z then
                neighbor.elev ← z;
            Q[neighbor.elev].insert(neighbor);
```

The second subroutine (Algorithm 3) calculates water level values for terrain nodes by accumulating water values along flow paths. It receives the matrix of DEM nodes, with flow directions assigned according to the flood subroutine, as its input. For all nodes within the matrix, a base water value of 1 and an in-degree of 0 is assigned, and all nodes are flagged as unvisited. All DEM nodes are then processed once, without altering the unvisited flag, adding 1 to the in-degree of the neighbor pointed to by the flow direction assigned to the node.

Afterwards, the subroutine iterates over all unvisited nodes with an in-degree of 0. Each node is flagged as visited, and if not located on the terrain boundary, its water level value is added to the neighbor its flow direction points to, and the in-degree of said neighbor is decreased by 1. The process then repeats at said neighbor (depth-first). In this manner, nodes are only flagged as visited, and unfit for further processing, when their in-degree reaches zero, and therefore when their water value has been increased by as many nodes as indicated by their initial in-degree. The end result of this subroutine is the calculation of water levels throughout the DEM as determined by the flow directions of its nodes.

Algorithm 3: RWFlood algorithm for flow accumulation calculation

```
Input: DEM node matrix with flow directions assigned between nodes
Output: DEM node matrix with flow directions and flow accumulation values
Define next(node) as the neighbor that follows node in the flow path according to
  node.flowDirection;
foreach node in matrix do
    InDegree[node] ← 0;
    Flow[node] ← 1;
    Visited[node] ← false;
foreach node in matrix do
    InDegree[next(node)] ← InDegree[next(node)] + 1;
foreach node in matrix where Visited[node] = false do
    while InDegree[node] = 0 do
        Visited[node] ← true;
        if next(node) is outside the terrain boundary then
            break while;
        Flow[next(node)] ← Flow[next(node)] + Flow[node];
        InDegree[next(node)] ← InDegree[next(node)] - 1;
        node ← next(node);
```

Example Figures 18 (first subroutine) and 19 (second subroutine) show a step-by-step diagram of its application to a 5x5 elevation grid.

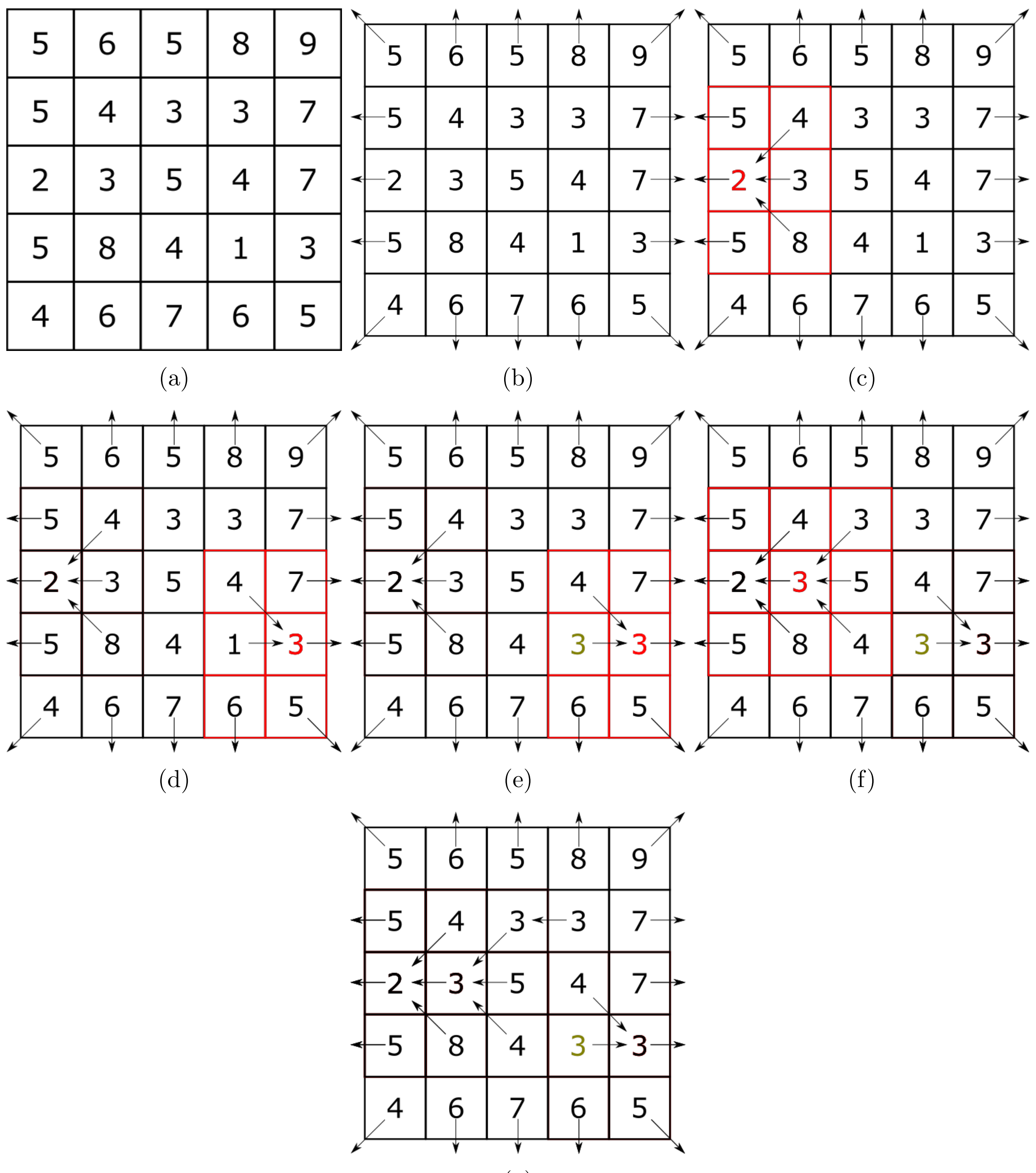


Figure 18: RWFlood application example. (a) Base grid with elevation data. (b) Outbound flow directions are assigned to boundary nodes. (c) The lowest boundary node (red) is processed; all neighbors without direction are made to point to it, and enqueued for their respective elevation values. (d) The next lowest boundary node (first element in the 3 queue) is processed. (e) As one of its neighbors is at a lower elevation, the neighbor is raised to 3. (f) The next 3 in the queue is processed. (g) Flow directions after all nodes are processed.

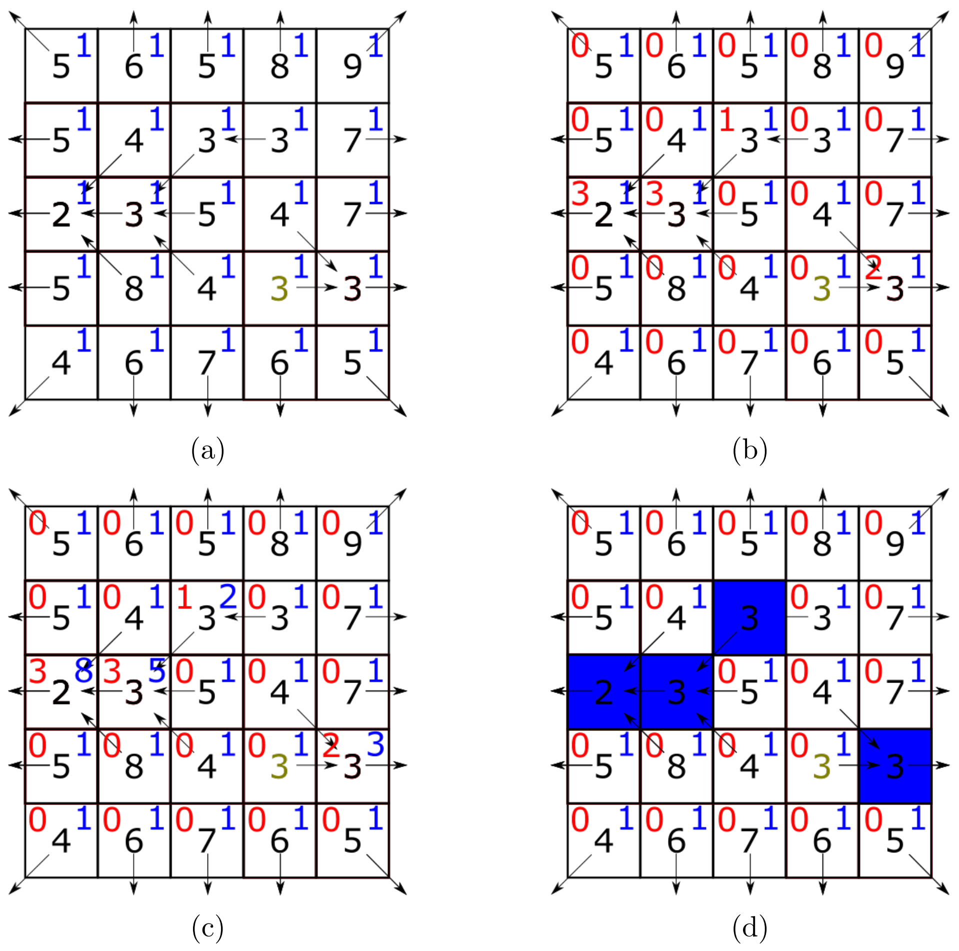


Figure 19: RWFlood application example, second subroutine (water level accumulation). (a) A water level (blue) is initialized for all nodes. (b) An in-degree value (red) is computed for all nodes. (c) Water values are added cumulatively along flow paths. (d) We highlight all nodes with water levels equal to or above 2 as part of the drainage network.

4.1.3 Gleyzer algorithm

The Gleyzer algorithm is a Strahler order calculation algorithm, divided into two main subroutines. In simple terms, the Gleyzer algorithm considers consecutive nodes of the same Strahler order to be a single stream, and keeps track of the node where said stream began. This allows it to recognize cases where two converging streams of the same order actually originated from the same node, thus avoiding an erroneous increase in Strahler order.

The first subroutine, *MakeDictionaries* (Algorithm 4), is an auxiliary procedure that builds the dictionaries required by the main function. Two dictionaries are created, which identify the two endpoint nodes of each edge within the network, and the list of edges to which each DEM node belongs.

```
Algorithm 4: MakeDictionaries function for Gleyzer algorithm
  Input: Drainage network edges
  Output: NodesPerEdge and EdgesPerNode dictionaries
  foreach edge in network do
      NodesPerEdge[edge.id] ← (edge.sourceNodeId, edge.destinationNodeId);
      EdgesPerNode[edge.sourceNodeId].insert(edge.id);
      EdgesPerNode[edge.destinationNodeId].insert(edge.id);
```

The second subroutine, *StreamOrdering*, is a recursive function that receives the ID of a network edge (*edgeId*), and the ID of the source node of said edge (*sourceNodeId*) as input, and stores the Strahler order of the edge in a global dictionary named *StreamOrders* and returns the Strahler order as its output. First, the function accesses the *EdgesPerNode* dictionary and checks if the source node is only associated to a single edge; if so, the source node of the edge is a drainage source of Strahler order 1, and hence, a value of 1 is assigned to the corresponding entry for *edgeId* in the *StreamOrders* dictionary.

Otherwise, for every edge associated to the source node and different from *edgeId*, the function retrieves the two nodes that form the edge from the NodesPerEdge dictionary, storing them into the *sourceId* and *destinationId* variables. The StreamOrdering function is then called recursively on this new edge, with the following caveat: if the ID of the original source node (sourceNodeId) coincides with the value stored in sourceId, then destinationId is passed to the recursive call. The result of the recursive call is stored in the *UpstreamOrders* dictionary, local to each function call.

Afterwards, the function proceeds to analyze the Strahler orders stored. Two auxiliary variables are initialized with a value of 0: *MaxOrder*, which stores the highest Strahler order encountered among all upstream nodes, and *MaxOrderCount*, which stores how many instances of MaxOrder have been located. These values are updated accordingly as the function iterates over all nodes stored in the UpstreamOrders local dictionary.

Finally, the StreamOrdering function returns a value that depends on MaxOrderCount. If MaxOrderCount is greater than 1, there are multiple (possibly braided) streams that converge into the *edgeId* edge, and therefore, we increase the order by 1 with respect to the MaxOrder value found. Else, we simply return MaxOrder.

Algorithm 5: StreamOrdering recursive function for Gleyzer algorithm

```
Input: edgeId, sourceNodeId
Result: StreamOrders[edgeId] : Strahler order for the network edge identified by edgeId
if $|EdgesPerNode[sourceNodeId]| = 1$ then
    StreamOrders[edgeId] ← 1;
else
    foreach $e \in EdgesPerNode[sourceNodeId]$ do
        if $e.id \neq edgeId$ then
            (sourceId, destinationId) ← NodesPerEdge[e.id];
            if $sourceId \neq sourceNodeId$ then
                UpstreamOrders[e.id] ← StreamOrdering(e.id, sourceId);
            else
                UpstreamOrders[e.id] ← StreamOrdering(e.id, destinationId);
    MaxOrder ← 0;
    MaxOrderCount ← 0;
    foreach $Order \in UpstreamOrders$ do
        if $Order > MaxOrder$ then
            MaxOrder ← Order;
            MaxOrderCount ← 1;
        else
            if $Order = MaxOrder$ then
                MaxOrderCount ← MaxOrderCount + 1;
    if $MaxOrderCount > 1$ then
        StreamOrders[edgeId] ← MaxOrder + 1;
    else
        StreamOrders[edgeId] ← MaxOrder;
```

4.1.4 Garbrecht-Martz flat resolution algorithm

In simple terms, the Garbrecht-Martz flat resolution algorithm evaluates the area surrounding a flat region, and uses this information to construct a microrelief to be superimposed on the flat region, allowing for unambiguous flow direction assignment in a way such that water will tend to descend away from peaks and towards lower zones in the neighborhood of the flat.

The Garbrecht-Martz flat resolution algorithm is divided into three subroutines. The first two subroutines compute the gradients towards lower terrain, and away from higher terrain respectively. The third subroutine combines the results of both subroutines into a single mapping of nodes to elevation increments. An elevation increment is defined to be $\frac{2}{100000} * VerticalResolution$.

The first function (Algorithm 6) takes a list of nodes belonging to flat areas as its input. Three empty node lists are created to track flat area nodes with at least one neighbor at a lower elevation (nodesWithDownslopeGradient), flat area nodes with at least one neighbor at a lower elevation detected during a given iteration (newDownslopeNodes), and flat area nodes that have already been processed in a given iteration (processedFlats). A mapping of nodes to elevation increments is also instantiated, with all increments being initially set to zero.

The main loop operates for as long as the list of flat area nodes (flatNodes) is not empty, with nodes being removed from it as they are added to the three previously mentioned lists. Each iteration scans the flatNodes list, and for each node, all of its eight neighbors are checked. If the central node has higher elevation than a neighbor, or the neighbor itself is contained in the nodesWithDownslopeGradient list (and hence, routing water through said neighbor would lead towards lower terrain), the central node is added to the newDownslopeNodes and processedFlats lists.

Once all nodes in the flatNodes list are scanned in an iteration, the newly detected nodes with downslope are added to the nodesWithDownslopeGradient list, and removed from the list of flat area nodes. Finally, the elevation increment of all nodes that remain in the flatNodes list is increased by 1.

When all nodes are removed from the flatNodes list, the result is a mapping of nodes to elevation increments, representing a gradient towards lower terrain. A higher elevation increment for a node implies that the node is located further away from a region of lower elevation.

Algorithm 6: Function to calculate the gradient towards lower terrain for Garbrecht-Martz imposed gradients algorithm

```
Input: flatNodes : the set of nodes that belong to a flat zone
Result: resultingGradient : the set of (node, totalIncrements) pairs to be applied
Let nodesWithDownslopeGradient, newDownslopeNodes, processedFlats be initially empty
  lists of nodes;
Initialize resultingGradient to a list of $|flatNodes|$ pairs of the form (node, 0), with
  $node \in flatNodes$;
while flatNodes is not empty do
    foreach node in flatNodes do
        foreach neighbor in node.neighbors do
            if nodesWithDownslopeGradient.contains(neighbor) or node.elev > neighbor.elev
              then
                newDownslopeNodes.insert(node);
                processedFlats.insert(node);
    foreach node in newDownslopeNodes do
        nodesWithDownslopeGradient.insert(node);
    newDownslopeNodes.clear();
    foreach node in processedFlats do
        flatNodes.remove(node);
    processedFlats.clear();
    foreach node in flatNodes do
        Add 1 to totalIncrements value for the corresponding entry in resultingGradient;
```

The second function (Algorithm 7) is similar, taking a list of nodes belonging to flat areas as its input, and returning a mapping of nodes to elevation increments that represents a gradient away from higher terrain. Three analogous processing lists are initialized (nodesWithUpslopeGradient, newUpslopeNodes and processedFlats), along with a mapping of nodes to increments with increment values initialized to zero. The most significant difference lies in how increments are applied: whereas the first function applies increments iteratively to all nodes that remain in the flatNodes list, the second function applies them iteratively to a specific class of nodes that are *not* contained in flatNodes.

The main loop operates for as long as the list of flat area nodes (flatNodes) is not empty. Each iteration scans the flatNodes list, and for each node, it checks the elevation of its neighbors. If the central node has at least one neighbor at a lower elevation, then it becomes uneligible for any increments, and is removed from the flatNodes list. Else, if at least one of its neighbors is either at a higher elevation, or contained in the nodesWithUpslopeGradient list, then the central node is added to the newUpslopeNodes and processedFlats lists.

After all nodes in the flatNodes list are processed in an iteration, newly detected nodes with upslope are added to the nodesWithUpslopeGradient list, and removed from the flatNodes list. Afterwards, the elevation increment of all nodes contained in the nodesWithUpslopeGradient (including those collected from previous iterations) is increased by 1. The resulting mapping of nodes to elevation increments represents a gradient away from higher elevations. A higher elevation increment for a node implies that the node is located closer to higher elevation areas.

Algorithm 7: Function to calculate the gradient away from higher terrain for Garbrecht-Martz imposed gradients algorithm

```
Input: flatNodes : the set of nodes that belong to a flat zone
Result: resultingGradient : the set of (node, totalIncrements) pairs to be applied
Let nodesWithUpslopeGradient, newUpslopeNodes, processedFlats be initially empty lists
  of nodes;
Initialize resultingGradient to a list of $|flatNodes|$ pairs of the form (node, 0), with
  $node \in flatNodes$;
while flatNodes is not empty do
    foreach node in flatNodes do
        foreach neighbor in node.neighbors do
            if node.elev > neighbor.elev then
                processedFlats.insert(node);
                Break out of neighbor loop;
            else
                if nodesWithUpslopeGradient.contains(neighbor) or
                  node.elev < neighbor.elev then
                    newUpslopeNodes.insert(node);
                    processedFlats.insert(node);
    foreach node in newUpslopeNodes do
        nodesWithUpslopeGradient.insert(node);
    newUpslopeNodes.clear();
    foreach node in processedFlats do
        flatNodes.remove(node);
    processedFlats.clear();
    foreach node in nodesWithUpslopeGradient do
        Add 1 to totalIncrements value for the corresponding entry in resultingGradient;
```

The third function (Algorithm 8) combines the results of the previous two functions by linearly adding the elevation increments contained in each mapping, producing a single gradient to be applied onto the flat zone. Additionally, if any pair of adjacent nodes is such that the increments produced by both gradients cancel each other out, an extra half-increment is applied to the node. The increments of two adjacent nodes are defined to cancel each other if the increment of the first node given by the gradient towards lower terrain is equal to the increment of the second node in the gradient away from higher terrain, and viceversa.

Algorithm 8: Function to calculate the final imposed gradient for Garbrecht-Martz imposed gradients algorithm

```
Input: gradient1, gradient2 : gradients calculated from the previous functions
Result: imposedGradient : the set of (node, increments) pairs to be applied on the DEM
Merge gradient1 and gradient2 sets into imposedGradient by adding the two totalIncrement
  values for each node;
Let halfIncrementNodes be an initially empty list of nodes;
foreach (node, mergedIncrement) in imposedGradient do
    foreach neighbor in node.neighbors do
        if $gradient1[node].value = gradient2[neighbor].value$ and
          $gradient2[node].value = gradient1[neighbor].value$ and
          !halfIncrementIds.contains(neighbor) then
            halfIncrementIds.insert(node);
foreach node in halfIncrementIds do
    Add a half increment to the mergedIncrement value for the corresponding entry in
      imposedGradient;
```

Example Figures 20 (first function) and 21 (second function) show the construction of the two gradients on a 7x7 terrain with a 5x5 flat zone. Figure 22 (third function) shows the combined gradient and final flow direction assignment for the flat zone.

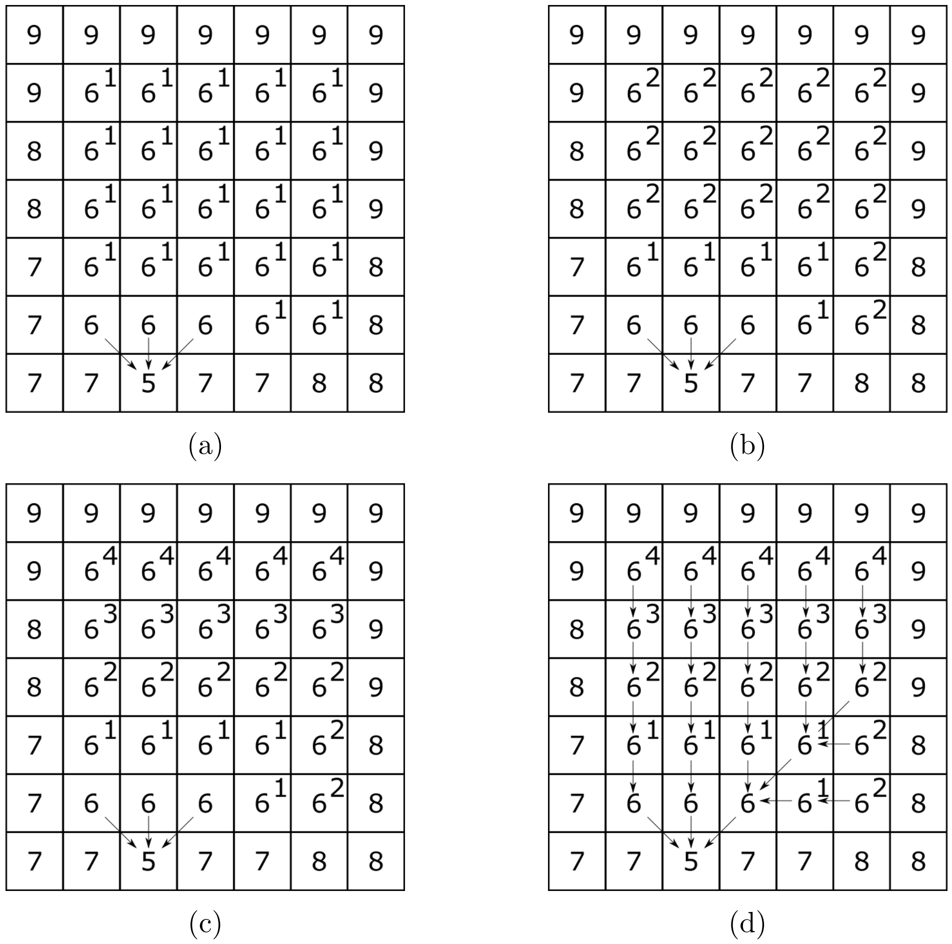


Figure 20: Garbrecht-Martz imposed gradients construction example, first function (gradient towards lower terrain). The central numbers indicate elevation. The number in the top right corner represents the amount of increments to be applied to the node. (a) A single increment is added for all nodes belonging to the flat zone that do not have a drainage direction. Nodes adjacent to drainage-possessing nodes are added to a list and excluded from further processing. (b) An extra increment is added for all non-excluded nodes; neighbors of excluded nodes are also added to the exclusion list. (c) Final result of repeating this process until all flat nodes without direction are added to the list. (d) Drainage directions that would result from applying the constructed gradient on the terrain.

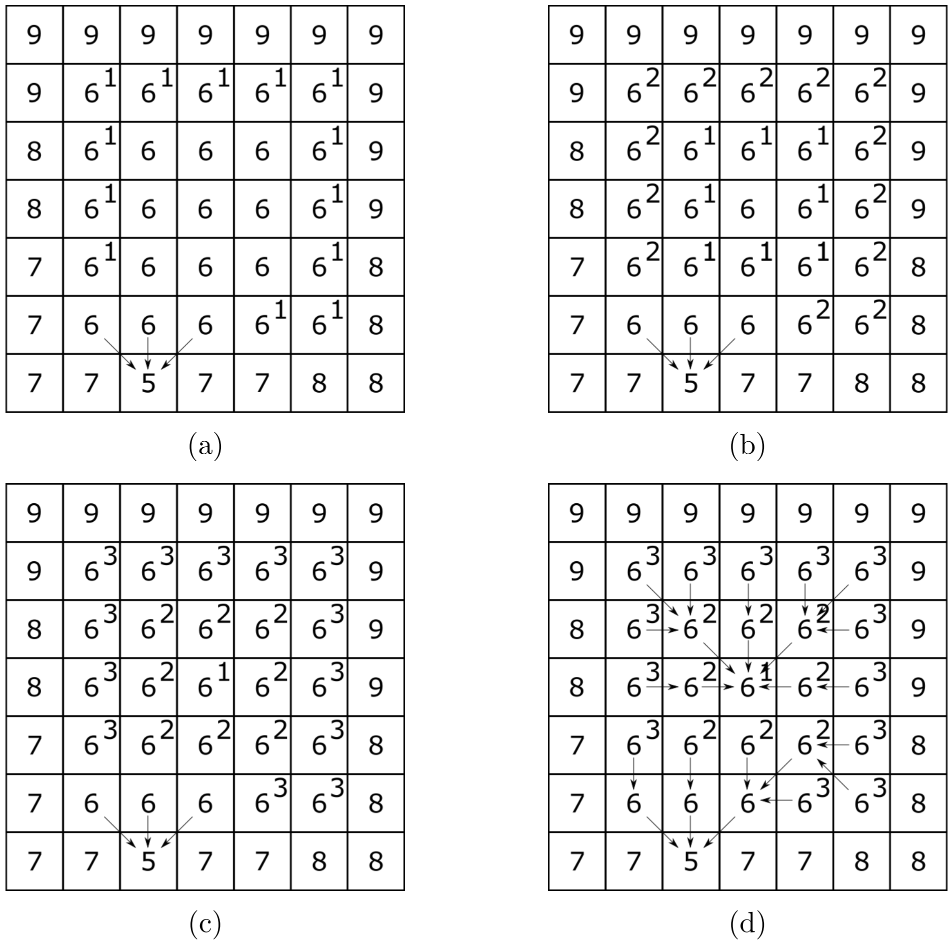
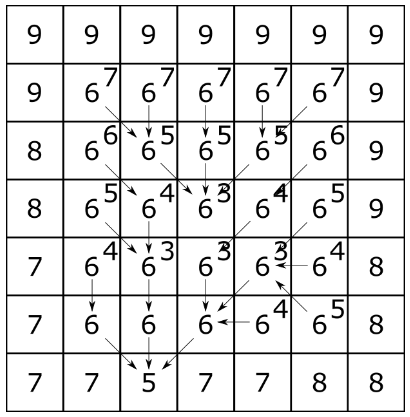
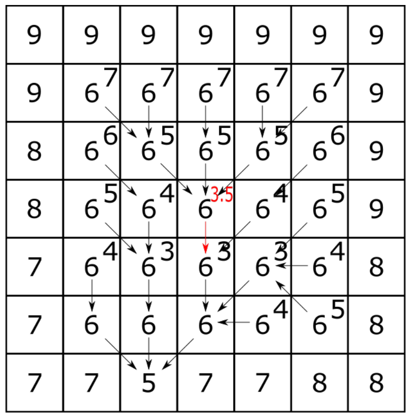


Figure 21: Garbrecht-Martz imposed gradients construction example, second function (gradient away from higher terrain). The central numbers indicate elevation. The number in the top right corner represents the amount of increments to be applied to the node. (a) All nodes belonging to the flat zone that do not have a drainage direction, and are adjacent to a higher node, are added to a list. All list elements are then incremented by 1. (b) Neighbors of list-belonging nodes that belong to the flat zone, and have no flow direction, are added to the list; all elements in the list are then incremented by 1. (c) Final result of repeating this process until all flat zone nodes without direction are added to the list. (d) Drainage directions that would result from applying the constructed gradient on the terrain.



(a)



(b)

Figure 22: Garbrecht-Martz imposed gradients construction example, third function (combined gradient). The central numbers indicate elevation. The number in the top right corner represents the amount of increments to be applied to the node. (a) Increments from both constructed gradients are added together, and flow directions are assigned according to a steepest descent strategy. (b) A flat region remains from the central node downward, as the increments provided by both gradients have canceled each other out in this area. Hence, an additional half increment is imposed on the central node, allowing us to assign a flow direction for it.

4.1.5 Garbrecht-Martz pit removal algorithm

The Garbrecht-Martz pit removal and depression breaching algorithm evaluates the surroundings of a closed depression, and determines whether an outlet for the depression can be created by breaching up to two nodes in the edges of the depression. If an outlet cannot be created in this manner, then the entire depression is simply filled up to the height of the lowest node in the neighborhood, creating a flat zone that may be resolved by the imposed gradients algorithm.

The Garbrecht-Martz pit removal and depression breaching algorithm, as implemented in our work, is divided into one main function (Algorithm 9) and seven auxiliary procedures (Algorithms 10 to 16) designed to simplify the main function.

The main function (Algorithm 9) receives the matrix of DEM nodes as its input, and produces a modified matrix with elevation values altered to remove closed depressions. First, an *inflowSinks* list is built, containing all nodes in the input matrix that have 1 or more neighbors at a higher elevation *and* exactly 0 at a lower elevation (sinks). A flag is associated to every element of the list, representing whether the sink has already been processed or not.

For each sink in inflowSinks, we first check if the sink has been flagged as processed; if so, we skip it and continue with the next sink. We initialize a *windowSize* value at 5, representing the width (in nodes) of the potential contributing area around an inflow sink. The contributing area is defined as the collection of all contiguous nodes around the inflow sink from where water may flow towards the sink along a path of downslopes and flats. This contributing area is calculated over a window centered on the inflow sink, with window width and height equal to windowSize.

Afterwards, we compute all potential outlets among the nodes belonging to the contributing area. A potential outlet is defined as a node within the contributing area that is adjacent to at least one node located both: 1) outside the contributing area, and 2) at a lower elevation.

If no potential outlets were found, we increase the value of windowSize by 2, and repeat from the contributing area calculation step onward. Otherwise, among the potential outlets computed, we select the outlet located at the lowest elevation (if multiple share the same value, the node with the steepest slope out of the contributing area is picked).

We then check the nodes located on the boundaries of the window. If there is at least one node on the window boundary with lower elevation than the lowest potential outlet, we also increase the value of windowSize by 2 and repeat from the contributing area calculation step onward.

If no such node is found, we also check the contributing area for nodes with elevations lower than the lowest potential outlet. If there is no such node, then all contributing area nodes with elevation equal to the lowest outlet collectively form a flat area, being ineligible for the pit breaching and filling algorithm; thus, we mark these nodes as processed and continue with the next sink in the inflowSinks list.

Otherwise, the contributing area does contain a closed depression, and therefore we proceed by calculating the potential breaching sites located within the contributing area. A contributing area node is defined as a potential breaching site if it satisfies all of the following conditions: 1) the node is located at the same elevation as the lowest potential outlet; 2) the node is adjacent to a node outside the contributing area and at a lower elevation; 3) the node is within 2 nodes (based on chessboard/Chebyshev distance, as shown in Algorithm 13) of a node located inside the contributing area and at a lower elevation.

Afterwards, if at least one potential breaching site was found, we select one. If more than one potential breaching site was detected, we select one according to the following two criteria in decreasing priority: steepest slope between the breaching site and a neighbor outside the contributing area, and steepest slope between the breaching site and a neighbor inside the contributing area. If there is more than one breaching site that satisfies both criteria, we arbitrarily select the first site detected. If no breaching sites were found, we skip to the final step.

Once a site is selected, we perform the breaching procedure by lowering the elevation of the selected site to the elevation of its neighbor outside the contributing area with which it forms the steepest slope. If the breaching site was found to be 2 nodes apart from the closest contributing area node with lower elevation, we also lower the elevation of the next node in said direction.

Finally, the elevation of all nodes within the contributing area at a lower elevation than the modified breaching site (or the lowest potential outlet, if no breaching sites were found) is increased to match it, and the nodes in the resulting flat area are flagged as processed, repeating the process for the next sink in the inflowSinks list.

Algorithm 9: Garbrecht-Martz depression breaching & filling algorithm

```
Input: DEM node matrix
Result: DEM node matrix modified to remove closed depressions through breaching or
        filling
Let inflowSinks be the list of nodes with 1 or more neighbors at higher elevation and 0 at
  lower elevation;
foreach inflowSink in inflowSinks do
    if inflowSink.processed = true then
        Continue inflowSinks loop;
    Let windowSize be an odd positive integer, initialized to 5;
    label ContributingAreaCalculation:
    contributingArea ← ComputeContributingArea(DEM, inflowSink, windowSize);
    potentialOutlets ← ComputePotentialOutlets(contributingArea);
    if potentialOutlets.empty then
        windowSize ← windowSize + 2;
        goto ContributingAreaCalculation;
    Let lowestOutlet be the node in potentialOutlets with minimal elevation (and with
      steepest slope out of the contributing area, if multiple nodes share the same elevation);
    Let windowBoundaryNodes be the list of nodes in windowNodes that compose the
      window's edges;
    if $\exists$ node $\in$ windowBoundaryNodes : node.elevation $<$ lowestOutlet.elevation then
        windowSize ← windowSize + 2;
        goto ContributingAreaCalculation;
    if $\nexists$ node $\in$ contributingArea : node.elevation $<$ lowestOutlet.elevation then
        foreach node in contributingArea where node.elevation = lowestOutlet.elevation do
            node.processed ← true;
        Continue inflowSinks loop;
    else
        potentialBreachingSites ← ComputePotentialBreachingSites(contributingArea,
          lowestOutlet);
    if !potentialBreachingSites.empty then
        selectedBreachingSite ← SelectBreachingSite(potentialBreachingSites,
          contributingArea);
        PerformBreaching(selectedBreachingSite, contributingArea);
    PerformFilling(contributingArea, lowestOutlet);
```

Algorithm 10: ComputeContributingArea function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: DEM, inflowSink node, windowSize
Result: contributingArea : list of nodes whose steepest descent flow converges to inflowSink
Let windowNodes be the list of nodes contained within a $windowSize * windowSize$
  window centered on inflowSink;
Let contributingArea be an initially empty list of nodes;
Let flaggedNodes be an initially empty list of nodes;
Flag inflowSink;
Add inflowSink to flaggedNodes;
foreach node in flaggedNodes do
    foreach neighbor in node.neighbors do
        if $neighbor.elevation \geq node.elevation$ and windowNodes.contains(neighbor) then
            Flag neighbor;
            flaggedNodes.insert(neighbor);
    flaggedNodes.remove(node);
    contributingArea.insert(node);
```

Algorithm 11: ComputePotentialOutlets function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: contributingArea
Result: potentialOutlets : list of nodes that are potential exit points for water flow within
        the contributing area
Let potentialOutlets be an initially empty list of nodes;
foreach node in contributingArea do
    if $\exists\ neighbor \in node.neighbors :$
     $!contributingArea.contains(neighbor) \wedge node.elevation > neighbor.elevation$ then
        potentialOutlets.insert(node);
```

Algorithm 12: ComputePotentialBreachingSites function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: contributingArea, lowestOutlet
Result: potentialBreachingSites : a list of candidate nodes on which to perform the
        depression breaching procedure
maxBreachingLength ← 2;
Let potentialBreachingSites be an initially empty list of nodes;
Let lowerContributors be a list of nodes in contributingArea with elevation ¡
  lowestOutlet.elevation;
foreach node in contributingArea where $node.elevation = lowestOutlet.elevation$ do
    foreach neighbor in node.neighbors where $neighbor \notin contributingArea \wedge$
      $neighbor.elevation$ ¡ $lowestOutlet.elevation$ do
        if $\exists\ contributor \in lowerContributors :$
          $ChebyshevDistance(neighbor, contributor) \leq maxBreachingLength$ then
            Add neighbor to potentialBreachingSites;
            neighbor.breachingDistance ← ChebyshevDistance(neighbor, contributor);
```

Algorithm 13: ChebyshevDistance function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: node1, node2
Result: distance : non-negative integer representing the Chebyshev/chessboard distance
        between the input nodes
diffX ← $|node1.row - node2.row|$;
diffY ← $|node1.column - node2.column|$;
distance ← max(diffX, diffY);
```

Algorithm 14: PerformFilling function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: contributingArea, lowestOutlet
Result: Depressions within the contributing area are filled up to the lowest outlet
foreach node in contributingArea where node.elevation ¡ lowestOutlet.elevation do
    node.elevation ← lowestOutlet.elevation;
    node.processed ← true;
```

Algorithm 15: SelectBreachingSite function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: potentialBreachingSites, contributingArea
Result: selectedBreachingSite
Let selectedBreachingSite be an initially null node;
if $|potentialBreachingSites| > 1$ then
    selectedBreachingSite ← node in potentialBreachingSites with max slope away from
      contributingArea, or max slope into contributingArea if multiple nodes are applicable;
else
    selectedBreachingSite ← potentialBreachingSites.first;
```

Algorithm 16: PerformBreaching function for Garbrecht-Martz depression breaching & filling algorithm

```
Input: selectedBreachingSite, contributingArea
Result: The elevation of the breaching site is lowered to create an outlet for the
        contributing area
Let lowestNeighbor be the node in selectedBreachingSite.neighbors outside
  contributingArea and with minimal elevation;
selectedBreachingSite.elevation ← lowestNeighbor.elevation;
if $selectedBreachingSite.breachingDistance = 2$ then
    Let pathIntoArea be the shortest path (in Chebyshev distance) from
      selectedBreachingSite to a node in contributingArea;
    pathIntoArea.next.elevation ← lowestNeighbor.elevation;
```

4.1.6 Normal vector similarity algorithm for terrace detection

The normal vector similarity algorithm (Algorithm 17) is aimed at assisting in the detection of geological terraces. Its input consists of a triangle face within the terrain mesh, whose normal vector will be taken as the reference vector, and an angle threshold value, which constrains the angle of other normal vectors to lie within the threshold in order to be considered similar to the reference vector. The subroutine iterates over all triangle faces in the DEM and computes the angle difference between the reference vector and the normal vector of said face. If the difference is smaller than the threshold value received as input, then the triangle is added to a list. The end result of this subroutine is a list of triangles with similar orientation, which may then be highlighted for visualization.

Algorithm 17: Normal vector similarity algorithm

```
Input: InputFace, AngleThreshold
Result: SimilarFaces : a list of triangle faces whose normal vectors lie within the specified
        range
InputVector ← InputFace.normalVector foreach face in terrain do
    FaceVector ← face.normalVector;
    Angle ← arccos $\frac{InputVector \cdot FaceVector}{InputVector.length * FaceVector.length}$;
    if $Angle \leq AngleThreshold$ then
        SimilarFaces.insert(face);
```

4.2 Implementation details

4.2.1 RWFlood algorithm

As a drainage network algorithm, the existing software design allows us to readily implement the RWFlood algorithm within a single RWFlood.h/cpp class by extending from the virtual DrainageAlgorithms class.

Hence, we code our header file as in Listing 1. In our header file we include several auxiliary functions to handle node flow directions, along with an enumeration for all 8 possible directions in the terrain grid. For handling flow directions, we make use of the flags field present in the runnel::Point class, a char-type member variable reserved for usage during the execution of algorithms. Under the assumption that we operate on a computer architecture which uses 8 bits to represent a char type, we employ these bits as direction indicators, setting the appropriate bit to 1 to represent flow direction.

Our implementation of the RWFlood algorithm involves two main functions: the flood function, as detailed in Algorithm 2, and the calculateWaterAccumulation function, as detailed in Algorithm 3. These are coded as described in Listings 2 and 3.

In Listing 2, we define `maxElev` and `minElev` and use these constants to create and handle the zero-indexed queue array (note that the queues in Algorithm 2 are `minElev`-indexed). Observing that the RWFlood algorithm modifies elevation data during the flooding process, we also define a `zValueStore` map to store the original values for later restoration. We initialize the DEM nodes by clearing any flags used by previously executed algorithms, and assign an outward-facing direction to all nodes located on the terrain boundary as required by the algorithm. The rest of the code is very similar to the corresponding lines in Algorithm 2, with the sole exception of the storage and later retrieval of original elevation data from `zValueStore`.

In Listing 3, we define a `maxWaterCount` variable which will store the highest water accumulation value recorded on the terrain. We use and store this value for visualization purposes, in order to assign edge colors based on a white-to-blue gradient by ascending water accumulation value. We also initialize the DEM nodes by resetting their water accumulation values to 1 and clearing any recorded upstream nodes from the `water_parent` member variable, in case a previously executed algorithm has modified them. The remaining code is also very similar to the pseudocode described in Algorithm 3, with the difference that we refer to nodes by their internal ID, where possible, instead of handling the node objects themselves.

4.2.2 Gleyzer algorithm

In the existing Runnel code, we find that the calculation and assignment of Strahler order values is performed in the Arbol (Tree) class, which is used to represent drainage network trees recursively. This calculation is performed by the `getNumberStrahlerHorton` function as described in Listing 4.

Hence, we choose to implement the Gleyzer algorithm within the Arbol class. For this purpose, we define additional functions to initialize the required structures, and we define a main function `computeNetworkStrahlerOrdering`, shown in Listing 5, to be exposed as a public function accessible to the rest of the system.

In most of the existing software, edges are handled as nodes in a single array, with each pair of consecutive nodes representing an edge, and obtained through the `getArbolEdges` function. We find this approach to be error-prone and difficult to debug when attempting to implement the Gleyzer algorithm correctly. Therefore, we instead define the EdgeList type as a vector of node pairs, each edge being represented by a pair object, and we initialize such a list after obtaining the necessary data from the `getArbolEdges` function.

We also define auxiliary functions for initialization. In Listing 6 we describe the `makeInflowingEdgeMap` function, which saves for each node a list of the IDs of the nodes that flow into it. Listing 7 shows the `makeUpstreamNodePerEdgeMap` function, which determines the ID of the upstream node for each edge in the edge list. These correspond to the NodesPerEdge and EdgesPerNode dictionaries populated by the MakeDictionaries pseudocode function shown in Algorithm 4. In the `makeUpstreamNodePerEdgeMap` function we consider the upstream node to be the highest (in terms of elevation) of the two nodes that compose the edge.

The actual calculation is performed by the recursive `streamOrdering` function described in Listing 8, to which we pass the dictionaries necessary for its operation. We choose to pass these dictionaries as parameters by reference, rather than defining global variables for all of them. One of these alternatives is necessary, as each recursive call requires accessing the data and may possibly modify some of it. The actual code of the `streamOrdering` function is nearly identical to the pseudocode presented in Algorithm 5.

4.2.3 Garbrecht-Martz flat resolution algorithm

The existing software does not explicitly support preprocessing procedures in its design. As a first approach to build a proof-of-concept, we choose to implement these preprocessing procedures as a new category of algorithms that may be executed `after` the terrain is constructed to alter model elevation data.

To implement this algorithm, we must be able to determine which nodes in the DEM belong to a flat zone. For this purpose, we consider a 3x3 square of points at the same elevation as the minimal unit of a flat. Thus, as shown in Listing 9 we implement a `extractFlatIds` function that scans the DEM for flats according to this definition, storing the IDs of the nodes that compose them.

Listing 10 shows the public function `run`, which performs the necessary calls and applies the combined gradient on the DEM. Listings 11 and 12 describe the `gradientTowardsLowerTerrain` and `gradientAwayFromHigherTerrain` functions shown in Algorithms 6 and 7, respectively, being very similar to their pseudocode descriptions.

For clarity, we opt to separate the addition of both gradients, and the detection of neutralized increments that require an extra half-increment, into different functions. Thus, Listing 13 shows the `combineGradients` function, which adds the two gradients together, while Listing 14 contains the `findHalfIncrementIds` function that determines the additional half-increments to be applied.

Finally, the `applyIncrements` function shown in Listing 15 performs the actual modification of elevation data.

4.2.4 Garbrecht-Martz pit removal algorithm

We implement the Garbrecht-Martz depression breaching algorithm in the `GMPitRemover` class, defining a public `run` function (shown in Listing 16) that handles the algorithm logic and calls the appropriate private functions previously described in pseudocode.

Within the `run` function in Listing 16, we detect a problem in execution while calculating the contributing area for an inflow sink. According to the original description shown in Algorithm 9, if the contributing area appears to be incomplete for a given window size, the window must be enlarged and the process must iterate until the window fully encompasses the contributing area. In practice, when operating on large DEMs, we observe that window sizes may grow into the hundreds or thousands, leading to computationally expensive recalculations of contributing areas on each iteration that make the software unusably slow in our preliminary tests. We also consider that theoretical contributing areas, according to this algorithm, may not always lie entirely within the boundaries of the DEM, which may result in calculations trying to expand contributing areas beyond them. Hence, we implement three additional considerations in our code:

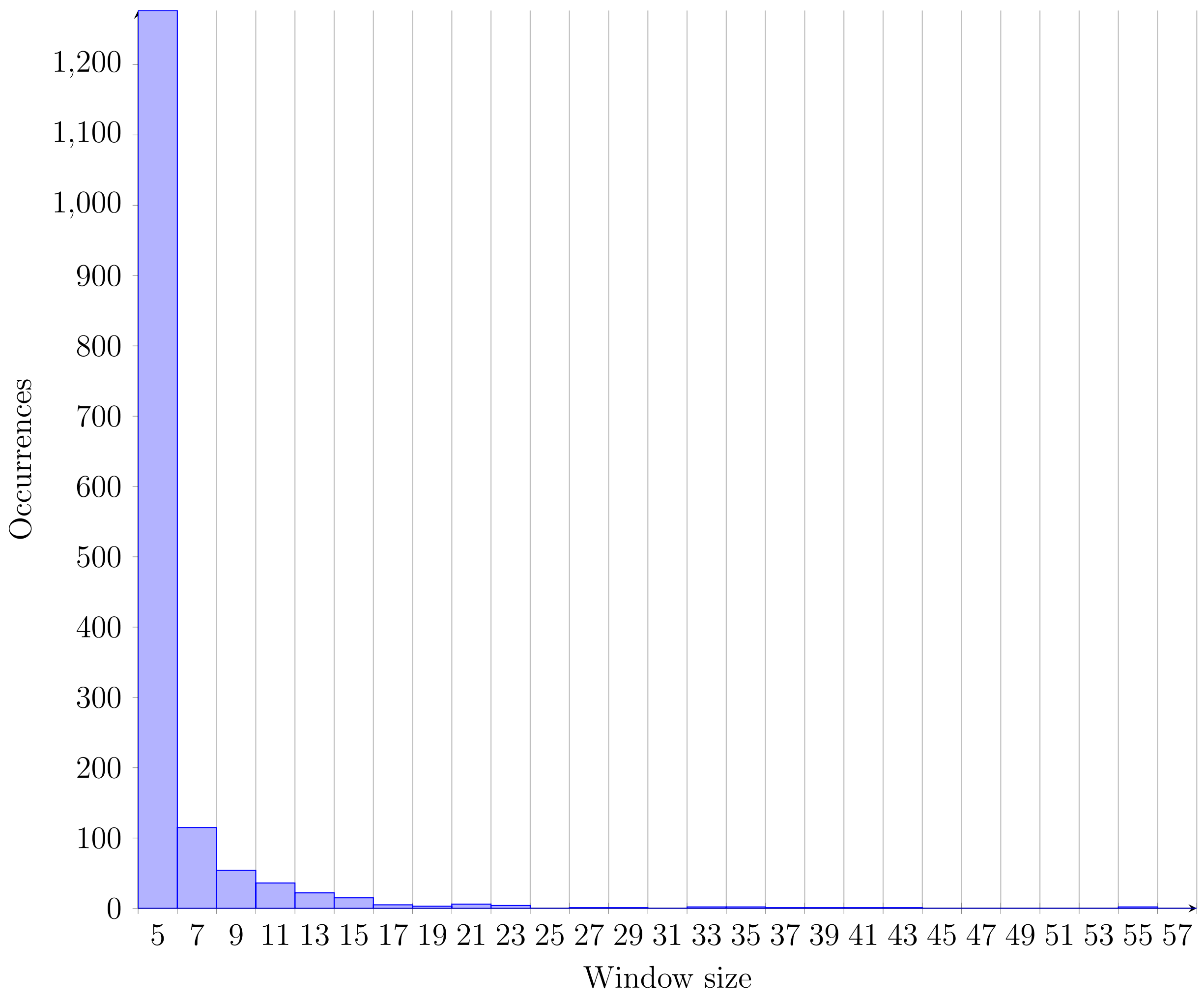


Figure 23: Plot depicting window size (in nodes, up to 60) vs. the number of closed depressions whose contributing areas lie entirely within a given window size. Most occurrences involve window sizes under 30.

- Rather than recomputing the entire contributing area on each iteration, we instead choose to build it incrementally by saving the results of previous iterations, thus analyzing only the new window boundary on every step.
- In Figure 23 we show the frequency distribution of window sizes for the contributing areas of closed depressions in the central Chile DEM. Based on this distribution, as a heuristic we set an upper window size limit of 30 to prevent excessive operations that are often unable to build a complete contributing area within the constraints of the DEM.
- If the window size for a given inflow sink attempts to expand beyond the height or width of the DEM, we skip said sink.

On the subject of flags, throughout the pseudocode description we require flagging nodes for two essentially different purposes: to determine which nodes belong to the partially constructed contributing area, and to prevent inflow sinks and depression boundaries from being considered more than once (for instance, when multiple inflow sinks belong to the same contributing area).

Therefore, we separate flag usage as follows:

- For contributing area calculation, to avoid adding a new flag field to the node data structure, we keep track of contributing nodes in each iteration by using three lists: `contributingArea`, which holds the full list of all nodes included in the contributing area up to that point; `pendingNodes`, which represents the 'flagged' nodes that must be processed in the current iteration, and `newNodes`, which represents the newly 'flagged' nodes that should be processed in the next iteration. We take care to ensure that we do not attempt to flag a node more than once. This is implemented in the `computeContributingArea` function shown in Listing 17.
- For avoiding duplicate evaluation, we use the pre-existing 'flags' field in the node data structure, setting this field to 1 when a node is flagged and 0 otherwise. Before processing an inflow sink, we check if it is already flagged, and skip it if that is the case.

Listings 18 and 19 show the functions in charge of obtaining potential outlets and breaching sites for the main algorithm. A potential outlet is defined as a node within the contributing area that has at least 1 neighbor that lies outside the contributing area and is at a lower elevation than the node. A potential breaching site is defined as a node within the contributing area located at the same elevation as the lowest potential outlet, with at least 1 neighbor outside the contributing area at a lower elevation, and within breaching length of a node within the contributing area and at a lower elevation than the lowest potential outlet.

Listing 20 describes the `performBreaching` function responsible for adjusting the elevation of breaching sites to create outlets for closed depressions, and Listing 21 shows the `nodeDistance` function used to compute the Chebyshev distance between nodes, described in Algorithm 13. As the existing DEM data structure holds nodes in a single array, rather than a matrix, we need to calculate the row and column of each node from its index in the array by relying on the terrain width value.

4.2.5 Peucker algorithm fix

The Peucker algorithm is implemented in the `PeuckerDrainageNetwork` class, with the core of the algorithm itself being implemented in the `calculateGrid` function shown in Listing 22.

We find a software defect in the final `for` loop of this function, where the code erroneously flags the highest node in the square instead of the node being processed, and due to which only a single point is arbitrarily flagged when processing a 2x2 region, even if multiple points within the square have the same elevation value.

Thus, our fix is straightforward as shown in Listing 23, substituting `max_point->setFlagsOn` for `pto->setFlagsOn` and therefore allowing the function to flag all required points.

4.2.6 Normal vector similarity algorithm for terrace detection

For terrace detection purposes, we decide to implement a category of algorithms that derive from a new virtual base class, `TerraceDetectionAlgorithm`, similarly to the class design of drainage network algorithms. We declare the header file of this class as described in Listing 24.

For the actual implementation of the normal vector similarity algorithm, we extend this class with our `NormalVectorSimilarityAlgorithm` class, with header and implementation files shown in Listing 25 and Listing 26 respectively.

As shown in Listing 26, the reference triangle is obtained by passing a user-clicked point from the interface.

5 Results and discussion

In this chapter we present the main results and comment on the main observations arising from our results. Full results with uncropped images are presented in Appendix B.

5.1 Drainage network extraction on raw terrain

Peucker algorithm fix: The fixed version of the Peucker algorithm produces results that may be described as less cluttered. This is necessarily so, as the software defect described had an effect of erroneously increasing the total number of nodes in the calculated drainage network. The fixed version allows for a clearer visual recognition of the streams in the higher areas of the DEM, as depicted in Figure 55 (Appendix B).

A direct consequence of the implemented solution, however, is the absence of proper drainage recognition in flat areas of the DEM (Figure 24). While the absence of drainage in a considerable portion of the DEM poses significant problems for any subsequent network analysis, one must also consider that the drainage calculated by the original implementation for flat areas is spurious in nature, given the arbitrary selection of flagged and non-flagged points in flats based on processing order. Hence, this effect is not undesirable, but rather reflects a more accurate implementation of the original algorithm.

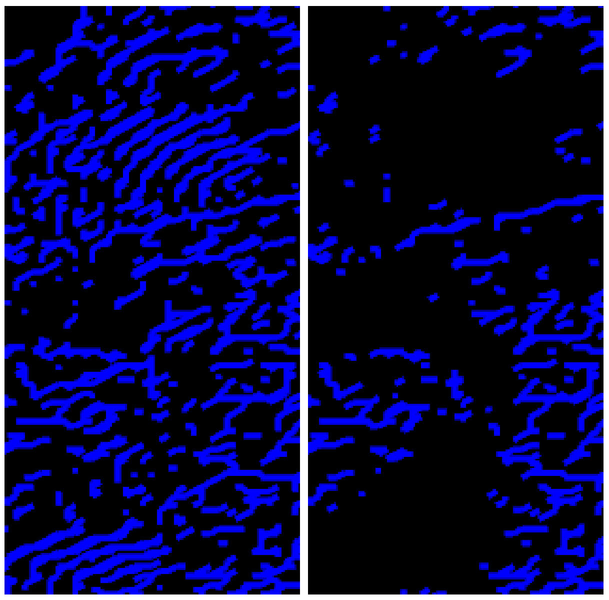


Figure 24: Image depicting the drainage network calculated by the erroneous (left) and corrected (right) Peucker algorithm over a large flat zone located in the central region of the DEM covering central Chile. As shown, the fixed algorithm is unable to calculate drainage over flat zones of the DEM.

Despite the performed fix, however, one must also consider that this algorithm may not be suitable for some purposes. Due to the limited nature of its drainage calculation algorithm, it provides no information about water accumulation throughout the network, on which other subsequent analysis algorithms may potentially rely to perform their calculations.

This algorithm also produces a fair amount of cycles within the network (e.g. Figure 25). While the Gleyzer algorithm for Strahler order calculation in braided networks may adapt to this situation, it might not necessarily be the case for other subsequent analysis algorithms. Therefore, one may prefer to utilize drainage network extraction algorithms that minimize the amount of cycles generated.

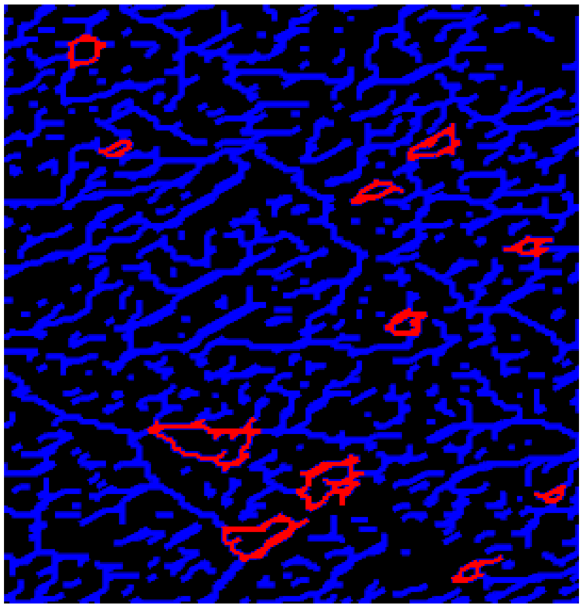


Figure 25: Image depicting drainage cycles (highlighted in red) within a section of the drainage network calculated by the Peucker algorithm on the DEM covering central Chile. The presence of loops may prove to be troublesome for analysis algorithms.

RWFlood algorithm: We observe that the results generated by RiverTools and the RWFlood algorithm in Runnel are fairly similar for higher and lower areas of the DEM alike (Figures 26 and 27 respectively).

The main difference between the results lies in the management of flat areas in the central regions of the DEM. For these areas, the results presented by Runnel show a large amount of parallel flow lines which prevent accurate network extraction (Figures 28 and 29), whereas RiverTools employs some mechanism to process these flats. This result is not unexpected, as the RWFlood algorithm does not incorporate any flat resolution method into its design, and we execute this algorithm on the raw DEM, without the involvement of any preprocessing procedures. Therefore, we believe that the RWFlood algorithm, coupled with the application of the Garbrecht-Martz flat resolution and depression breaching algorithms, may provide results of good quality in terms of their similarity with RiverTools results.

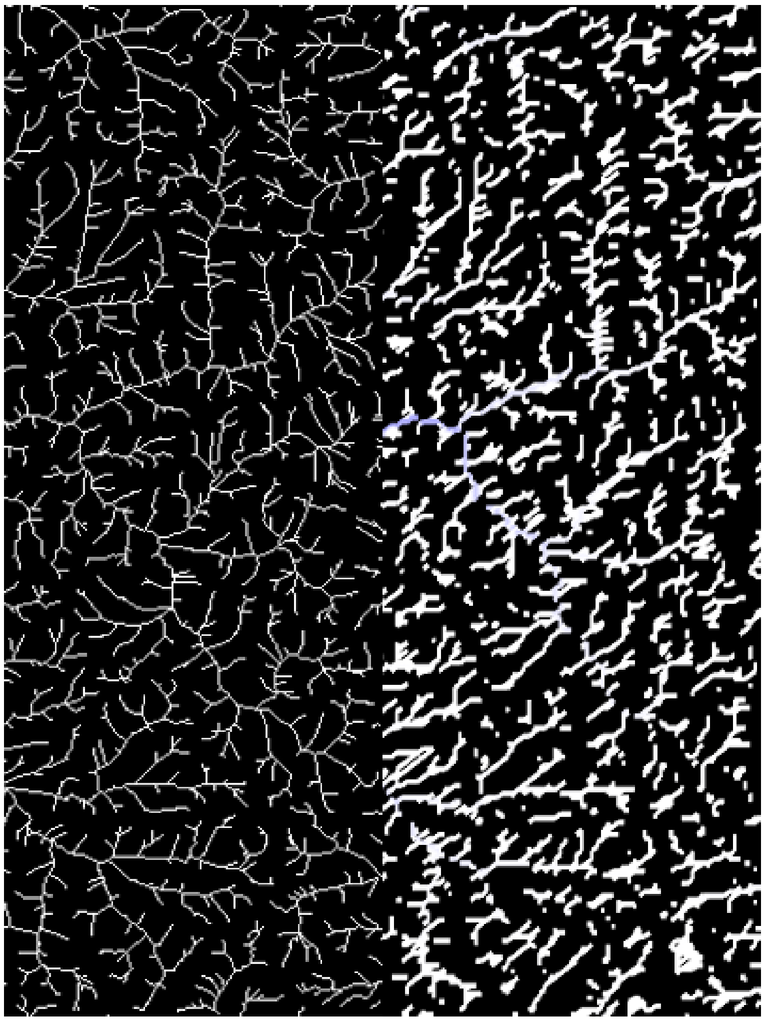


Figure 26: RiverTools (left) and RWFlood (right) drainage network extraction results for the higher areas of the DEM covering central Chile. We consider the RWFlood results to be fairly similar, albeit less connected.

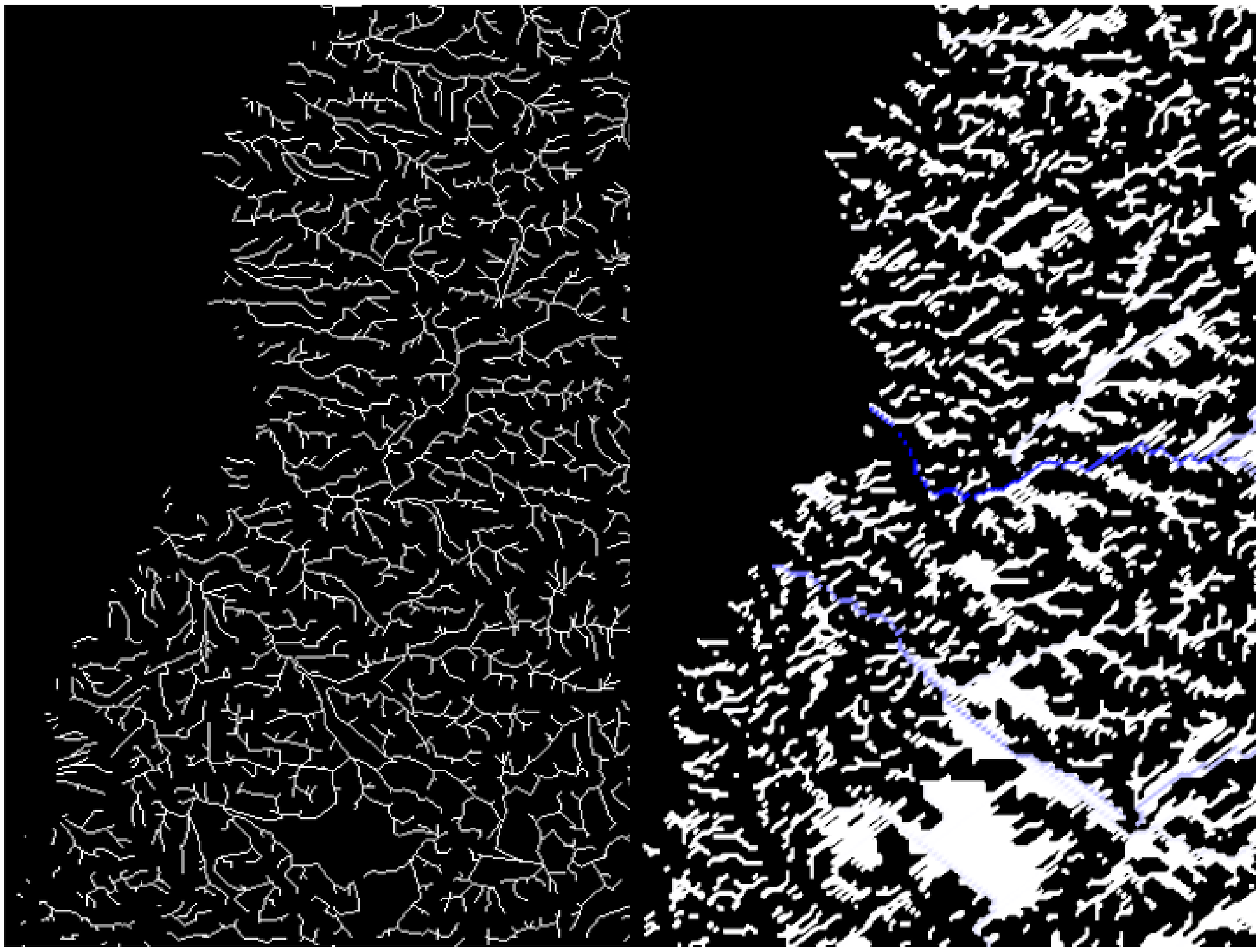


Figure 27: RiverTools (left) and RWFlood (right) drainage network extraction results for the lower areas of the DEM covering central Chile. We consider the RWFlood results to be fairly similar, although less so than in the higher elevation case.

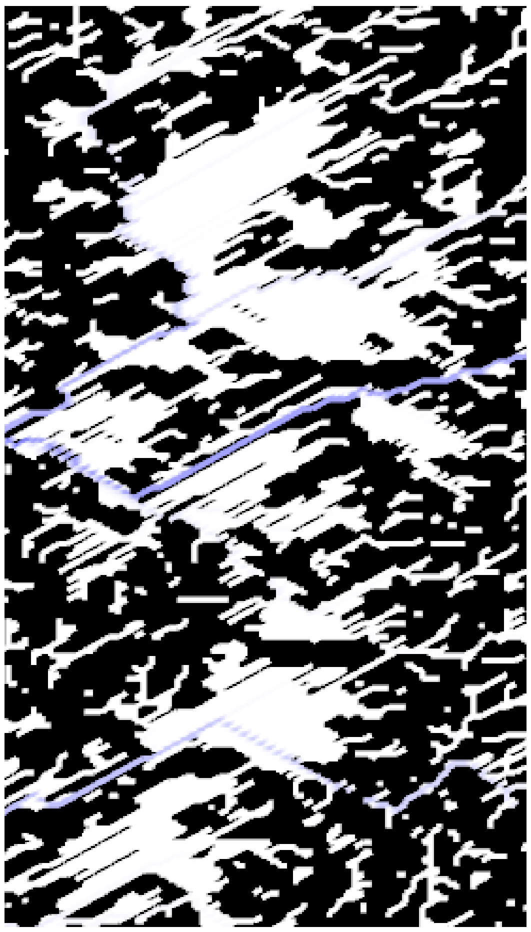


Figure 28: Example of parallel flow lines calculated by the RWFlood algorithm in flat areas of the DEM. This image depicts a large flat zone located in the central region of the DEM covering central Chile.

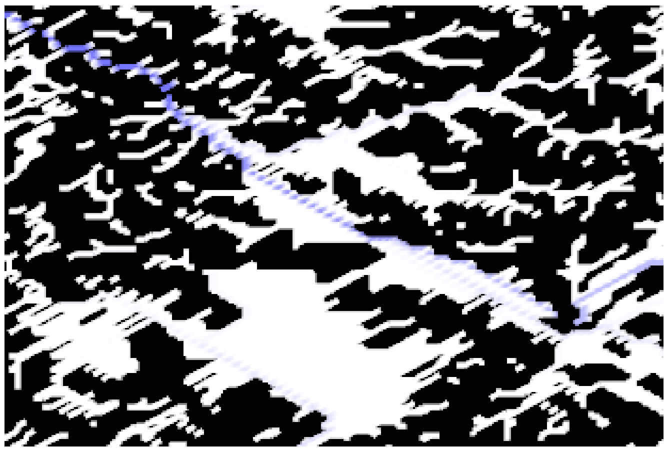


Figure 29: Example of parallel flow lines calculated by the RWFlood algorithm in flat areas of the DEM. This image depicts a smaller flat zone located in the lower left region of the DEM covering central Chile.

5.2 Preprocessing techniques

Garbrecht-Martz flat resolution algorithm: Regarding the application of the imposed gradients algorithm before executing the Peucker algorithm, as shown in Figure 30, we note that the flat areas in the central regions of the DEM show a greater amount of drainage nodes after preprocessing. This result implies that a gradient has been correctly applied on the flat regions, given that the Peucker algorithm will flag all points on a flat 2x2 square (thus excluding all four nodes from the drainage network), but will flag less than four in squares that contain different elevation values.

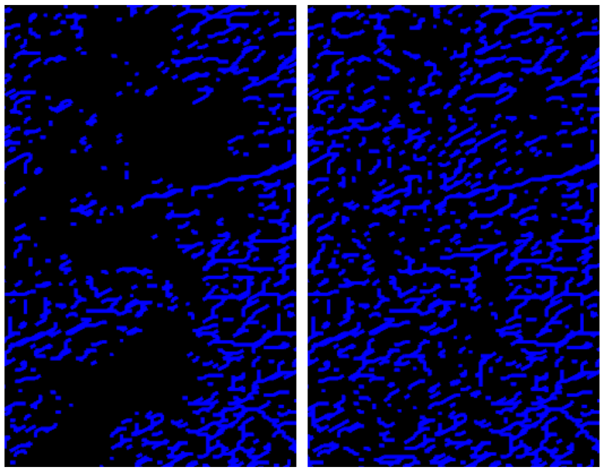


Figure 30: Drainage network extraction results produced in Runnel by the fixed Peucker algorithm on raw terrain (left), and terrain preprocessed to remove flat zones (right), for a flat zone located in the central area of the DEM covering central Chile. As depicted, the imposed gradients preprocessing algorithm allows the Peucker algorithm to achieve better results in flat areas previously devoid of drainage.

Concerning the O'Callaghan algorithm, as shown in Figure 31, we observe a similar but less pronounced effect, with isolated nodes appearing in flat areas previously devoid of drainage (upper left area of the image), and previously isolated nodes becoming connected (central and bottom area of the image). The appearance of these nodes reflects a greater maximum water accumulation value in said areas, as the visualization provided by Runnel for O'Callaghan algorithm results employs a user-defined water level threshold to determine which nodes should be rendered blue. This is directly attributable to the Garbrecht-Martz flat resolution algorithm, as the imposition of microrelief on previously flat terrain allows the O'Callaghan algorithm to accumulate water into the lowest neighbors of each node within the processed area (as per its steepest descent strategy), whereas applying the algorithm on flat terrain generates isolated drainage nodes with no lower neighbors.

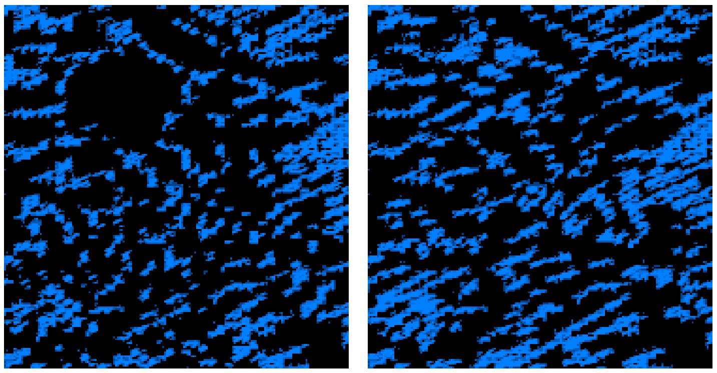


Figure 31: Drainage network extraction results produced in Runnel by the O'Callaghan algorithm on raw terrain (left), and terrain preprocessed to remove flat zones (right), for a flat zone located in the central area of the DEM covering central Chile. As depicted, the imposed gradients preprocessing algorithm allows the O'Callaghan algorithm to achieve better results in flat areas where it would normally produce isolated nodes.

With respect to the RWFlood algorithm, the execution of the imposed gradients algorithm results in the removal of an important proportion of parallel flow lines throughout flat regions of the DEM (Figures 32 and 33), allowing for the extraction of a drainage network of higher quality than by applying the RWFlood algorithm directly on the raw DEM.

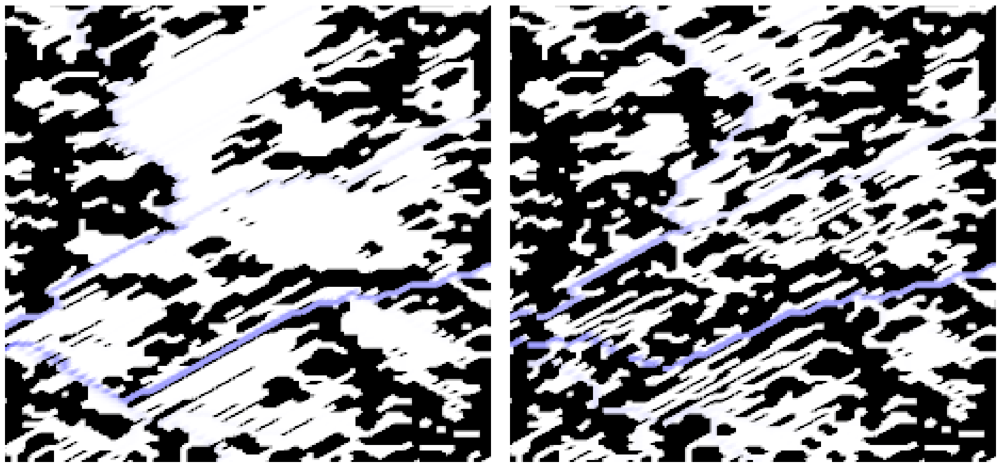


Figure 32: Drainage network extraction results produced in Runnel by the RWFlood algorithm on raw terrain (left), and terrain preprocessed to remove flat zones (right), for a flat zone located in the central area of the DEM covering central Chile. As depicted, the imposed gradients preprocessing algorithm allows the RWFlood algorithm to obtain a better defined drainage network, reducing the parallel flow phenomenon.

An overarching observation of these results involves the presence of isolated nodes. The drainage network generated in the preprocessed case contains a large amount of disconnected nodes in the regions altered by the imposed gradients algorithm, most notably in the Peucker and O'Callaghan networks. We believe that some postprocessing method to link these nodes may be necessary to obtain a full drainage network for the DEM.

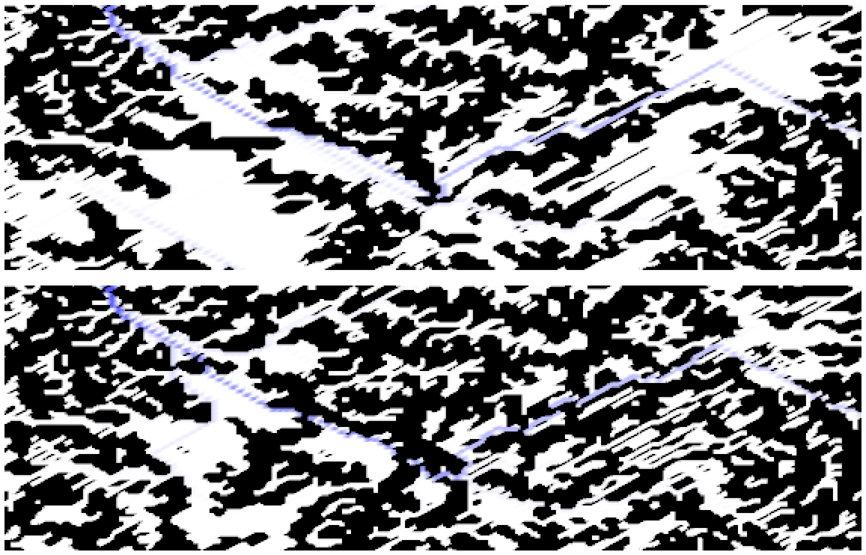


Figure 33: Drainage network extraction results produced in Runnel by the RWFlood algorithm on raw terrain (top), and terrain preprocessed to remove flat zones (bottom), for a flat zone located in the southern area of the DEM covering central Chile. As depicted, the imposed gradients preprocessing algorithm allows the RWFlood algorithm to obtain a better defined drainage network, reducing the parallel flow phenomenon.

Garbrecht-Martz depression breaching algorithm: The visually observable effect of the depression breaching algorithm on drainage networks is much less significant compared to the flat resolution algorithm. This is an expected result, as our implementation of the depression breaching algorithm limits the contributing area of a closed depression to a maximum of 30 nodes in each direction on the two-dimensional plane, and the overall impact of the depression breaching algorithm on node elevation is lower than in the imposed gradients algorithm, in which the elevation of every node within a flat region is altered.

With the Peucker algorithm we are only able to observe small scale differences, such as in the area depicted in Figure 34. In this case, two small zones previously devoid of drainage (and therefore flat, given the Peucker algorithm implementation) develop several streams and isolated nodes after executing the depression breaching algorithm. This is due to the presence of small closed depressions in these zones, whose breaching & filling process introduces minor elevation differences that subsequently allow the Peucker algorithm to keep some nodes unflagged, thus displaying them in blue.

The results generated by the O'Callaghan algorithm also show a small effect in terms of appearance and disappearance of blue nodes in throughout the terrain (e.g. Figure 35). This reflects a change in water accumulation values in each zone, as a result of elevation modifications introduced by the breaching & filling process that drives nodes either above or below the water level threshold for visualization.

We do not spot any visible differences with the RWFlood algorithm.

As the filling phase of the depression breaching algorithm generates new flat zones throughout the terrain, it is necessary to execute a flat resolution algorithm afterwards. Therefore, one cannot rely on the depression breaching algorithm alone to produce satisfactory results.

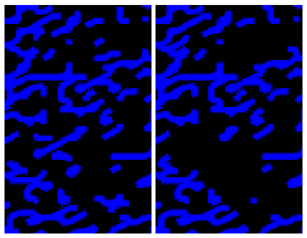


Figure 34: Example of small differences caused by the Garbrecht-Martz depression breaching & filling algorithm on a Peucker-based drainage network. Left side depicts drainage calculation on raw terrain. Right side shows drainage calculation on preprocessed terrain. These differences are attributable to minor elevation changes in closed depressions that allow the Peucker algorithm to keep some nodes unflagged.

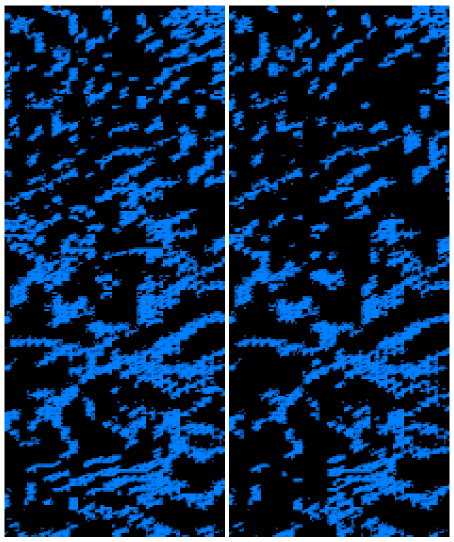


Figure 35: Example of small differences caused by the Garbrecht-Martz depression breaching & filling algorithm on an O'Callaghan-based drainage network. Left side depicts drainage calculation on raw terrain. Right side shows drainage calculation on preprocessed terrain. These differences are attributable to water accumulation changes induced by local terrain relief alterations from the breaching & filling mechanisms.

5.3 Drainage network extraction on fully preprocessed terrain

In Figure 36 we note that the fully preprocessed DEM is still insufficient to extract a full drainage network based on the Peucker algorithm. Isolated nodes are abundant in zones shown to be originally flat. We also observe this phenomenon in Figure 37 (O'Callaghan algorithm), albeit the water level visualization threshold given by the user also has an effect on whether a stream may appear disconnected or not. This illustrates that, while some degree of elevation change can be introduced through the Garbrecht-Martz preprocessing algorithms, there is no guarantee that a connected path may be found with a given strategy for drainage network extraction, and thus a postprocessing algorithm to construct a full drainage tree from flagged nodes or water accumulation data may be desirable in all cases regardless.

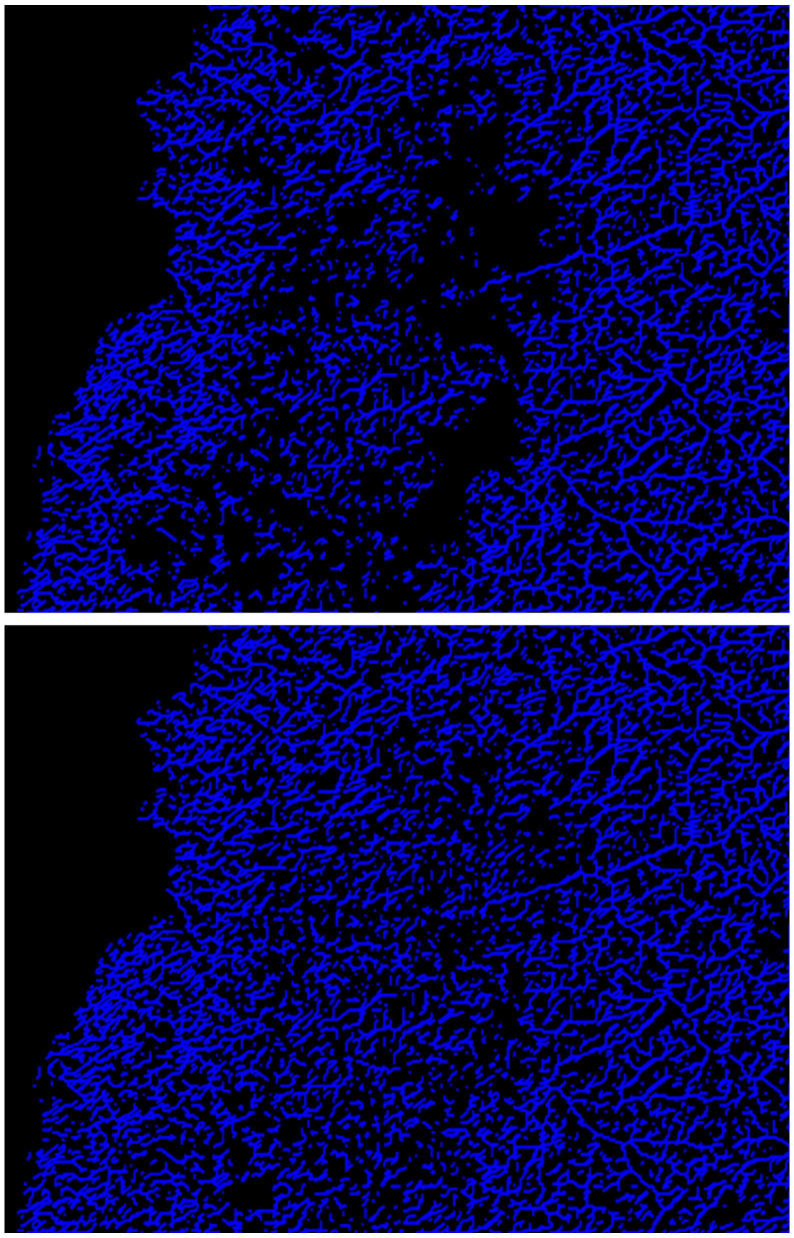


Figure 36: Drainage network extraction results produced in Runnel by the fixed Peucker algorithm on raw terrain (top), and terrain preprocessed to remove flat zones and closed depressions (bottom), for the DEM covering central Chile. The fully preprocessed DEM is insufficient to produce a fully connected drainage network.

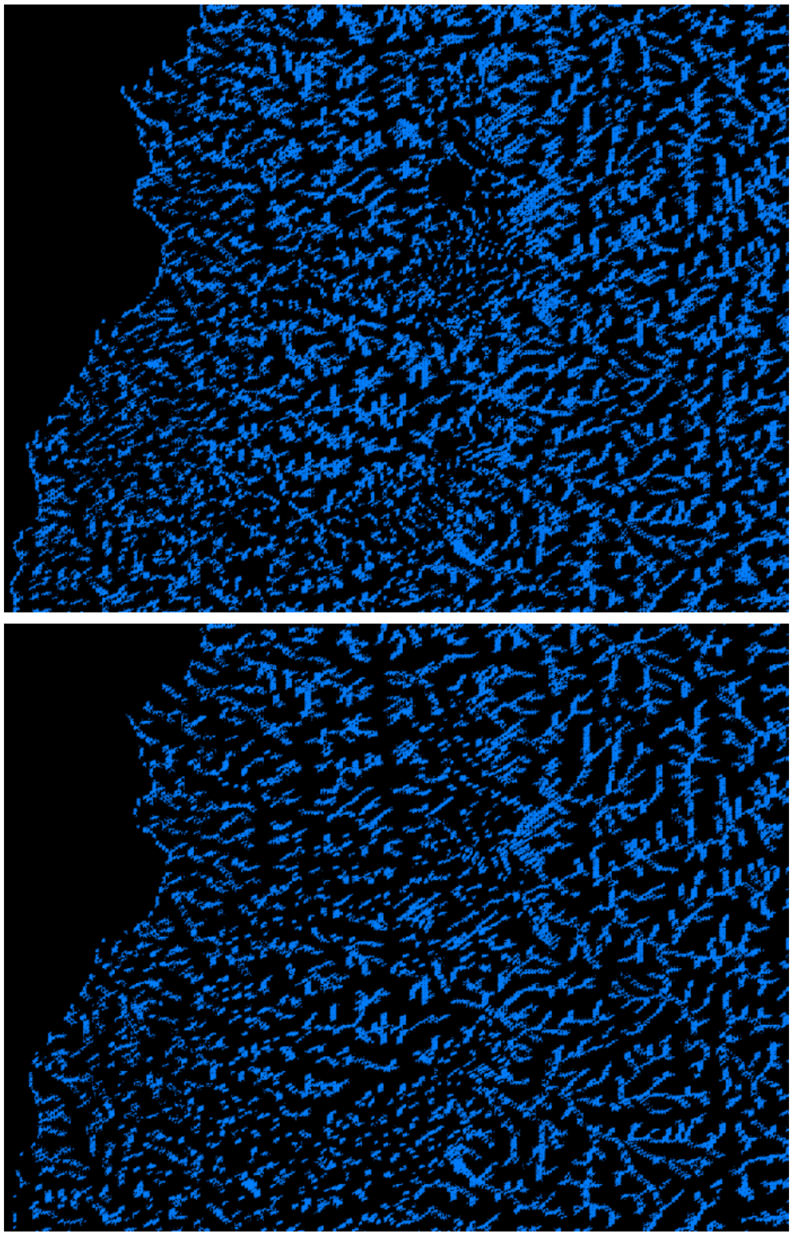


Figure 37: Drainage network extraction results produced in Runnel by the O'Callaghan algorithm on raw terrain (top), and terrain preprocessed to remove flat zones and closed depressions (bottom), for the DEM covering central Chile. While the drainage network appears better defined in the preprocessed case, a large amount of isolated nodes and disconnected streams are visible.

As a result, in Figures 38 (Peucker) and 39 (O'Callaghan), we observe that the drainage networks generated by RiverTools and Runnel are visually similar for the higher areas of the DEM (right side in each image), but Runnel networks break down in formerly flat areas processed by the Garbrecht-Martz algorithms (center).

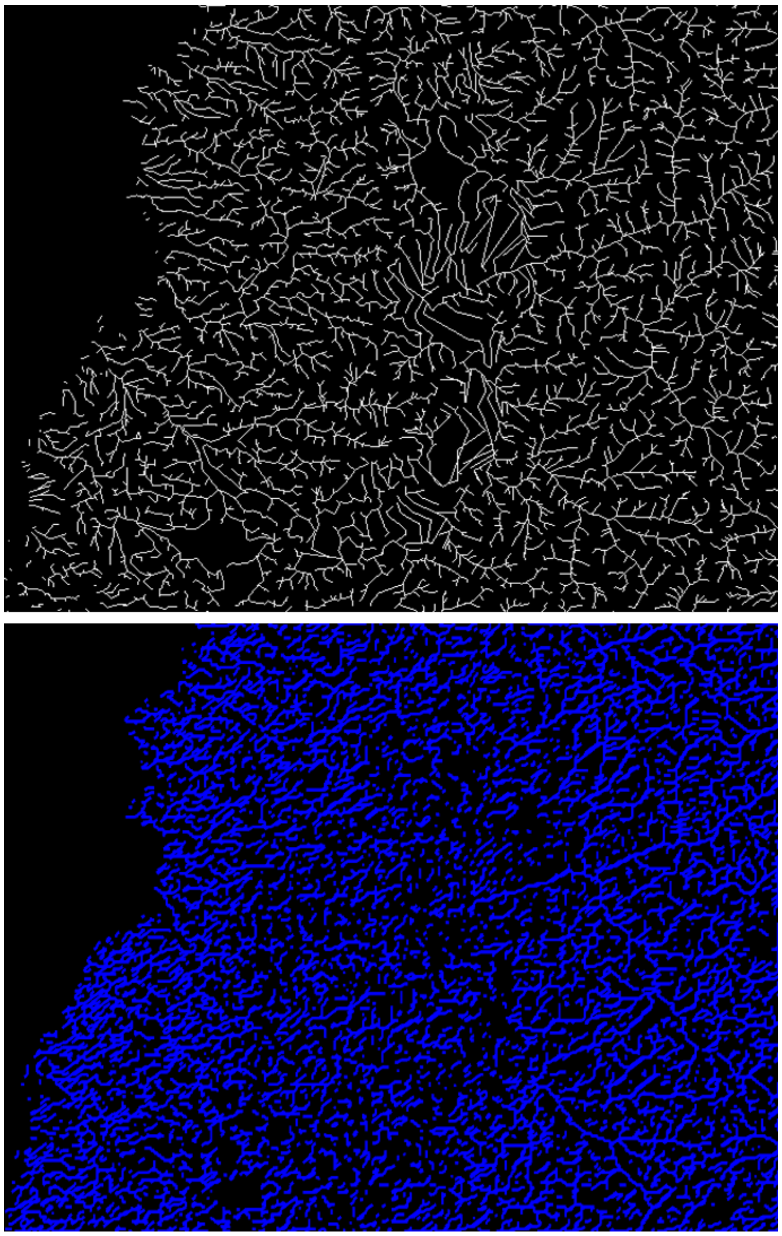


Figure 38: Drainage network extraction results produced by RiverTools (top) and the Peucker algorithm in Runnel (bottom) after applying the imposed gradients and depression breaching preprocessing algorithms. We consider the networks to be similar at higher elevations (right), but Peucker results decrease in quality beyond the preprocessed flats.

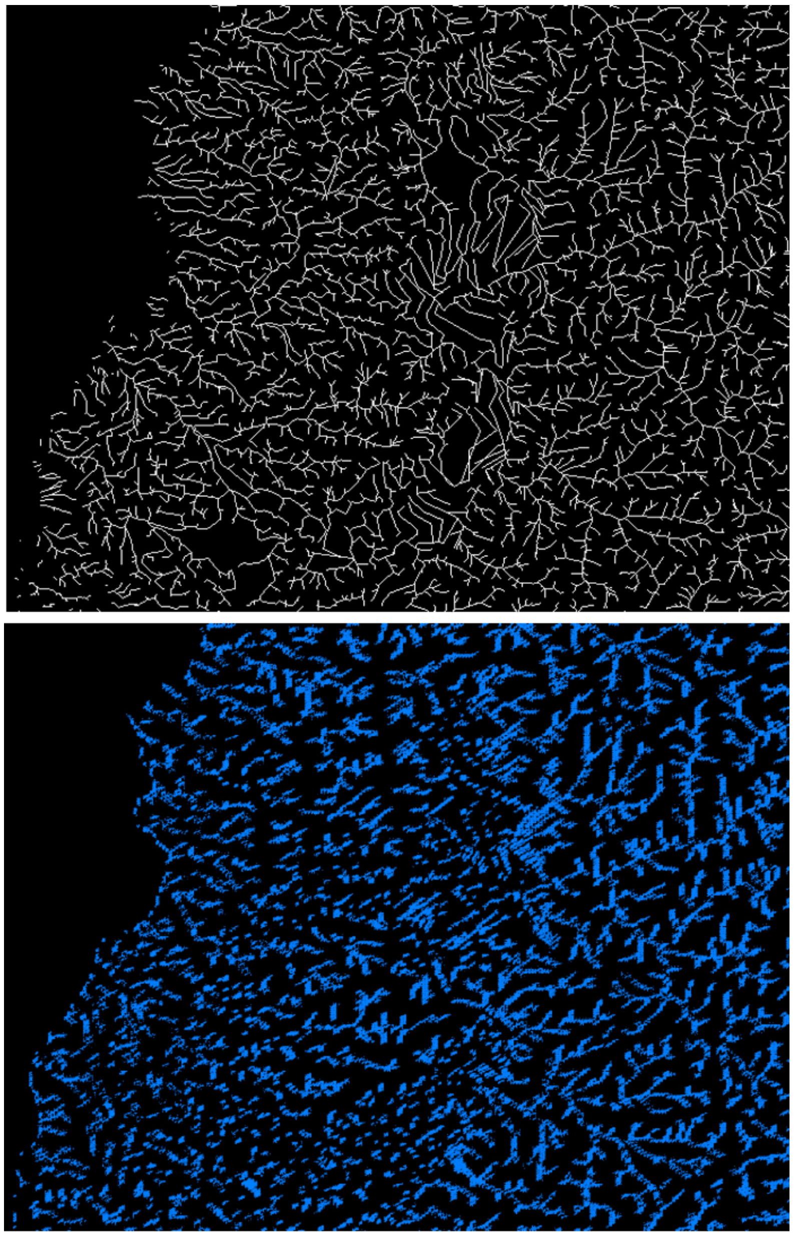


Figure 39: Drainage network extraction results produced by RiverTools (top) and the O'Callaghan algorithm in Runnel (bottom) after applying the imposed gradients and depression breaching preprocessing algorithms. We consider the networks to be fairly similar in structure, although O'Callaghan results tend to produce a large quantity of isolated and disconnected nodes in lower areas of the DEM (left).

Figure 40 shows some improvement over the application of RWFlood on raw terrain, as previously mentioned, reducing the density of parallel flow lines in flat areas. As shown in Figure 41, the application of the Garbrecht-Martz preprocessing algorithms has a significant impact on flat areas, allowing for a better defined drainage network with fewer isolated nodes that more closely resembles the results provided by RiverTools. However, we again highlight the necessity of a postprocessing linking algorithm in Runnel, as the RWFlood results also display interrupted and disconnected streams that cannot be directly used for subsequent drainage analysis algorithms.

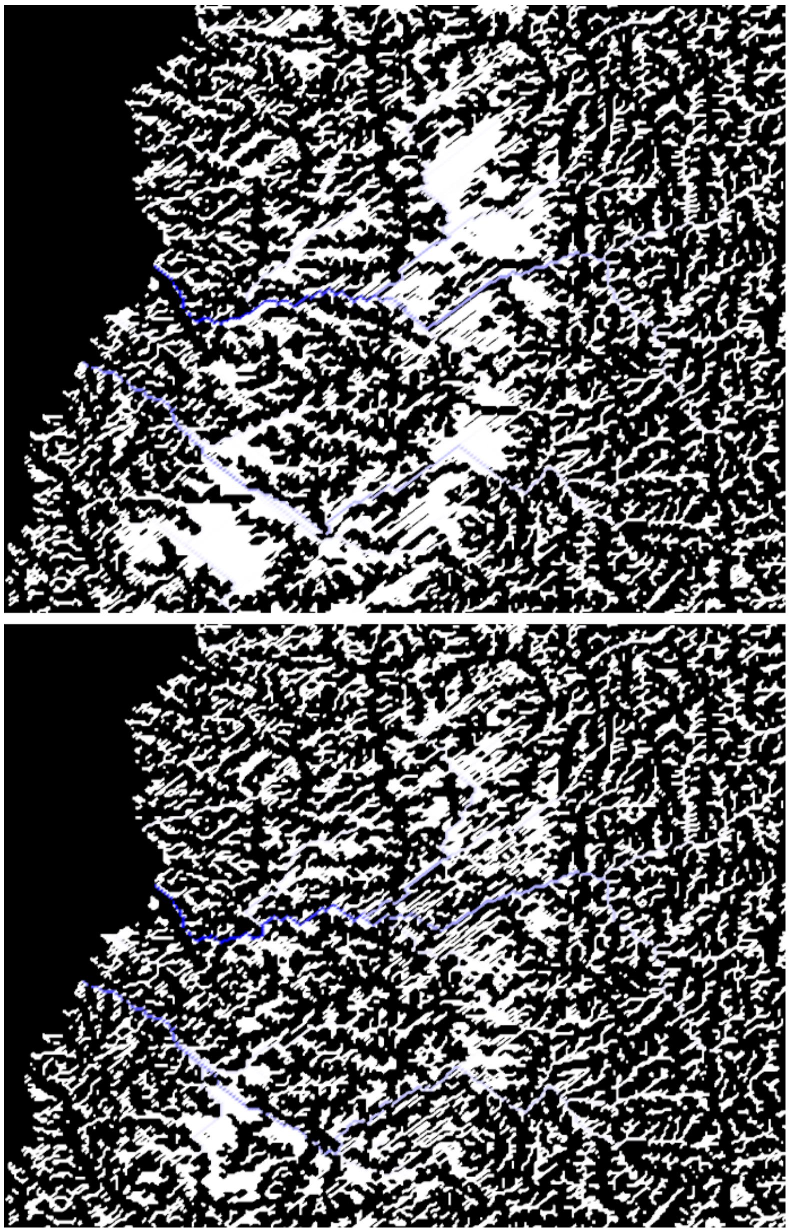


Figure 40: Drainage network extraction results produced in Runnel by the RWFlood algorithm on raw terrain (top), and terrain preprocessed to remove flat zones and closed depressions (bottom), for the DEM covering central Chile. Parallel flow lines calculated by the algorithm in flat areas are significantly reduced, improving the quality of the extracted drainage network.

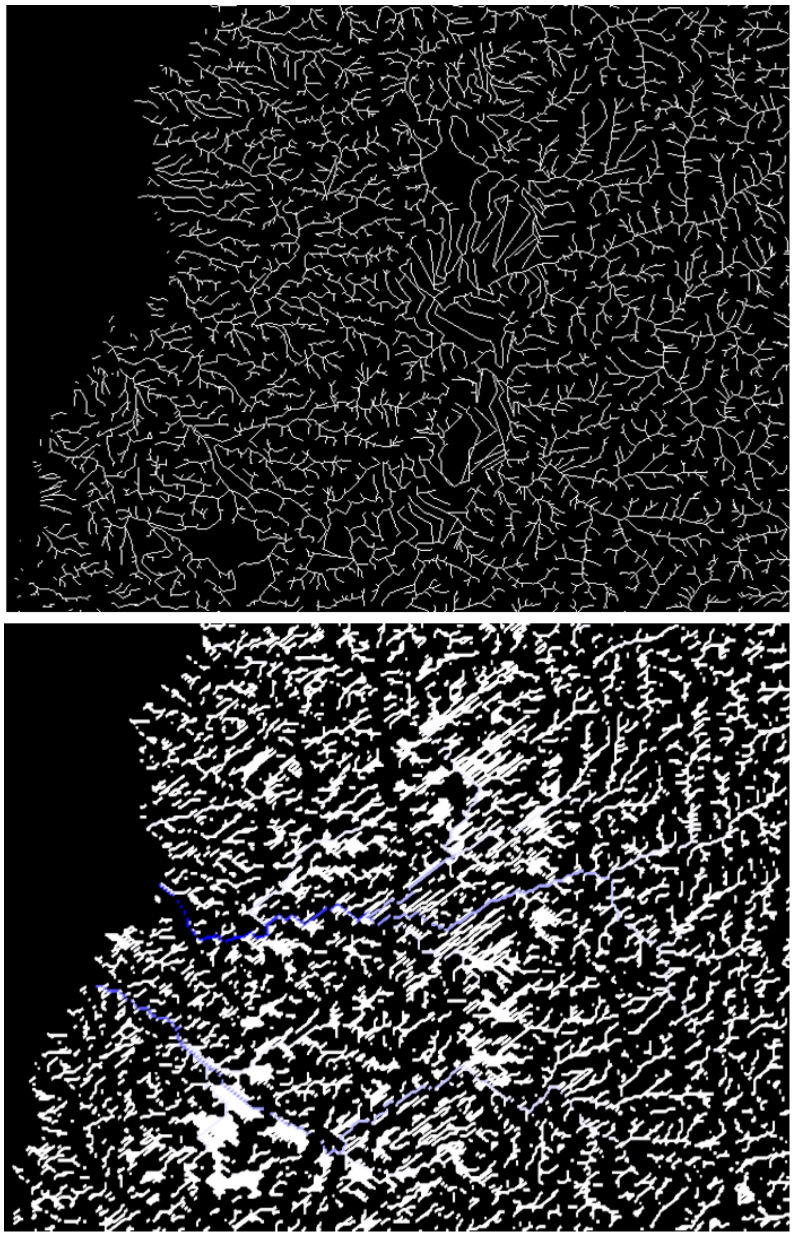


Figure 41: Drainage network extraction results produced by RiverTools (top) and the RWFlood algorithm in Runnel (bottom) after applying the imposed gradients and depression breaching preprocessing algorithms. We consider the RWFlood results to be adequate and similar to those provided by RiverTools, although the handling of flat areas in Runnel is less effective and results in poorer network visibility.

The main rivers depicted in Figure 42 appear to coincide with the main streams detected by the RWFlood algorithm (shown in blue hue). Given the low resolution of the DEM used, we consider these results to reflect positively on the effectiveness of the RWFlood algorithm together with the preprocessing procedures implemented in our work.

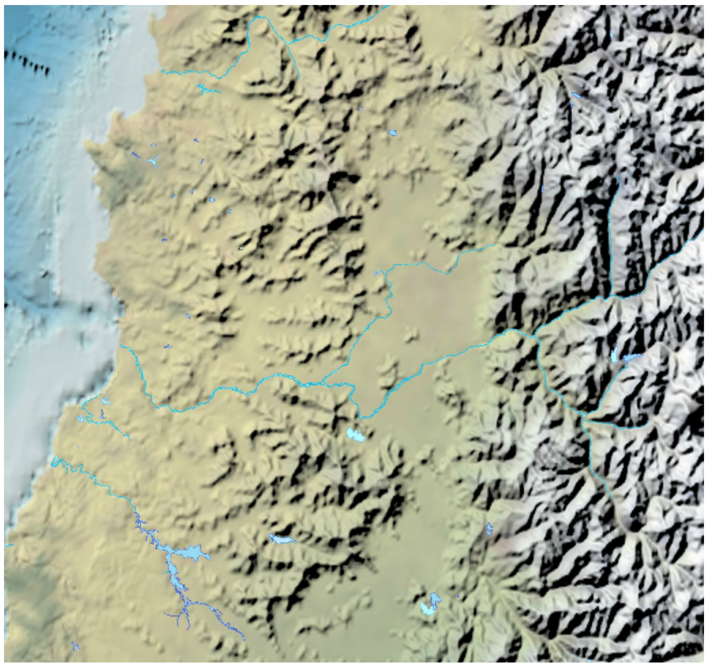


Figure 42: Map produced by the Military Geographic Institute of Chile, depicting main rivers located in central Chile within the boundaries of the corresponding DEM.

5.4 Strahler order calculation

In Figure 43 (based on the Peucker algorithm), the left side illustrates the existing problem with Strahler order calculation, by which multiple order 4-6 streams are present in the highest areas of the DEM where most streams should be of low order. These high order streams do not appear to follow a downstream direction, instead taking seemingly erratic turns throughout the terrain. The right side, on the other hand, shows the results produced by our implementation of the Gleyzer algorithm. Our results show an improvement in terms of regularity in this sense, with no such erratic high order streams present. Similarly, Figure 44 (based on the RWFlood algorithm) shows that this regularity is not limited to Peucker-based networks.

However, these results are still unsuitable for any practical application, as our implementation is seemingly unable to retain the Strahler order of a stream as it progresses towards the ocean along the terrain, causing most of the stream segments within the network to be of order 1 or 2. This behavior is entirely unintended, and ultimately provides no useful information regarding the relative importance of streams within the network.

Closer manual inspection of the results on a 3D terrain mesh appears to indicate that Strahler orders are calculated and retained correctly only until a local minimum (elevation-wise) is reached along the path (Figure 45).

As shown in Algorithm 4, part of the information necessary for the Gleyzer algorithm to work is a dictionary indicating the direction of flow for every pair of adjacent nodes (i.e. which node is the source, and which node is the destination). To compute this data, we need to rely on the elevation values of the nodes in each edge, as shown in Listing 7. Some drainage network algorithms do

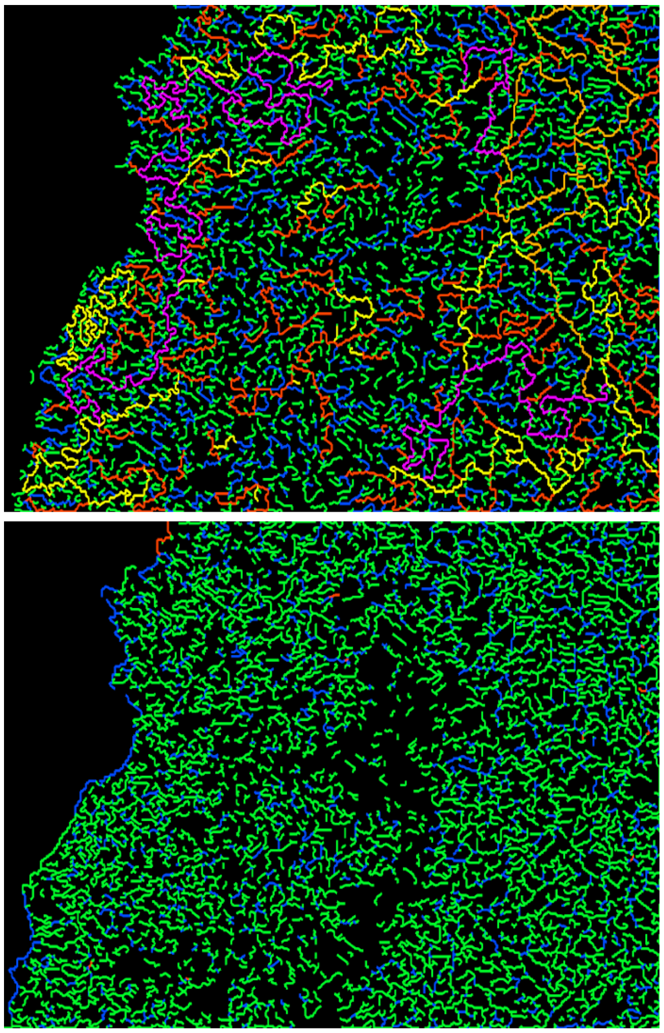


Figure 43: Strahler order calculation results with the base mechanism provided by Runnel (top) and with our implementation of the Gleyzer algorithm (bottom) on fully preprocessed terrain with a Peucker network. Strahler orders are color-coded: green for order 1, blue for order 2, red for order 3, yellow for order 4, orange for order 5 and purple for order 6. As shown, the original results are erratic, with high order streams visible in the highest areas of the DEM. Our results show greater regularity, but streams are disconnected and order data is not correctly transmitted downstream.

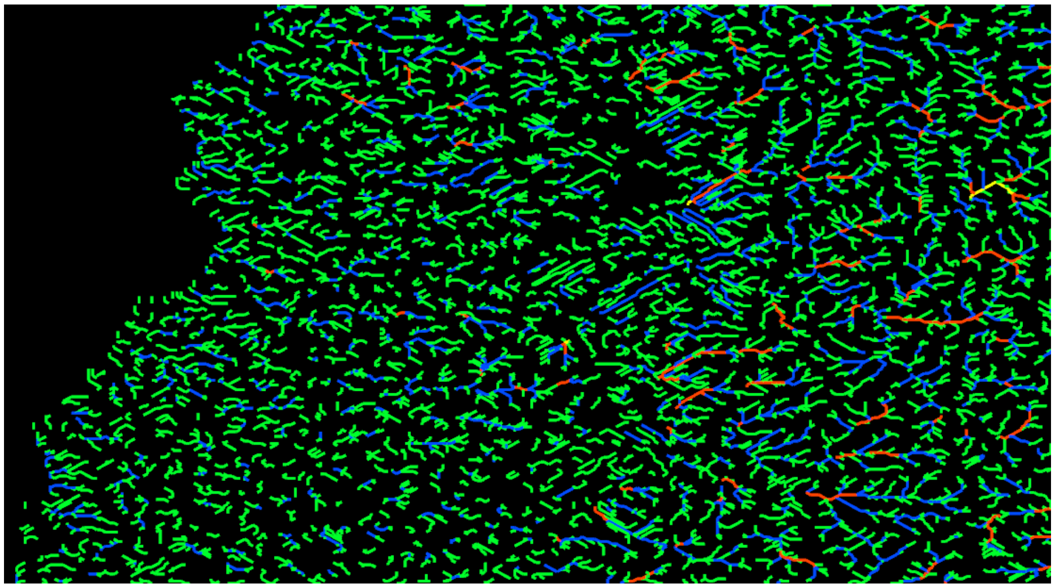


Figure 44: Strahler order calculation results with our implementation of the Gleyzer algorithm on fully preprocessed terrain with an RWFlood network. Strahler orders are color-coded: green for order 1, blue for order 2, red for order 3, yellow for order 4.

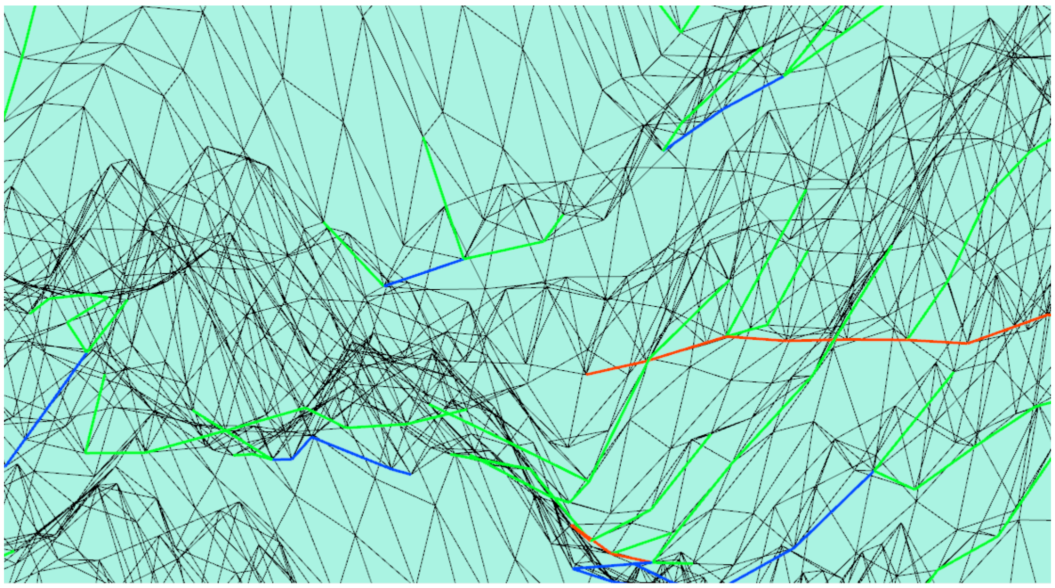


Figure 45: Example of the influence of local minima on Gleyzer algorithm results, executed on raw terrain with a Callaghan-based drainage network. The descending order 3 stream (red, center-right area of image) ends at a local minimum between nearby hills.

perform an alternate (not necessarily elevation-based) source-destination calculation themselves (e.g. O'Callaghan algorithm, which saves source nodes in the `water_parent` variable within the node data structure). However, other algorithms do not provide any information in this sense (e.g. Peucker algorithm, which only considers whether a node belongs to the stream network or not). We cannot make any assumptions, therefore, on the source-destination relation of nodes in a stream segment, and thus we only use elevation data directly, defining the source node as the highest of the two.

The unintended consequence of this approach is that, when a local minimum is reached along a stream, no other adjacent nodes will list the local minimum as a source node, and hence, no node may inherit the Strahler order it carries (Algorithm 5). Therefore, this approach is unworkable in its current form.

A possible solution to this problem entails filling isolated local minima to the elevation of their next lowest neighbor. In this sense, our implementation of the Garbrecht-Martz depression breaching & filling algorithm may be inadequate, as some single-node depressions may have contributing areas greater than 30 nodes in width, which we have chosen as an upper limit due to running times becoming impractically high when contributing areas grow too large. A comparison of the Gleyzer algorithm applied to raw terrain, and to terrain preprocessed to remove closed depressions, shows that some higher order streams grow longer and surpass a local minimum, which may be attributed to said node being successfully filled by the preprocessing algorithm (Figure 46).

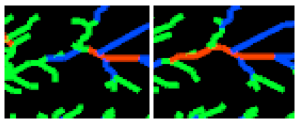


Figure 46: Example of the influence of a specific local minimum on Gleyzer algorithm results before (left) and after (right) applying the depression breaching algorithm. The order 3 stream (red) grows beyond the local minimum where it originally ended, suggesting that the elevation of the local minimum was raised through the depression breaching algorithm.

5.5 Terrace visualization

Figure 47 shows the results of the normal vector similarity algorithm applied on raw terrain. We set an angle threshold of 1 degree, and as our reference triangle, we select a perfectly flat triangle located in the ocean (leftmost region of DEM), allowing us to visualize all flat regions within the DEM.

Figure 48 shows the results under the same conditions, applied to fully preprocessed terrain. A visual comparison of the results presented by both figures highlights the effectiveness of the Garbrecht-Martz flat resolution algorithm in altering the elevation of nodes within flat areas of the DEM.

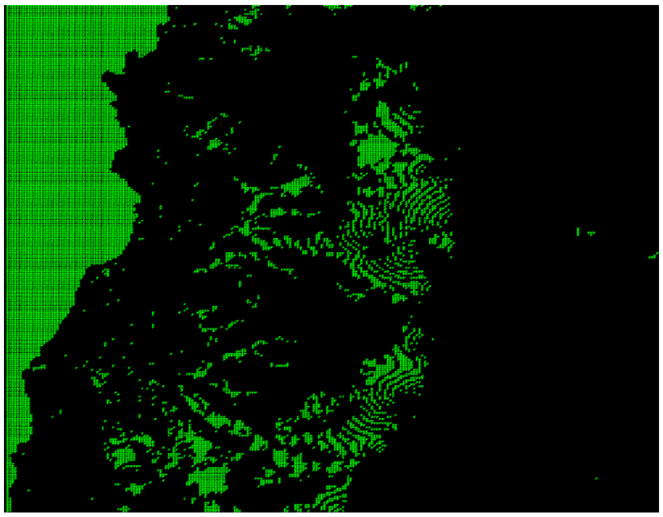


Figure 47: Results of the normal vector similarity algorithm for terrace visualization applied with an angle threshold of 1 degree on a perfectly flat reference triangle. The image shows all approximately flat zones within the DEM highlighted in green.

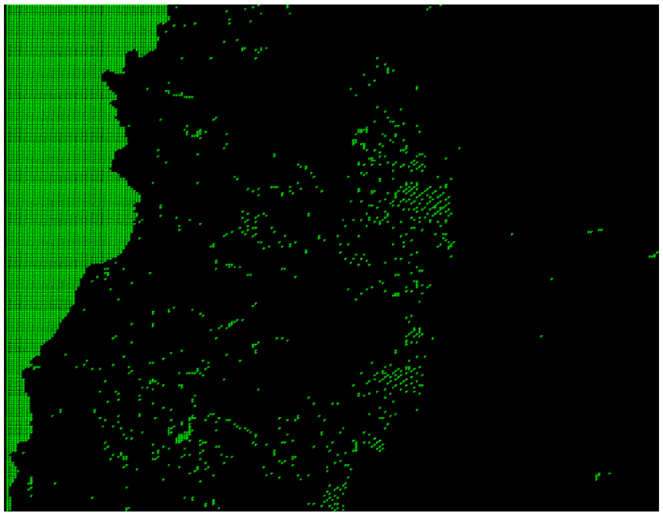


Figure 48: Results of the normal vector similarity algorithm for terrace visualization applied with an angle threshold of 1 degree on a perfectly flat reference triangle in the ocean, with terrain fully preprocessed to remove closed depressions and flats (on land).

5.6 High resolution DEM

As shown in Figures 49 and 50, the currently implemented visualization mechanisms for the Peucker and O'Callaghan algorithms are inadequate for DEMs of larger sizes.

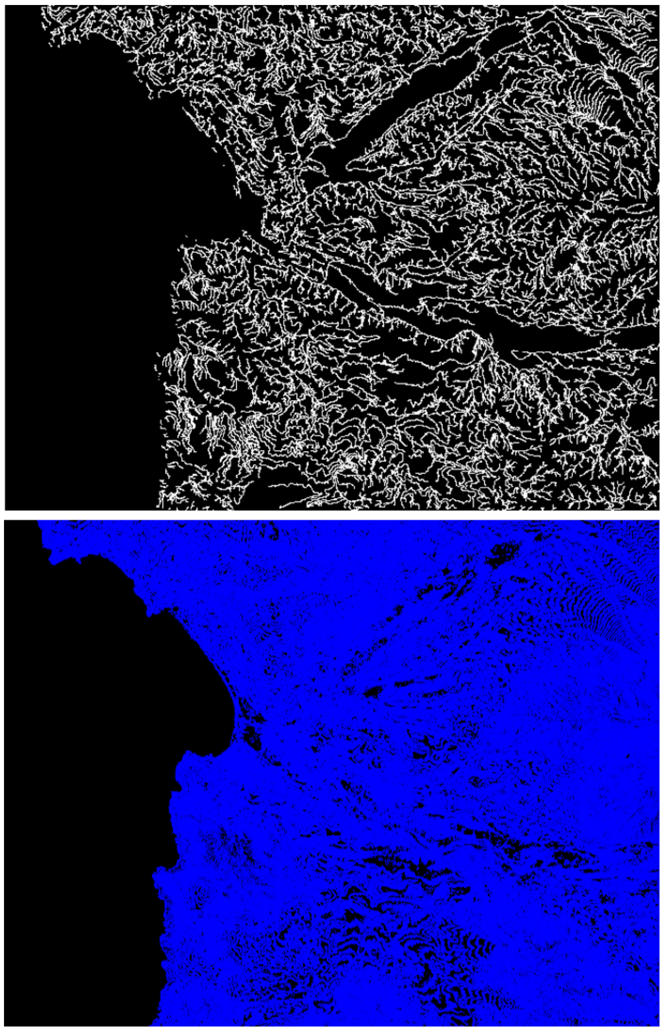


Figure 49: Drainage networks extracted by RiverTools (top), and by the Peucker algorithm in Runnel (bottom), on the 4030x3080 DEM covering a zone within the Petorca province.

As the Peucker algorithm only determines whether nodes belong to the network or not, it is not possible to filter nodes based on water accumulation values. Hence, due to the high density of nodes within the visualization, most of the image is shown in blue, making it difficult to identify individual streams. This problem may be overcome by filtering streams by Strahler order, as done by RiverTools, which would require a correct implementation of the previously discussed Gleyzer algorithm.

With respect to the O'Callaghan algorithm, although a water level filtering mechanism is present, the lines drawn by Runnel are too small to be displayed adequately with a high node density, resulting in a visually cluttered and disjointed network that does not allow us to recognize significant streams within the region.

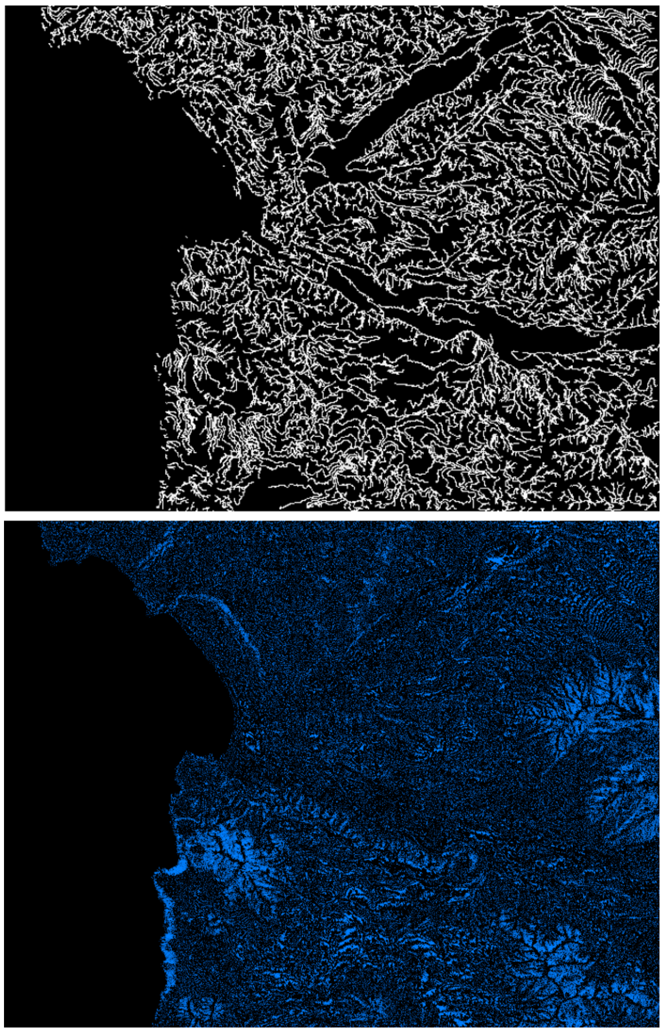


Figure 50: Drainage networks extracted by RiverTools (top), and by the O'Callaghan algorithm in Runnel (bottom), on the 4030x3080 DEM covering a zone within the Petorca province.

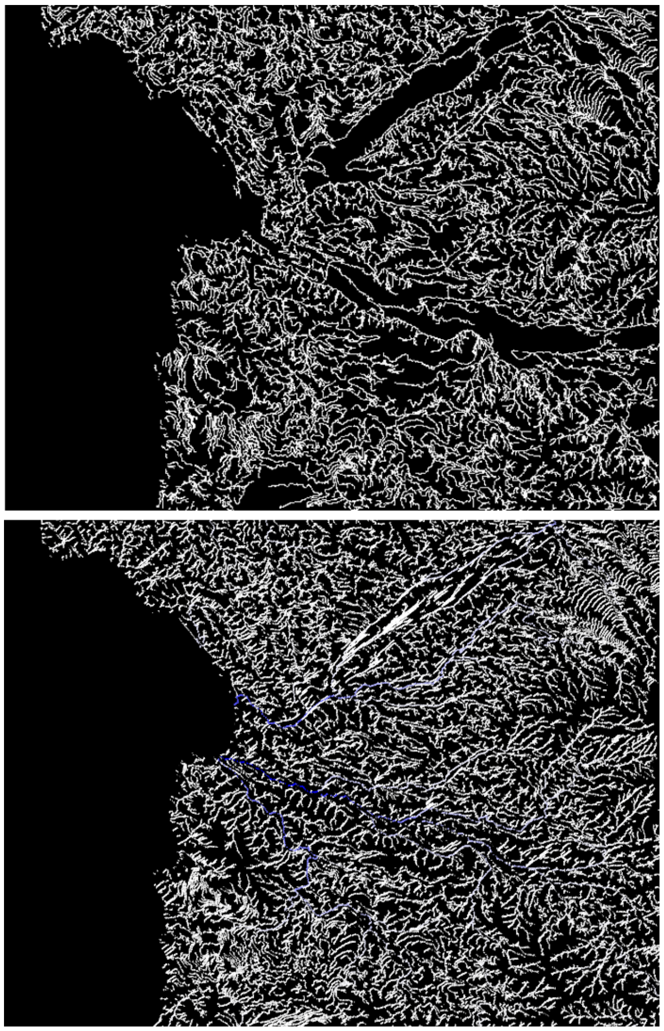


Figure 51: Drainage networks extracted by RiverTools (top), and by the RWFlood algorithm in Runnel (bottom), on the 4030x3080 DEM covering a zone within the Petorca province.

The RWFlood visualization shown in Figure 51, filtered by water levels, allows for better visual assessment of the constructed drainage network. In this figure, three major streams are depicted in blue hue within the region. These streams appear to roughly correspond to the rivers depicted by the Military Geographic Institute reference map from Figure 52.

The drainage networks generated by both programs are visually similar in upstream zones (right side on each image), with the most significant difference being the presence of large areas devoid of drainage in RiverTools results. This can be attributed to the difference in filtering mechanisms used, as the RiverTools results depicted have been filtered by Strahler order, whereas the Runnel results are filtered by water level. As water level is calculated nodewise, a stream of low order stretching over a long sequence of nodes may reach a flow accumulation value that surpasses the filtering threshold, allowing it to be displayed in Runnel yet remaining hidden in RiverTools.

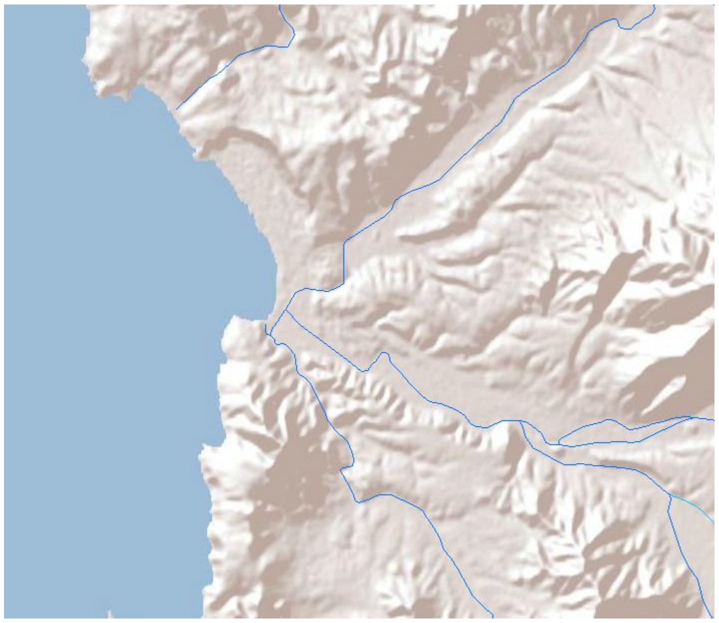


Figure 52: Map produced by the Military Geographic Institute of Chile, depicting major streams located in the area of Petorca province covered by the 4030x3080 DEM.

5.7 Execution time

Figure 53 shows execution times in seconds for the Peucker, O'Callaghan and RWFlood drainage algorithms, with a varying amount of samples per DEM axis (250, 500, 1000, 2000 and 4000 samples per axis), on the high resolution DEM covering the Petorca province. Similarly, Figure 54 shows running times up to 2000 samples per axis for the Garbrecht-Martz imposed gradients and depression breaching preprocessing algorithms applied on the same DEM. All tests were performed on an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz computer with 32 GB of RAM, 64-bit Ubuntu Linux and a GeForce GTX 680 GPU.

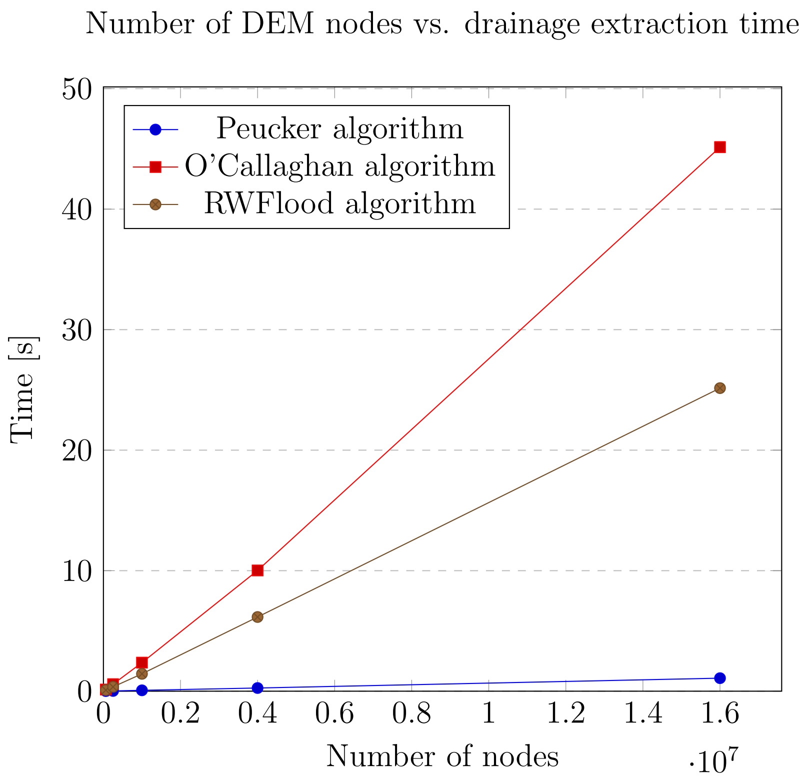


Figure 53: Plot depicting the execution time of the Peucker, O'Callaghan and RWFlood drainage extraction algorithms as a function of DEM size in nodes.

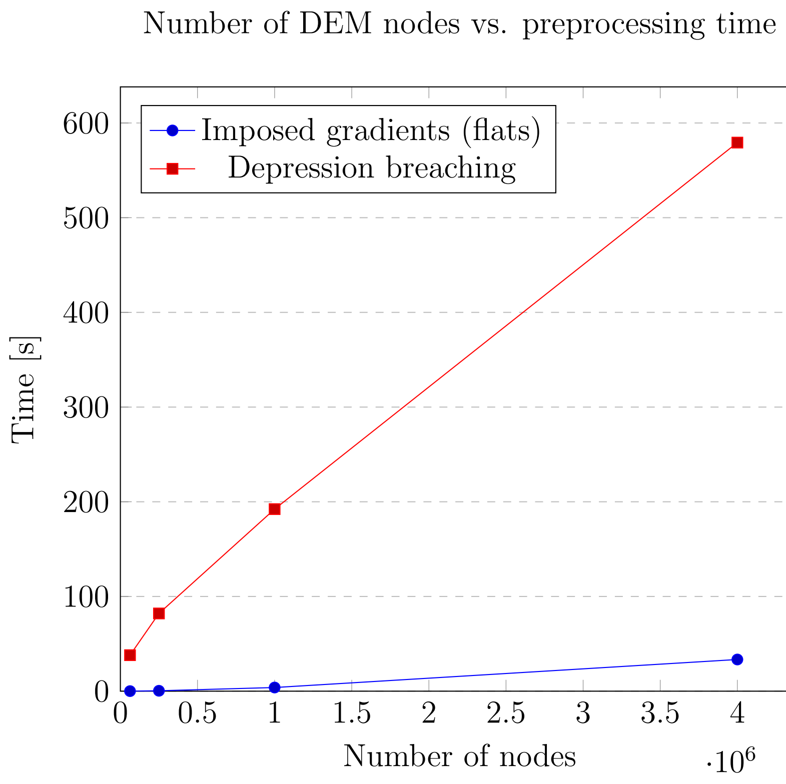


Figure 54: Plot depicting the execution time of the Garbrecht-Martz imposed gradients and depression breaching algorithms as a function of DEM size in nodes.

As shown in Figure 53, all three drainage algorithms run in linear time in terms of the DEM size (measured in nodes). The Peucker algorithm runs very fast compared to the other two algorithms. The RWFlood also provides fast results by comparison with the O'Callaghan algorithm, showing that RWFlood is both efficient and, as previously discussed, effective.

The preprocessing algorithms in Figure 54 run significantly slower than the drainage network algorithms, as they perform multiple passes over all nodes within each area to be modified. The depression breaching algorithm is the slowest of all algorithms described in our work, as determining the full extent of the contributing area for an isolated depression entails repeating computations for every window size up until the correct size that encompasses the entire area.

6 Conclusions

In our work we have successfully addressed several key concerns with the base version of Runnel, as presented in Chapters 1 & 3. Particularly, with respect to the completion of our stated goals and objectives, we have been able to improve and extend the software in terms of drainage network extraction, preprocessing algorithms and terrace recognition features.

Our implementation of preprocessing algorithms appears to work as intended, allowing drainage network extraction algorithms to obtain better results. Concerning drainage network extraction itself, we are inclined to believe that the RWFlood algorithm, coupled with the preprocessing algorithms implemented, produces acceptable results by visual comparison with the RiverTools software.

However, the abundance of isolated nodes and disconnected segments within the drainage networks extracted by Runnel show that a postprocessing algorithm may also be necessary to build a fully connected drainage tree for the DEM. A connected drainage tree is a prerequisite for any drainage analysis algorithms that compute geomorphologically relevant metrics and indicators (e.g. hypsometric curves, basin ratios, surface area drained per stream, etc.) from a drainage network.

Further work in this regard may involve the selection or definition of appropriate similarity metrics to compare drainage networks formally, rather than based on visual assessment. Challenges to such an approach may involve, for instance, definining a sensible weighting scheme for segments of different Strahler order within the drainage tree, as similarity between major streams should be more relevant than between order 1 or 2 streams.

Regarding the Gleyzer algorithm, our work at this stage remains unsatisfactory and unusable. It is necessary to evaluate whether the Garbrecht-Martz depression breaching implementation can be improved to eliminate the 30-node contributing area constraint without significantly increasing running time, and if Gleyzer algorithm results are effectively enhanced in this way.

With respect to terrace visualization, the proposed algorithm works as intended, allowing the user to view all sloping zones within the threshold specified. It remains to be seen whether this tool is helpful at locating geological terraces in DEMs. A high resolution DEM of an area whose terraces are known beforehand may be necessary to evaluate this aspect. A potential improvement to this tool, especially for larger DEMs, may lie in allowing the user to perform vector similarity searches either globally or within a contiguous region.

7 Future work

In order to improve the Runnel software and perform a detailed, formal validation of its results, multiple issues need to be addressed.

We identify the following three main problems in our work that must be solved:

- Support for elevation data expressed in meters: The Qt mechanism employed by Runnel to load GeoTIFF files only supports unsigned 8-bit greyscale elevation data, and therefore, elevation data must be scaled to the 0-255 integer range. In areas of high relief (i.e. with a pronounced difference between the highest and lowest elevations), such as in the central Chile DEM used in our work, a single integer value may represent all elevations within a range of dozens of meters. This may lead to the generation of artificial flat surfaces in zones where elevation differences are relatively small and lie entirely within these ranges. Additionally, analysis algorithms that perform node operations in order of increasing or decreasing elevation (e.g. RWFlood) may potentially produce very different results if the full range of elevations is used, rather than a constrained 0-255 range. It is necessary, thus, to allow for elevation data in meters (e.g. 16-bit) in order to achieve greater accuracy.
- Corrected implementation of the Gleyzer algorithm for Strahler order calculation: As previously discussed, the results provided by our implementation of the Gleyzer algorithm are insufficient in that they do not allow us to properly calculate Strahler orders for the entire drainage network, thus remaining unusable. As Strahler orders are an important metric to gauge the relevance of a specific stream within a basin, developing a correct mechanism to compute these values may allow for additional analysis and visualization functions (e.g. filtering streams by Strahler order, or identifying the highest Strahler order of a subnetwork).
- Postprocessing algorithm to construct a fully connected drainage tree: As shown in our results, the currently implemented drainage extraction algorithms often do not generate a full drainage tree for the entire region, producing disjointed networks instead. A postprocessing algorithm may be necessary to link disconnected streams and isolated nodes into a drainage tree.

For further validation of the generated results, it may be desirable to define a similarity metric in order to compare networks formally, rather than according to visual assessment. To define such a metric, several aspects must be considered:

- Drainage networks generated by RiverTools are only available as images, with the actual data being stored in a proprietary format. Therefore, RiverTools drainage trees must be reconstructed from said images.
- As RiverTools prunes network branches that form cycles, it may be necessary to remove cycles from Runnel results as well for drainage trees to be comparable.
- It may be possible to consider two drainage trees similar even if Strahler order 1 branches differ significantly, as long as streams of higher Strahler order are similar in both. In that

case, it may be necessary to employ a weighting scheme for graph edges that gives streams of higher order a greater impact on the similarity metric used.

- While two tree graphs may be similar in terms of their topology (total nodes, in/out degrees, etc.), it is also important to consider the geographic location of their nodes. For example, a certain major stream may be present in the topology of both trees, but might have different direction or geographical starting point. Thus, it may be necessary to consider the geographic coordinates of graph nodes, in addition to graph topology itself, in order to construct an adequate similarity metric.

Other ways in which the software may be improved are as follows:

- Design and implementation of large DEM support: Managing DEMs in internal memory, as done by Runnel, is inadequate and insufficient for very high resolution DEMs, which may be too large to be visualized or loaded completely into memory. Hence, Runnel may be improved in this regard by designing and implementing a mechanism to manage large DEMs by either of the two approaches described in Chapter 2: reducing memory usage through the application of succinct data structures, or usage of external memory structures and algorithms. We believe succinct data structures to be preferable as a first approach, as the external memory approach would involve a significant performance hit that would be best avoided when possible, and adapting preprocessing algorithms to external memory is likely to require a very careful design to minimize disk accesses.
- Implementation of other drainage analysis techniques (hypsometric analysis, drainage area, etc.): Aside from drainage network extraction, RiverTools and other GIS's provide additional analysis tools on constructed networks. These techniques include, but are not limited to, determining the surface area drained by a stream, identifying basins and sub-basins and computing the hypsometric curve of a basin. It is desirable, therefore, for the Runnel software to be extended with such functionality. Proper calculation of drainage networks is, however, a prerequisite.
- User experience improvements: The user interface provided by Runnel is inconsistent with that of other GIS software used in fluvial geomorphology (e.g. RiverTools, ArcGIS, GRASS GIS). While we do not recommend imitating the interface of other commercial software, we believe that leveraging the familiarity of instructors and students with similar software may prove to be beneficial for the eventual adoption of Runnel as an educational tool. The current interface also contains some software defects that lead to unexpected program failures, which need to be addressed.

8 Appendix A: Code listings

This appendix contains relevant code listings discussed throughout our work (particularly in Chapter 4). Full source code is available at `https://bitbucket.org/dgajardo/runnel`.

8.1 RWFlood algorithm

Listing 1: RWFlood algorithm class header file

```
#ifndef RWFLOODALGORITHM_H
#define RWFLOODALGORITHM_H
#include "drainagealgorithms.h"
#include <painters/shaders/shaderrwflood.h>
#include <UI/drainageAlgorithm/rwfloodconf.h>

class RWFloodAlgorithm : public DrainageAlgorithms
{
    Q_OBJECT
public:
    RWFloodAlgorithm();
    ~RWFloodAlgorithm();
    virtual void run(Terrain *ter);
    virtual void glewReady();
    virtual void render(glm::mat4 matrix, float exag_z, glm::vec3
       color);
    virtual QString getName();
    virtual QWidget* getConf();
    virtual std::vector<glm::vec3> getPathTree();

private:
    Terrain* ter;
    ShaderRWFlood* shader;
    RWFloodConf conf;
    int maxWaterCount;
    int waterThreshold;
    std::vector<glm::vec3> drainagePoints;
    std::vector<glm::vec3> drainageColors;
    enum Direction {
        TOP_LEFT,
        TOP,
        TOP_RIGHT,
        LEFT,
        RIGHT,
        BOTTOM_LEFT,
        BOTTOM,
        BOTTOM_RIGHT
    };

    void flood(std::vector<runnel::Point*>&);
    void calculateWaterAccumulation(std::vector<runnel::Point*>&);
    void getDrainagePoints();

    bool initializeDirection(std::vector<runnel::Point*>&,runnel::
       Point*);
    std::vector<runnel::Point*> computeNeighborhood(runnel::Point*)
       ;
    void setDirectionTowardsAdjacentPoint(std::vector<runnel::Point
       *>&,
                                          runnel::Point*,
                                          runnel::Point*);
    bool isDirectedOutsideTerrainBoundary(runnel::Point*);
    int getNextPointId(runnel::Point*);

public slots:
    void changeAttr();
};
#endif // RWFLOODALGORITHM_H
```

Listing 2: flood function in RWFloodAlgorithm.cpp

```
void RWFloodAlgorithm::flood(std::vector<runnel::Point*>& points)
{
    const int maxElev = (int)(ter->max_bounding.z);
    const int minElev = (int)(ter->min_bounding.z);
    const std::size_t arraySize = maxElev - minElev + 1;

    std::vector<std::queue<runnel::Point*>> queueArray(arraySize,
                                                  std::queue<
                                                     runnel::
                                                     Point*>()
                                                     );
    std::map<int, float> zValueStore;

    for (runnel::Point* point : points) {
        point->flags = 0;
        bool isBoundaryPoint = initializeDirection(points, point);
        if (isBoundaryPoint) {
            queueArray[point->coord.z - minElev].push(point);
        }
    }

    for (int z = minElev; z <= maxElev; ++z) {
        while (!queueArray[z-minElev].empty()) {
            runnel::Point* point = queueArray[z-minElev].front();
            queueArray[z-minElev].pop();

            std::vector<runnel::Point*> neighborhood =
               computeNeighborhood(point);
            for (runnel::Point* neighbor : neighborhood) {
                if (neighbor->flags==0) {
                    setDirectionTowardsAdjacentPoint(points,
                       neighbor, point);
                    if (neighbor->coord.z < z) {
                        /* Warning: this modifies the data!
                         * The original values are saved for later
                           restoration. */
                        zValueStore[neighbor->ident] = neighbor->
                           coord.z;
                        neighbor->coord.z = z;
                    }
                    queueArray[neighbor->coord.z - minElev].push(
                       neighbor);
                }
            }
        }
    }

    // Restore modified z-coordinate values to their original
       values.
    for (auto &entry : zValueStore) {
        points[entry.first]->coord.z = entry.second;
    }
}
```

Listing 3: calculateWaterAccumulation function in RWFloodAlgorithm.cpp

```
void RWFloodAlgorithm::calculateWaterAccumulation(std::vector<
   runnel::Point*>& points)
{
    int maxWaterCount = 1;
    std::vector<int> inboundDegree(points.size());
    std::vector<bool> pointIsVisited(points.size());

    for (runnel::Point* point : points) {
        points[point->ident]->water_value = 1;
        inboundDegree[point->ident] = 0;
        pointIsVisited[point->ident] = false;
        point->water_parent.clear();
    }
    for (runnel::Point* point : points) {
        if (!isDirectedOutsideTerrainBoundary(point)) {
            inboundDegree[getNextPointId(point)]++;
        }
    }

    for (runnel::Point* point : points) {
        if (!pointIsVisited[point->ident]) {
            runnel::Point* currentPoint = point;
            while (inboundDegree[currentPoint->ident] == 0) {
                pointIsVisited[currentPoint->ident] = true;
                if (isDirectedOutsideTerrainBoundary(currentPoint))
                    break;
                int nextPointId = getNextPointId(currentPoint);
                int currentPointId = currentPoint->ident;
                points[nextPointId]->water_value += points[
                   currentPointId]->water_value;
                points[nextPointId]->water_parent.push_back(points[
                   currentPointId]);
                if (points[nextPointId]->water_value >
                   maxWaterCount) {
                    maxWaterCount = points[nextPointId]->
                       water_value;
                }
                inboundDegree[nextPointId]--;
                currentPoint = points[nextPointId];
            }
        }
    }

    this->maxWaterCount = maxWaterCount;
}
```

8.2 Gleyzer algorithm

Listing 4: getNumberStrahlerHorton function in arbol.cpp

```
void arbol::getNumberStrahlerHorton(){
    if(hijos.size() == 0){
        number_strahler_horton = 1;
        return;
    }

    std::unordered_map<int, int> ordenes;
    for(arbol*h : hijos){
        if(h->number_strahler_horton == NO_NUMBER_STRAHLER_HORTON){
            h->getNumberStrahlerHorton();
        }
        ordenes[h->number_strahler_horton] +=1;
    }
    int mayor_ord = 0;
    for( auto ord : ordenes){
        if(mayor_ord < ord.first){
            mayor_ord = ord.first;
        }
    }
    if(ordenes[mayor_ord] > 1){
        number_strahler_horton = mayor_ord + 1;
    }else{
        number_strahler_horton = mayor_ord;
    }
}
```

Listing 5: computeNetworkStrahlerOrdering function in arbol.cpp

```
void arbol::computeNetworkStrahlerOrdering()
{
    std::vector<runnel::Point*> edges;
    this->getArbolEdges(edges);
    // EdgeList type: std::vector<std::pair<runnel::Point*, runnel
       ::Point*>>
    EdgeList edgeList = makeEdgeList(edges);
    std::map<int, bool> visitedEdges;
    std::map<int, std::vector<int>> inflowingEdgesPerNode =
       makeInflowingEdgeMap(edgeList);
    std::map<int, int> upstreamNodePerEdge =
       makeUpstreamNodePerEdgeMap(edgeList);
    std::map<int, int> streamOrders;
    std::map<int, int> originatingNode(upstreamNodePerEdge); //
       Must be initialized to each edge's upstream node.

    for (int i = 0; i < edgeList.size(); i++) {
        if (!visitedEdges[i]) {
            streamOrdering(i, visitedEdges, inflowingEdgesPerNode,
               upstreamNodePerEdge, streamOrders, originatingNode);
        }
    }

    this->updateStrahlerOrder(edgeList, streamOrders);
}
```

Listing 6: makeInflowingEdgeMap function in arbol.cpp

```
std::map<int, std::vector<int>> arbol::makeInflowingEdgeMap(
   EdgeList &edgeList)
{
    std::map<int, std::vector<int>> inflowingEdgeMap;

    for (auto iter = edgeList.begin(); iter != edgeList.end(); std
       ::advance(iter, 1)) {
        runnel::Point* p1 = (*iter).first;
        runnel::Point* p2 = (*iter).second;
        if (p1->coord.z < p2->coord.z) {
            // We save the edge's index in edgeList, not the point'
               s ID!
            inflowingEdgeMap[p1->ident].push_back(iter - edgeList.
               begin());
        } else {
            inflowingEdgeMap[p2->ident].push_back(iter - edgeList.
               begin());
        }
    }

    return inflowingEdgeMap;
}
```

Listing 7: makeUpstreamNodePerEdgeMap function in arbol.cpp

```
std::map<int, int> arbol::makeUpstreamNodePerEdgeMap(EdgeList &
   edgeList)
{
    std::map<int, int> upstreamNodePerEdge;

    for (auto iter = edgeList.begin(); iter != edgeList.end(); std
       ::advance(iter, 1)) {
        runnel::Point* p1 = (*iter).first;
        runnel::Point* p2 = (*iter).second;
        if (p1->coord.z < p2->coord.z) {
            upstreamNodePerEdge[iter - edgeList.begin()] = p2->
               ident;
        } else {
            upstreamNodePerEdge[iter - edgeList.begin()] = p1->
               ident;
        }
    }

    return upstreamNodePerEdge;
}
```

Listing 8: streamOrdering function in arbol.cpp

```
int arbol::streamOrdering(int edgeIndex, std::map<int, bool> &
   visitedEdges,
                          std::map<int, std::vector<int>> &
                             inflowingEdgesPerNode,
                          std::map<int, int> &upstreamNodePerEdge,
                          std::map<int, int> &streamOrders,
                          std::map<int, int> &originatingNode)
{
    visitedEdges[edgeIndex] = true;

    if (inflowingEdgesPerNode[upstreamNodePerEdge[edgeIndex]].size
       () == 0) {
        streamOrders[edgeIndex] = 1;
    } else {
        std::map<int, std::pair<int,int>> upstreamOrders;

        for (int inflowingEdgeIndex : inflowingEdgesPerNode[
           upstreamNodePerEdge[edgeIndex]]) {
            if (!visitedEdges[inflowingEdgeIndex]) {
                upstreamOrders[inflowingEdgeIndex] = std::make_pair
                   (streamOrdering(inflowingEdgeIndex, visitedEdges
                   , inflowingEdgesPerNode, upstreamNodePerEdge,
                   streamOrders, originatingNode), originatingNode[
                   inflowingEdgeIndex]);
            } else {
                upstreamOrders[inflowingEdgeIndex] = std::make_pair
                   (streamOrders[inflowingEdgeIndex],
                   originatingNode[inflowingEdgeIndex]);
            }
        }

        int maxOrder = 0;
        int maxOrderCount = 0;
        int maxOrderOrigin = -1;

        for (auto iter = upstreamOrders.begin(); iter !=
           upstreamOrders.end(); std::advance(iter, 1)) {
            std::pair<int,int> orderOriginPair = (*iter).second;
            int order = orderOriginPair.first;
            int origin = orderOriginPair.second;

            if (order > maxOrder) {
                maxOrder = order;
                maxOrderCount = 1;
                maxOrderOrigin = origin;
            } else if (order == maxOrder) {
                if (origin != maxOrderOrigin) {
                    maxOrderCount += 1;
                }
            }
        }

        if (maxOrderCount > 1) {
            streamOrders[edgeIndex] = maxOrder + 1;
            originatingNode[edgeIndex] = upstreamNodePerEdge[
               edgeIndex];
        } else {
            streamOrders[edgeIndex] = maxOrder;
            originatingNode[edgeIndex] = maxOrderOrigin;
        }
    }

    return streamOrders[edgeIndex];
}
```

8.3 Garbrecht-Martz flat resolution algorithm

Listing 9: extractFlatIds function in garbrechtmartz.cpp

```
std::set<int> GarbrechtMartz::extractFlatIds()
{
    std::set<int> flatIds;
    int width = ter->width;

    for (runnel::Point* point : ter->struct_point) {
        if (point->coord.z == 0) {
            // No-data point. Do not consider as part of a flat
               area.
            continue;
        }

        int id = point->ident;
        bool hasGradient = false;
        std::set<int> localFlatIds;

        // We define a 2x2 point region to be the minimal unit of a
           flat area.
        for (int i = 0; i <= 1 && !hasGradient; ++i) {
            for (int j = 0; j <= 1 && !hasGradient; ++j) {
                localFlatIds.insert(id);
                int neighborIndex = id + j*width + i;
                if (neighborIndexIsOutOfRange(id, neighborIndex)) {
                        continue;
                    }
                    if (ter->struct_point[id]->coord.z !=
                            ter->struct_point[neighborIndex]->coord.z)
                               {
                        hasGradient = true;
                    } else {
                        localFlatIds.insert(neighborIndex);
                    }
                }
            }

            if (!hasGradient) {
                for (int flatId : localFlatIds) {
                    flatIds.insert(flatId);
                }
            }
        }

        return flatIds;
}
```

Listing 10: run function in garbrechtmartz.cpp

```
void GarbrechtMartz::run(Terrain* ter)
{
    this->ter = ter;
    std::set<int> flatIds = extractFlatIds();
    std::unordered_map<int,int> towardsLower =
       gradientTowardsLowerTerrain(flatIds);
    std::unordered_map<int,int> awayFromHigher =
       gradientAwayFromHigherTerrain(flatIds);
    std::unordered_map<int,int> combinedGradient = combineGradients
       (towardsLower, awayFromHigher);
    std::set<int> halfIncrementIds = findHalfIncrementIds(
       combinedGradient, towardsLower, awayFromHigher);
    applyIncrements(combinedGradient, halfIncrementIds);
}
```

Listing 11: gradientTowardsLowerTerrain function in garbrechtmartz.cpp

```
std::unordered_map<int,int> GarbrechtMartz::
   gradientTowardsLowerTerrain(std::set<int> flatIds)
{
    int width = ter->width;
    int prevFlatIdCount = -1;
    std::unordered_map<int,int> idIncrementMap;
    std::set<int> flatIdsMarkedForDeletion;
    std::set<int> idsMarkedForDownslopeGradient;
    std::set<int> downslopeGradientIds;

    while (!flatIds.empty()) {
        if ((int)flatIds.size() == prevFlatIdCount) {
            // Remaining flat areas are isolated and cannot be
               resolved by this algorithm.
            break;
        }

        for (int id : flatIds) {
            for (int i = -1; i <= 1; ++i) {
                for (int j = -1; j <= 1; ++j) {
                    int neighborIndex = id + j*width + i;
                    if (neighborIndexIsOutOfRange(id, neighborIndex
                       )) {
                        continue;
                    }
                    if ((downslopeGradientIds.find(neighborIndex)
                       !=
                         downslopeGradientIds.end()) ||
                            (ter->struct_point[id]->coord.z >
                             ter->struct_point[neighborIndex]->
                                coord.z)) {
                        // Point 'id' has a downslope gradient.
                        idsMarkedForDownslopeGradient.insert(id);
                        flatIdsMarkedForDeletion.insert(id);
                    }
                }
            }
        }

        prevFlatIdCount = flatIds.size();

        for (int id : flatIdsMarkedForDeletion) {
            flatIds.erase(id);
        }

        flatIdsMarkedForDeletion.clear();

        for (int id : idsMarkedForDownslopeGradient) {
            downslopeGradientIds.insert(id);
        }

        idsMarkedForDownslopeGradient.clear();

        for (int id : flatIds) {
            /* Increase elevation of remaining flat-belonging
               points
             * without a downslope gradient. */
            idIncrementMap[id]++;
        }
    }

    return idIncrementMap;
}
```

Listing 12: gradientAwayFromHigherTerrain function in garbrechtmartz.cpp

```
std::unordered_map<int, int> GarbrechtMartz::
   gradientAwayFromHigherTerrain(std::set<int> flatIds)
{
    int width = ter->width;
    int prevFlatIdCount = -1;
    std::unordered_map<int,int> idIncrementMap;
    std::set<int> flatIdsMarkedForDeletion;
    std::set<int> idsMarkedForUpslopeGradient;
    std::set<int> upslopeGradientIds;

    while (!flatIds.empty()) {
        if ((int)flatIds.size() == prevFlatIdCount) {
            // Remaining flat areas are isolated and cannot be
               resolved by this algorithm.
            break;
        }

        for (int id : flatIds) {
            bool adjacentToHigherTerrain = false;
            bool adjacentToLowerTerrain = false;

            for (int i = -1; i <= 1 && !adjacentToLowerTerrain; ++i
               ) {
                for (int j = -1; j <= 1 && !adjacentToLowerTerrain;
                    ++j) {
                    int neighborIndex = id + j*width + i;
                    if (neighborIndexIsOutOfRange(id, neighborIndex
                       )) {
                        continue;
                    }
                    if (ter->struct_point[id]->coord.z >
                            ter->struct_point[neighborIndex]->coord
                               .z) {
                        // Point 'id' has a downslope gradient. Do
                           not increment.
                        adjacentToLowerTerrain = true;
                        flatIdsMarkedForDeletion.insert(id);
                    } else if ((upslopeGradientIds.find(
                       neighborIndex) !=
                                upslopeGradientIds.end()) ||
                               (ter->struct_point[id]->coord.z <
                                ter->struct_point[neighborIndex]->
                                   coord.z)) {
                        // Point 'id' has an upslope gradient.
                        adjacentToHigherTerrain = true;
                    }
                }
            }

            if (adjacentToHigherTerrain && !adjacentToLowerTerrain)
                {
                idsMarkedForUpslopeGradient.insert(id);
                flatIdsMarkedForDeletion.insert(id);
            }
        }

        prevFlatIdCount = flatIds.size();

        for (int id : flatIdsMarkedForDeletion) {
            flatIds.erase(id);
        }

        flatIdsMarkedForDeletion.clear();

        for (int id : idsMarkedForUpslopeGradient) {
            upslopeGradientIds.insert(id);
        }

        idsMarkedForUpslopeGradient.clear();

        for (int id : upslopeGradientIds) {
            idIncrementMap[id]++;
        }
    }

    return idIncrementMap;
}
```

Listing 13: combineGradients function in garbrechtmartz.cpp

```
std::unordered_map<int, int> GarbrechtMartz::combineGradients(std::
   unordered_map<int, int> gradientTowardsLowerTerrain, std::
    unordered_map<int, int> gradientAwayFromHigherTerrain)
{
    // Copy first map to new map, using the copy constructor.
    std::unordered_map<int,int> combinedGradient(
        gradientTowardsLowerTerrain);

    // Add second map's contents to the new map.
    for (auto &entry : gradientAwayFromHigherTerrain) {
        int id = entry.first;
        int incrementCount = entry.second;
        combinedGradient[id] += incrementCount;
    }

    return combinedGradient;
}
```

Listing 14: findHalfIncrementIds function in garbrechtmartz.cpp

```
std::set<int> GarbrechtMartz::findHalfIncrementIds(std::
    unordered_map<int, int> combinedGradient, std::unordered_map<int
    , int> towardsLower, std::unordered_map<int, int> awayFromHigher
    )
{
    int width = ter->width;
    std::set<int> halfIncrementIds;

    // Check if gradients on both maps cancel each other.
    for (auto &entry : combinedGradient) {
        int id = entry.first;
        for (int i = -1; i <= 1; ++i) {
            for (int j = -1; j <= 1; ++j) {
                int neighborIndex = id + j*width + i;
                if (neighborIndexIsOutOfRange(id, neighborIndex)) {
                    continue;
                }
                if (towardsLower[id] == awayFromHigher[
                    neighborIndex] &&
                        awayFromHigher[id] == towardsLower[
                            neighborIndex] &&
                        halfIncrementIds.find(neighborIndex) ==
                        halfIncrementIds.end()) {
                    // The gradients cancel each other. Extra half-
                        increment required.
                    halfIncrementIds.insert(id);
                }
            }
        }
    }

    return halfIncrementIds;
}
```

Listing 15: applyIncrements function in garbrechtmartz.cpp

```
void GarbrechtMartz::applyIncrements(std::unordered_map<int,int>
   combinedGradient, std::set<int> halfIncrementIds)
{
    for (auto &entry : combinedGradient) {
        ter->struct_point[entry.first]->coord.z += entry.second*
           elevationIncrement;
    }

    for (int id : halfIncrementIds) {
        ter->struct_point[id]->coord.z += elevationIncrement/2.0;
    }
}
```

8.4 Garbrecht-Martz pit removal algorithm

Listing 16: run function in gmpitremover.cpp

```
void GMPitRemover::run(Terrain *ter)
{
    this->ter = ter;
    std::vector<runnel::Point*> inflowSinks;

    for (runnel::Point *node : ter->struct_point) {
        node->flags = 0;
        if (isInflowSink(node)) {
            inflowSinks.push_back(node);
        }
    }

    for (runnel::Point *inflowSink : inflowSinks) {
        if (inflowSink->flags == 1) {
            continue;
        }
        int windowSize = BASE_WINDOW_SIZE;
        bool potentialOutletFound = false;

        std::vector<runnel::Point*> contributingArea;
        std::vector<runnel::Point*> potentialOutlets;
        runnel::Point* lowestOutlet;
```

```
std::vector<runnel::Point*> potentialBreachingSites;
runnel::Point* breachingSite;

while (!potentialOutletFound) {
    if (windowSize > ter->width || windowSize > ter->height
        || windowSize > 30) {
        break;
    }
    contributingArea = computeContributingArea(inflowSink,
       windowSize, contributingArea);
    potentialOutlets = findPotentialOutlets(
       contributingArea);

    if (potentialOutlets.size() == 0) {
        windowSize += 2;
        continue;
    }

    lowestOutlet = getLowestPotentialOutlet(
       potentialOutlets, contributingArea);

    std::vector<int> windowBoundaries =
       defineWindowBoundaries(inflowSink, windowSize);
    std::vector<int> boundaryContributorIds =
       getBoundaryContributors(windowBoundaries,
       contributingArea);

    if (!hasBoundaryContributorBelowLowestOutlet(
       boundaryContributorIds, lowestOutlet)) {
        potentialOutletFound = true;
    } else {
        windowSize += 2;
    }
}

if (potentialOutletFound == false) {
    continue;
}

if (!containsClosedDepressions(contributingArea,
   lowestOutlet)) {
    for (runnel::Point *node : contributingArea) {
        if (node->coord.z == lowestOutlet->coord.z) {
            flagNode(node);
        }
    }
```

```
        } else {
            potentialBreachingSites = findPotentialBreachingSites(
               contributingArea, lowestOutlet);

            if (potentialBreachingSites.size() == 0) {
                fillDepressions(contributingArea, lowestOutlet);
            } else if (potentialBreachingSites.size() > 1) {
                breachingSite = getSteepestSlopeBreachingSite(
                   potentialBreachingSites, contributingArea);
            } else {
                breachingSite = potentialBreachingSites.front();
            }

            performBreaching(breachingSite, contributingArea);
            fillDepressions(contributingArea, breachingSite);
        }
    }
}
```

Listing 17: computeContributingArea function in gmpitremover.cpp

```
std::vector<runnel::Point*> GMPitRemover::computeContributingArea(
   runnel::Point *centralInflowSink, int windowSize, std::vector<
   runnel::Point*> contributingArea)
{
    std::vector<runnel::Point*> pendingNodes;
    std::vector<int> windowBoundaries = defineWindowBoundaries(
       centralInflowSink, windowSize);

    if (!contributingArea.empty()) {
        std::vector<int> boundaryIds = getBoundaryIds(
           windowBoundaries);
        for (int nodeId : boundaryIds) {
            runnel::Point* node = ter->struct_point[nodeId];
            if (std::find(contributingArea.begin(),
               contributingArea.end(), node) != contributingArea.
               end()) {
                continue;
            }
            for (runnel::Point* neighbor : computeNeighborhood(node
               , windowBoundaries)) {
                if (std::find(contributingArea.begin(),
                   contributingArea.end(), neighbor) !=
                   contributingArea.end() && node->coord.z >=
                   neighbor->coord.z) {
                    pendingNodes.push_back(node);
                    break;
                }
            }
        }
    } else {
        pendingNodes.push_back(centralInflowSink);
    }

    while (!pendingNodes.empty()) {
        std::vector<runnel::Point*> newNodes;

        for (runnel::Point* node : pendingNodes) {
            contributingArea.push_back(node);
            for (runnel::Point* neighbor : computeNeighborhood(node
               , windowBoundaries)) {
                if (neighbor->coord.z >= node->coord.z && std::find
                   (contributingArea.begin(), contributingArea.end
                   (), neighbor) == contributingArea.end() && std::
                   find(pendingNodes.begin(), pendingNodes.end(),
                   neighbor) == pendingNodes.end() && std::find(
                   newNodes.begin(), newNodes.end(), neighbor) ==
                   newNodes.end()) {
                    newNodes.push_back(neighbor);
                }
            }
        }

        pendingNodes = newNodes;
    }

    return contributingArea;
}
```

Listing 18: findPotentialOutlets function in gmpitremover.cpp

```
std::vector<runnel::Point*> GMPitRemover::findPotentialOutlets(std
   ::vector<runnel::Point *> contributingArea)
{
    std::vector<runnel::Point*> potentialOutlets;

    for (runnel::Point* node : contributingArea) {
        for (runnel::Point* neighbor : computeNeighborhood(node)) {
            if (std::find(contributingArea.begin(),
               contributingArea.end(), neighbor) ==
               contributingArea.end() && node->coord.z > neighbor->
               coord.z) {
                potentialOutlets.push_back(node);
                break;
            }
        }
    }

    return potentialOutlets;
}
```

Listing 19: findPotentialBreachingSites function in gmpitremover.cpp

```
std::vector<runnel::Point*> GMPitRemover::
   findPotentialBreachingSites(std::vector<runnel::Point*>
   contributingArea, runnel::Point* lowestOutlet)
{
    // Nodes in contributing area whose elevation == lowestOutlet's
        elevation
    std::vector<runnel::Point*> sameElevContributors;

    // Nodes in contributing area whose elevation < lowestOutlet's
       elevation
    std::vector<runnel::Point*> lowerElevContributors;

    // Nodes in sameElevContributors adjacent to a node: 1) outside
        the
    // contributing area && 2) with elevation < lowestOutlet's
       elevation
    std::vector<runnel::Point*> outflowingBoundaryNodes;

    std::vector<runnel::Point*> potentialBreachingSites;

    std::copy_if(contributingArea.begin(), contributingArea.end(),
       std::back_inserter(sameElevContributors),
                 [lowestOutlet] (runnel::Point* p) {
                     return p->coord.z == lowestOutlet->coord.z;
                 });

    std::copy_if(contributingArea.begin(), contributingArea.end(),
       std::back_inserter(lowerElevContributors),
                 [lowestOutlet] (runnel::Point* p) {
                     return p->coord.z < lowestOutlet->coord.z;
                 });

    for (runnel::Point* node : sameElevContributors) {
        for (runnel::Point* neighbor : computeNeighborhood(node)) {
            if (std::find(contributingArea.begin(),
               contributingArea.end(), neighbor) ==
               contributingArea.end() && neighbor->coord.z <
               lowestOutlet->coord.z) {
                outflowingBoundaryNodes.push_back(node);
                break;
            }
        }
    }

    for (runnel::Point* node : outflowingBoundaryNodes) {
        for (runnel::Point* lowerContributor :
           lowerElevContributors) {
            if (nodeDistance(node, lowerContributor) <=
               MAX_BREACHING_LENGTH) {
                potentialBreachingSites.push_back(node);
                break;
            }
        }
    }

    return potentialBreachingSites;
}
```

Listing 20: performBreaching function in gmpitremover.cpp

```
void GMPitRemover::performBreaching(runnel::Point *breachingSite,
   std::vector<runnel::Point*> contributingArea)
{
    std::vector<runnel::Point*> lowerElevContributors;

    std::copy_if(contributingArea.begin(), contributingArea.end(),
                 std::back_inserter(lowerElevContributors),
                 [breachingSite] (runnel::Point* p) {
                     return p->coord.z < breachingSite->coord.z;
                 });

    std::vector<int> breachingLengths;

    for (runnel::Point* lowerContributor : lowerElevContributors) {
        breachingLengths.push_back(nodeDistance(breachingSite,
           lowerContributor));
    }

    int breachingLength = breachingLengths.empty() ?
                0 :
                *std::min_element(breachingLengths.begin(),
                                          breachingLengths.end());

    // We identify the neighbor to which flow should be directed
       from
    // the breaching site, and we lower the latter's elevation to
       match it

    for (runnel::Point* neighbor : computeNeighborhood(
       breachingSite)) {
        if (std::find(contributingArea.begin(),
                      contributingArea.end(),
                      neighbor) == contributingArea.end() &&
                neighbor->coord.z < breachingSite->coord.z) {
            breachingSite->coord.z = neighbor->coord.z;
            break;
        }
    }

    if (breachingLength > 1) {
        // We find the next node to be breached
        std::vector<int> contributingAreaDistance;

        for (runnel::Point* contributor : contributingArea) {
            contributingAreaDistance.push_back(nodeDistance(
               breachingSite, contributor));
        }

        // If there is more than one node with min. Chebyshev
           distance, we
        // arbitrarily pick the first one we encounter
        auto minDistanceIter = std::min_element(
           contributingAreaDistance.begin(),
           contributingAreaDistance.end());
        int minDistanceIndex = minDistanceIter -
           contributingAreaDistance.begin();
        contributingArea[minDistanceIndex]->coord.z = breachingSite
           ->coord.z;
    }
}
```

Listing 21: nodeDistance function in gmpitremover.cpp

```
int GMPitRemover::nodeDistance(runnel::Point *p1, runnel::Point *p2
   )
{
    // Compute the Chebyshev/chessboard distance between p1 and p2
       on the grid.
    int x1 = p1->ident % ter->width;
    int x2 = p2->ident % ter->width;
    int y1 = p1->ident / ter->width;
    int y2 = p2->ident / ter->width;
    return glm::max(glm::abs(x1-x2), glm::abs(y1-y2));
}
```

8.5 Peucker algorithm fix

Listing 22: calculateGrid function in peuckerdrainagenetwork.cpp

```
void PeuckerDrainageNetwork::calculateGrid(Terrain *ter){
    int width = ter->width;
    int height = ter->height;

    for (runnel::Point* point : ter->struct_point) {
        point->flags = 0;
    }

    for(unsigned int i = 0; i < ter->struct_point.size() ; ++i){
        int fila = i/width;

        if ( (i+1) % width == 0){
            continue;
        }
        if( fila == (height-1)){
            break;
        }
        runnel::Point *points[4];
        points[0] = ter->struct_point[i];
        points[1] = ter->struct_point[i + 1];
        points[2] = ter->struct_point[i + width];
        points[3] = ter->struct_point[i + width + 1];

        if(!points[0] || !points[1] || !points[2] || !points[3]){
            continue;
        }

        runnel::Point *max_point = points[0];
        for(runnel::Point* pto: points){
            if(pto->coord.z > max_point->coord.z){
                max_point = pto;
            }
        }

        for(runnel::Point* pto: points){
            if(pto->coord.z == max_point->coord.z){
                max_point->setFlagsOn(runnel::Point::PEUCKER);
            }
        }
    }
}
```

Listing 23: Software defect fix for calculateGrid function in peuckerdrainagenetwork.cpp

```
for(runnel::Point* pto: points){
    if(pto->coord.z == max_point->coord.z){
        pto->setFlagsOn(runnel::Point::PEUCKER);
    }
}
```

8.6 Normal vector similarity algorithm for terrace detection

Listing 24: Header file for TerraceDetectionAlgorithm virtual class

```
#ifndef TERRACEDETECTIONALGORITHM_H
#define TERRACEDETECTIONALGORITHM_H
#include <QWidget>
#include "terrain.h"
#include "lib/glm/glm.hpp"

class TerraceDetectionAlgorithm : public QObject
{
    Q_OBJECT
    public:
        TerraceDetectionAlgorithm();
        virtual ~TerraceDetectionAlgorithm();
        virtual void run(glm::vec3 coords, Terrain *ter) = 0;
        virtual void render(glm::mat4 matrix, float exag_z, glm::
           vec3 color) = 0;
        virtual void glewReady() = 0;
        virtual QString getName() = 0;
        virtual QWidget* getConf() = 0;

   signals:
        void reload();
};

#endif // TERRACEDETECTIONALGORITHM_H
```

Listing 25: Header file for NormalVectorSimilarityAlgorithm class

```
#ifndef NORMALVECTORSIMILARITYALGORITHM_H
#define NORMALVECTORSIMILARITYALGORITHM_H
#include "terracedetectionalgorithm.h"
```

```
#include "UI/terraceDetectionAlgorithm/normalvectorsimilarityconf.h
   "

#include <painters/shaders/shadernormalvectorsimilarity.h>

class NormalVectorSimilarityAlgorithm : public
   TerraceDetectionAlgorithm
{
    Q_OBJECT
    public:
        NormalVectorSimilarityAlgorithm();
        virtual ~NormalVectorSimilarityAlgorithm();
        virtual void run(glm::vec3 point, Terrain *terr);
        virtual void render(glm::mat4 matrix, float exag_z, glm::
           vec3 color);
        virtual void glewReady();
        virtual QString getName();
        virtual QWidget* getConf();

    private:
        Terrain* ter;
        runnel::Triangle* baseTriangle;
        float angleThreshold;
        glm::vec3 clickedPoint;
        std::vector<glm::vec3> terraceVertices;
        NormalVectorSimilarityConf conf;
        ShaderNormalVectorSimilarity* shader;

    public slots:
        void changeAttr();
};

#endif // NORMALVECTORSIMILARITYALGORITHM_H
```

Listing 26: Implementation of NormalVectorSimilarityAlgorithm class

```
#include "normalvectorsimilarityalgorithm.h"

NormalVectorSimilarityAlgorithm::NormalVectorSimilarityAlgorithm():
    TerraceDetectionAlgorithm()
{
    ter = 0;
    baseTriangle = 0;
    clickedPoint = glm::vec3(0.0f,0.0f,0.0f);
    shader = 0;

    QObject::connect(&conf, SIGNAL(changeAttr()), this, SLOT(
        changeAttr()));
}

NormalVectorSimilarityAlgorithm::~NormalVectorSimilarityAlgorithm()
{
    if (shader) {
        delete shader;
    }
}

void NormalVectorSimilarityAlgorithm::run(glm::vec3 point, Terrain
   *terr){
    this->clickedPoint = point;
    this->ter = terr;

    terraceVertices.clear();

    baseTriangle = this->ter->getClosestTriangle(clickedPoint);
    baseTriangle->calculateNormalVector();

    float angle;

    for(runnel::Triangle* tri : this->ter->struct_triangle){
        tri->calculateNormalVector();
        angle = glm::acos(glm::dot(baseTriangle->normal, tri->
           normal) /
                           glm::length(baseTriangle->normal)*glm::
                              length(tri->normal));
        if (angle <= angleThreshold) {
            for(runnel::Point* p : tri->points) {
                terraceVertices.push_back(p->coord);
            }
        }
    }

    shader->fillPositionBuffer(terraceVertices);
}

void NormalVectorSimilarityAlgorithm::render(glm::mat4 matrix,
   float exag_z, glm::vec3 color){
    angleThreshold = glm::radians(conf.getAngleThreshold());
    if (shader){
        shader->render(matrix, exag_z, color);
    }
}

void NormalVectorSimilarityAlgorithm::glewReady(){
    shader = new ShaderNormalVectorSimilarity();
}

QString NormalVectorSimilarityAlgorithm::getName(){
    return QString("Normal␣Vector␣Similarity");
}

QWidget* NormalVectorSimilarityAlgorithm::getConf(){
    return &conf;
}

void NormalVectorSimilarityAlgorithm::changeAttr(){
    this->run(clickedPoint,ter);
    emit reload();
}
```

9 Appendix B: Full results

This appendix contains uncropped side-to-side views of results provided by Runnel and RiverTools.

9.1 Drainage network extraction on raw terrain

Figure 55 shows the results given by the corrected Peucker algorithm. In Figure 56 we present the results of drainage network extraction as performed by RiverTools and by the newly implemented algorithm RWFlood.

9.2 Preprocessing techniques

Figures 57 to 65 compare the results of drainage network extraction produced in Runnel by the Peucker, O'Callaghan and RWFlood algorithms, when applied on raw (left) and preprocessed (right) terrain. Particularly, Figures 57, 58 and 59 involve terrain preprocessed with the Garbrecht-Martz imposed gradients method for flat zone resolution; Figures 60, 61 and 62 show terrain preprocessed with the Garbrecht-Martz depression breaching algorithm, and Figures 63, 64 and 65 consider terrain preprocessed with both algorithms.

9.3 Drainage network extraction on preprocessed terrain

In Figures 66, 67 and 68 we compare the results of drainage network extraction as performed by RiverTools and by the Peucker, O'Callaghan and RWFlood algorithms, respectively, after applying preprocessing for both flat zone resolution and depression breaching & filling. Figure 69 contains a reference map of central Chile, produced by the Military Geographic Institute of Chile [5], which depicts the main rivers located within said zone.

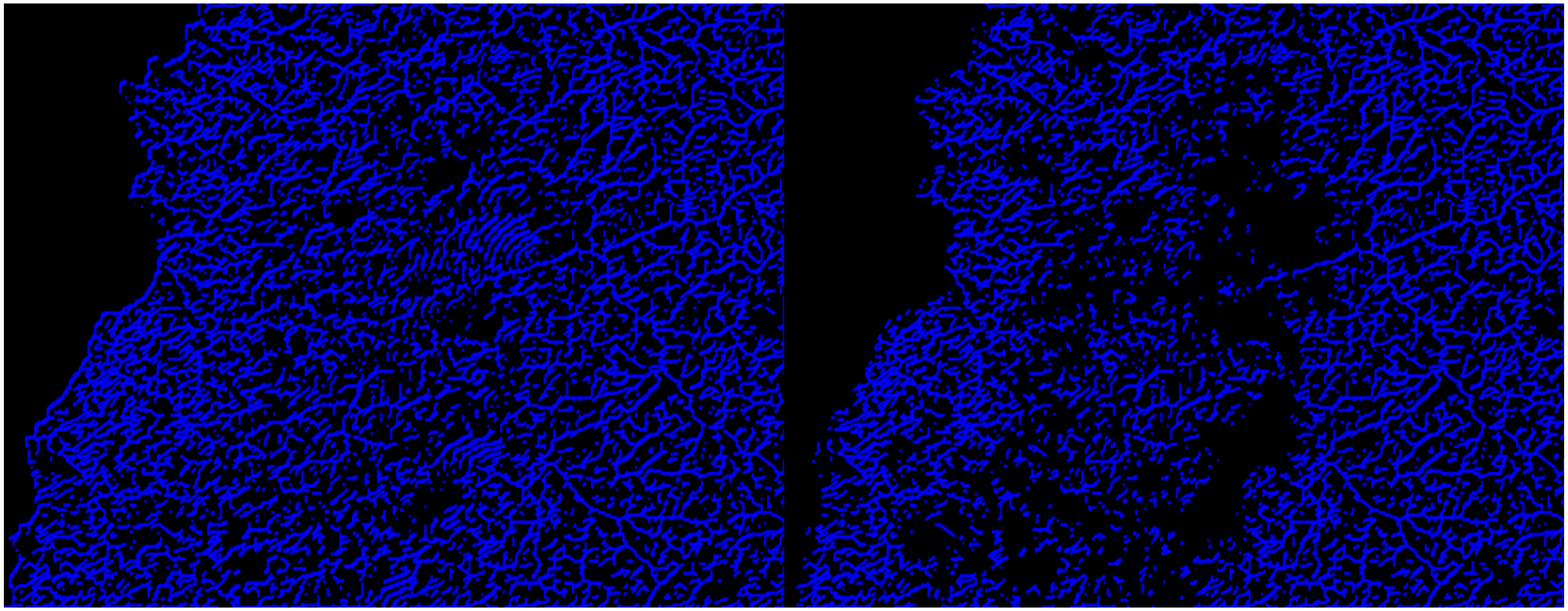


Figure 55: Comparison of drainage network extraction results produced in Runnel by the Peucker algorithm before (left) and after (right) fixing the software defect described in Chapters 3 and 4.

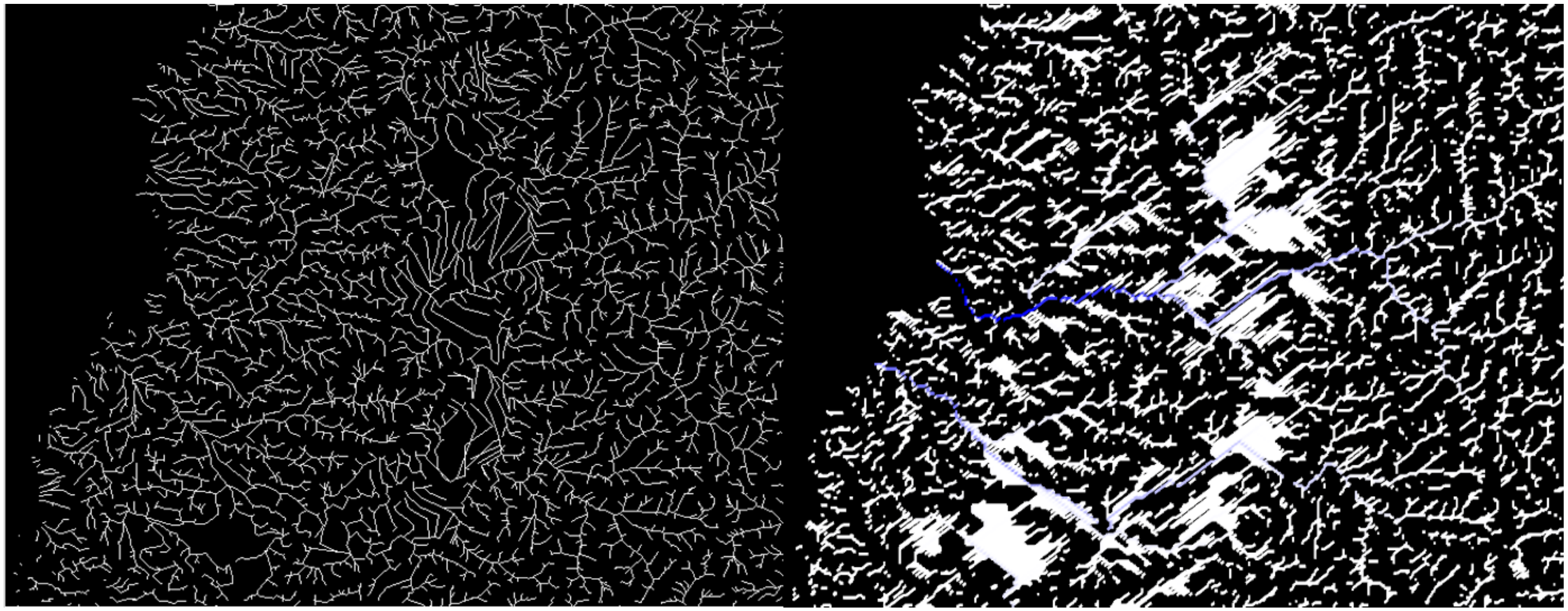


Figure 56: Comparison of drainage network extraction results produced by RiverTools (left) and the newly implemented RWFlood algorithm (right). Blue hue in the RWFlood algorithm results represents a higher water accumulation value.

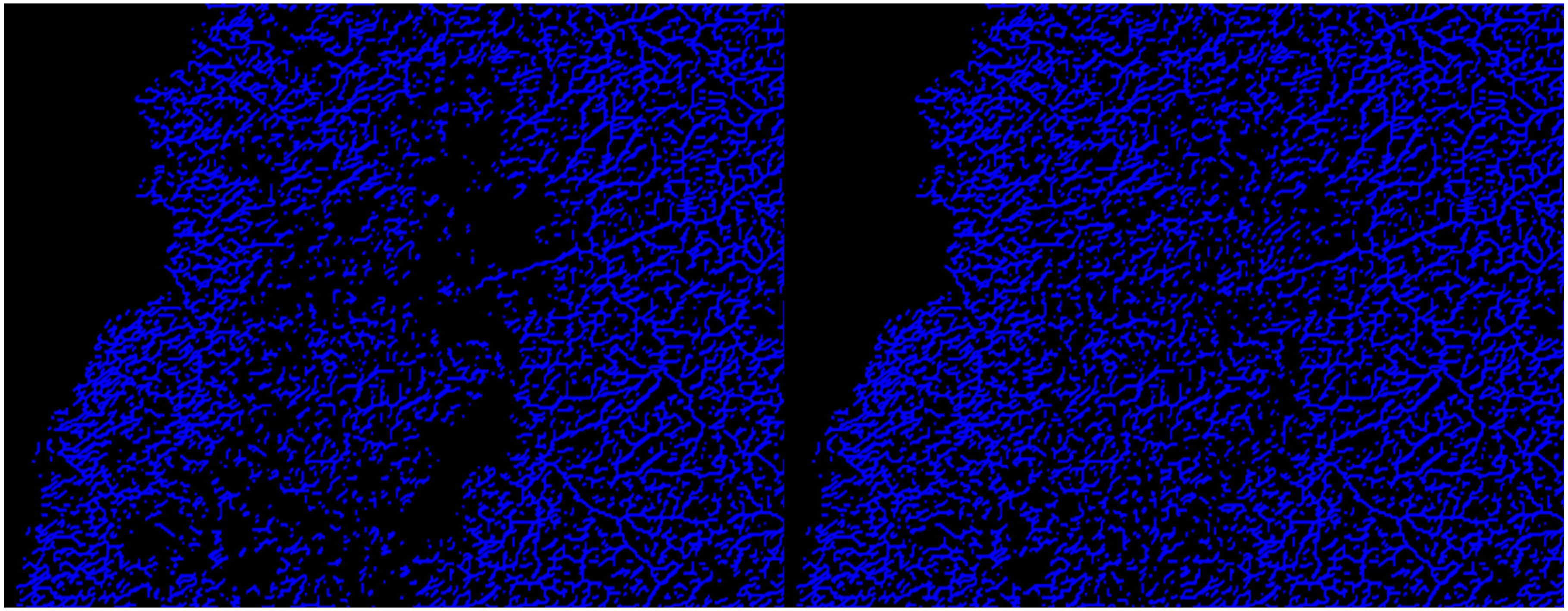


Figure 57: Comparison of drainage network extraction results produced in Runnel by the fixed Peucker algorithm on raw terrain (left) and terrain preprocessed to remove flat zones (right).

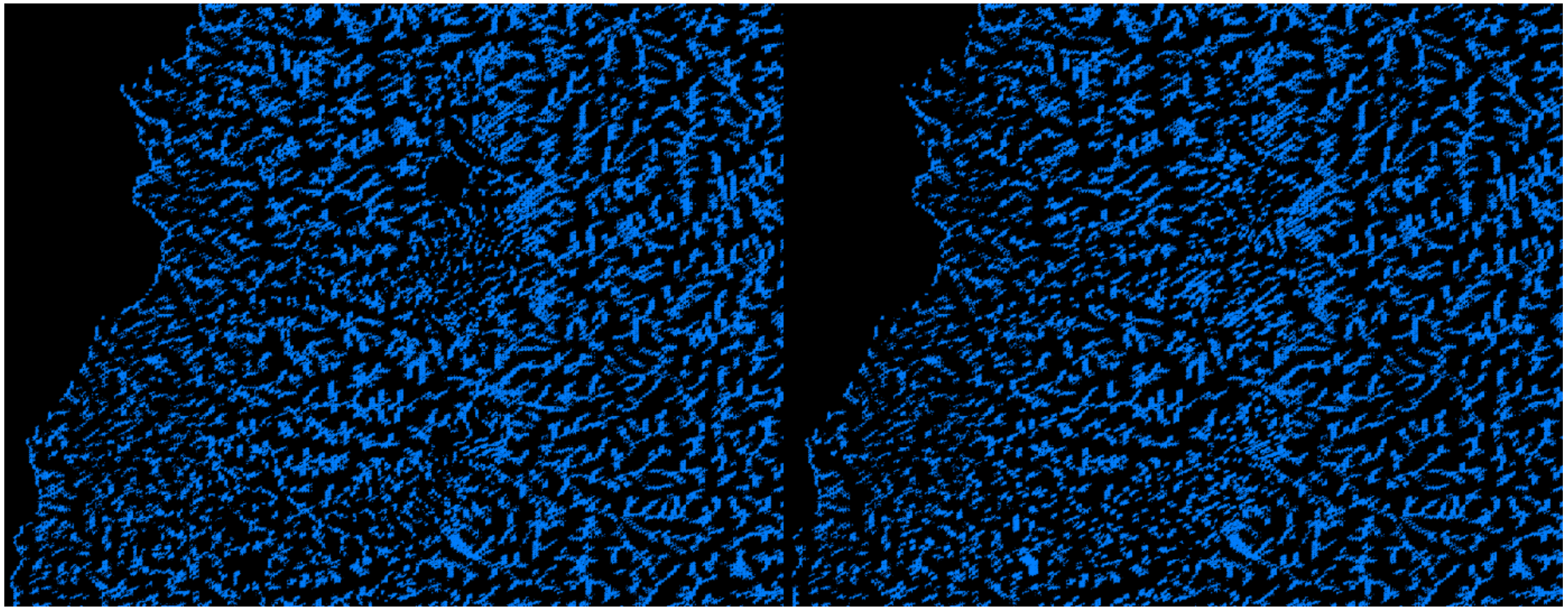


Figure 58: Comparison of drainage network extraction results produced in Runnel by the Callaghan algorithm on raw terrain (left) and terrain preprocessed to remove flat zones (right).

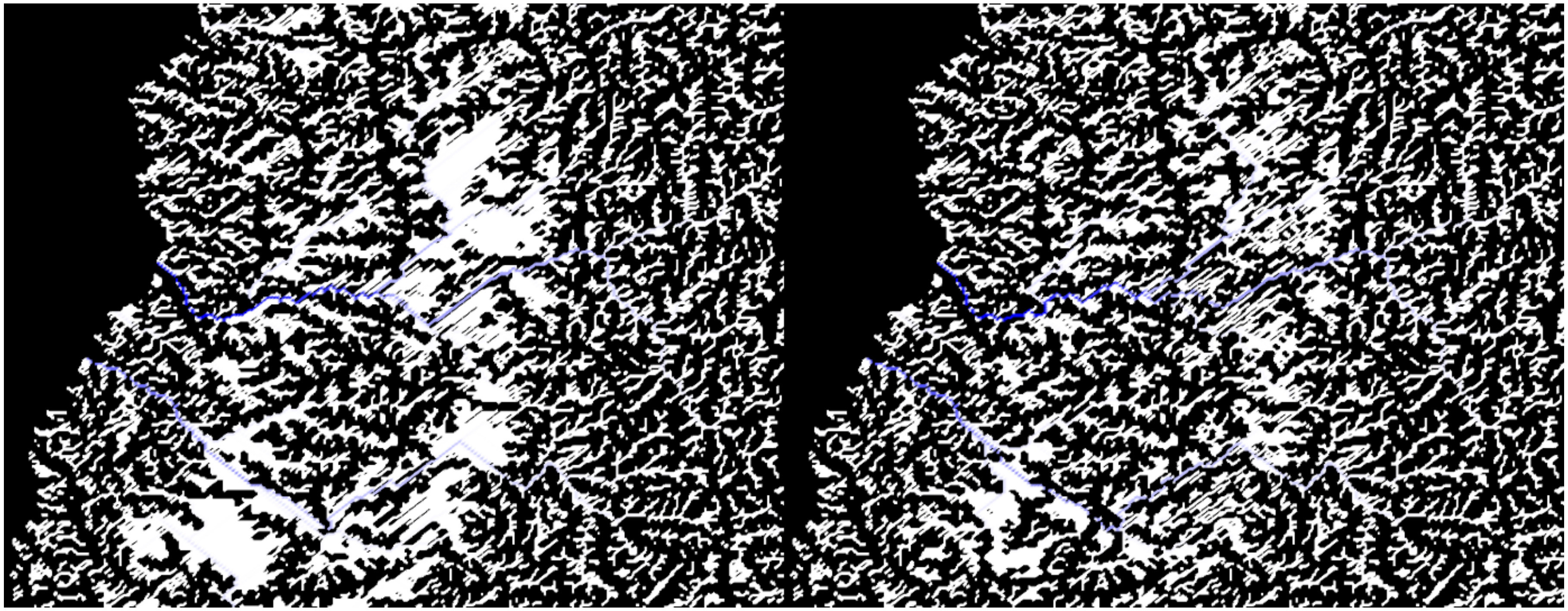


Figure 59: Comparison of drainage network extraction results produced in Runnel by the newly implemented RWFlood algorithm on raw terrain (left) and terrain preprocessed to remove flat zones (right).

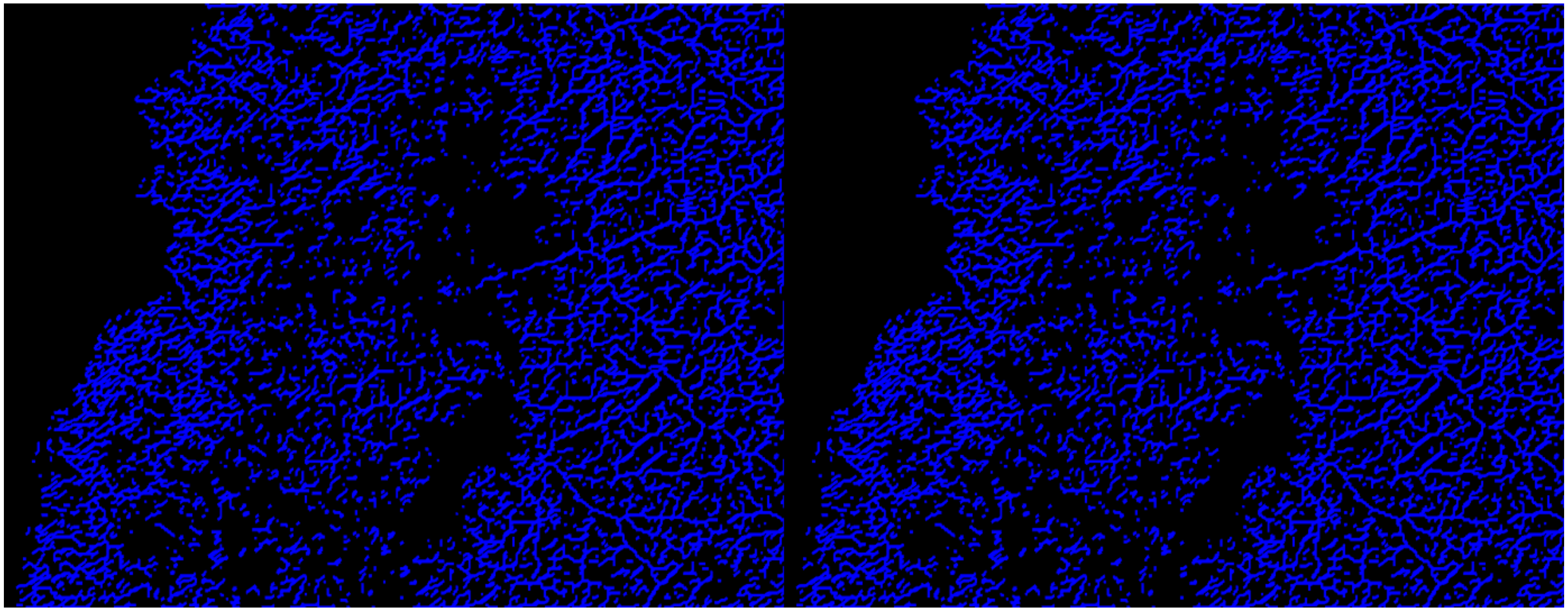


Figure 60: Comparison of drainage network extraction results produced in Runnel by the Peucker algorithm on raw terrain (left) and terrain preprocessed to breach and fill depressions (right).

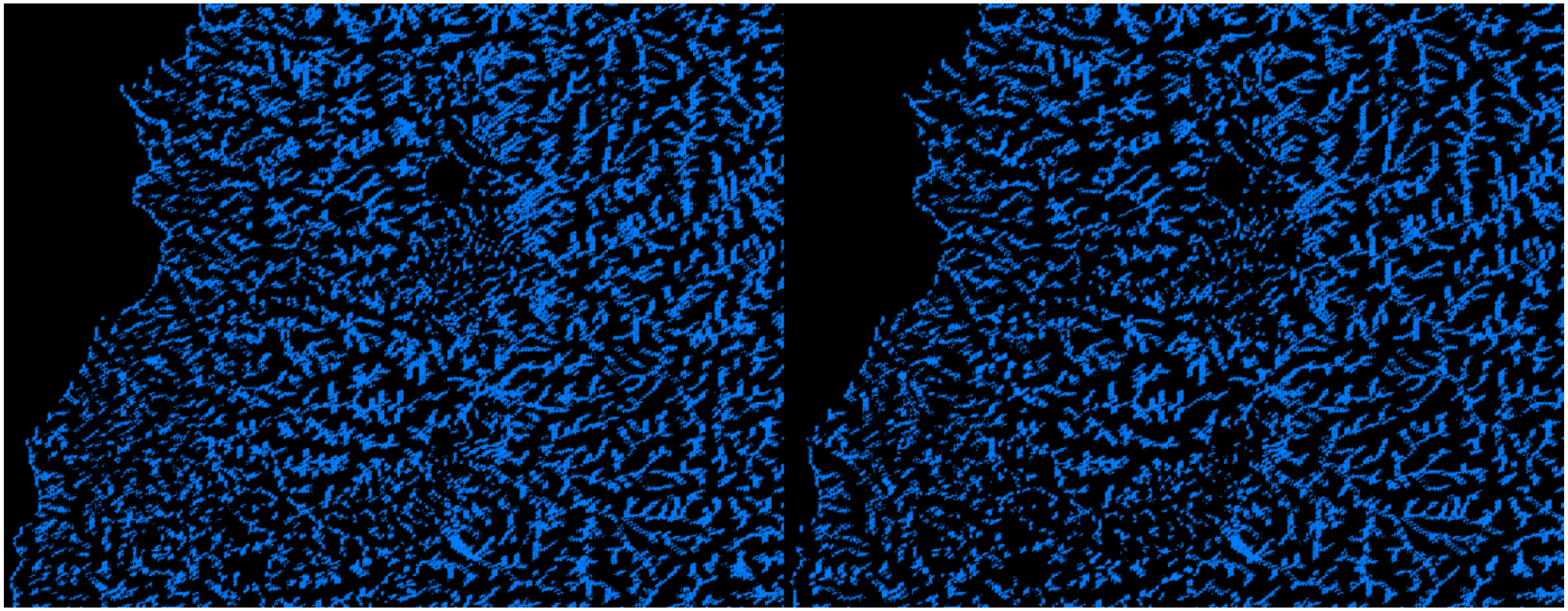


Figure 61: Comparison of drainage network extraction results produced in Runnel by the O'Callaghan algorithm on raw terrain (left) and terrain preprocessed to breach and fill depressions (right).

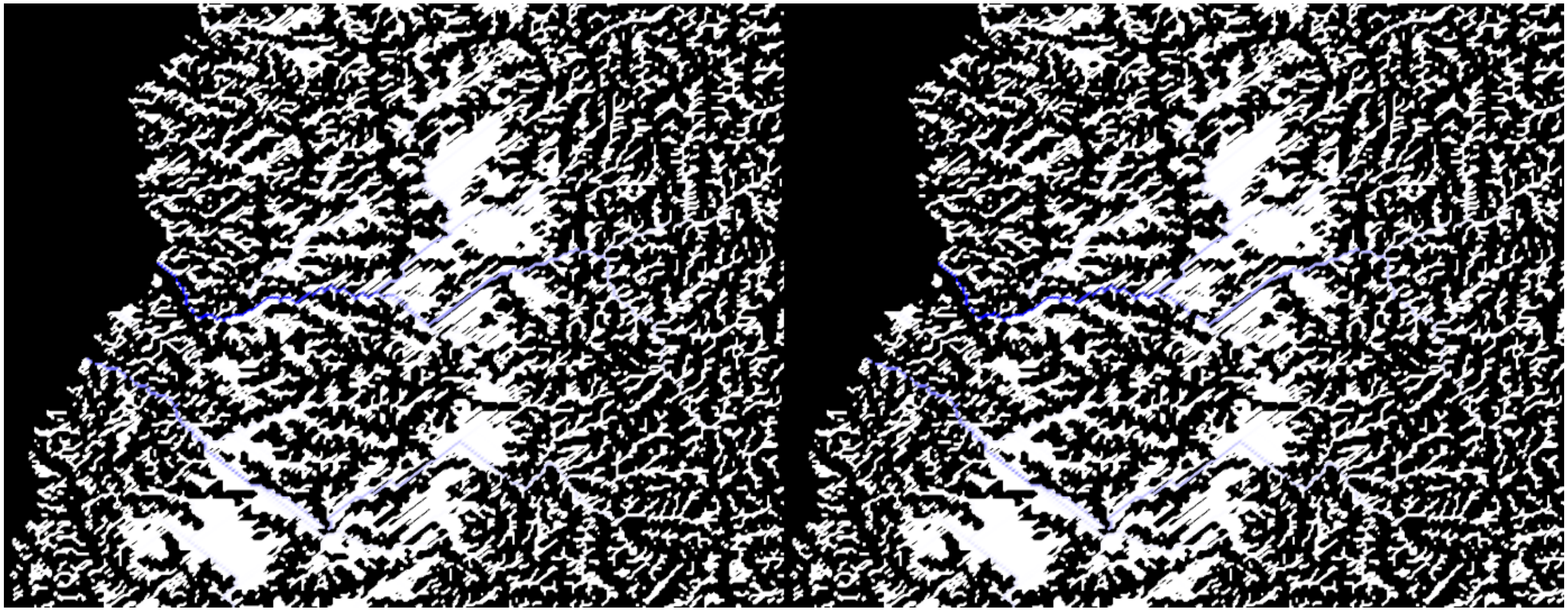


Figure 62: Comparison of drainage network extraction results produced in Runnel by the RWFlood algorithm on raw terrain (left) and terrain preprocessed to breach and fill depressions (right).

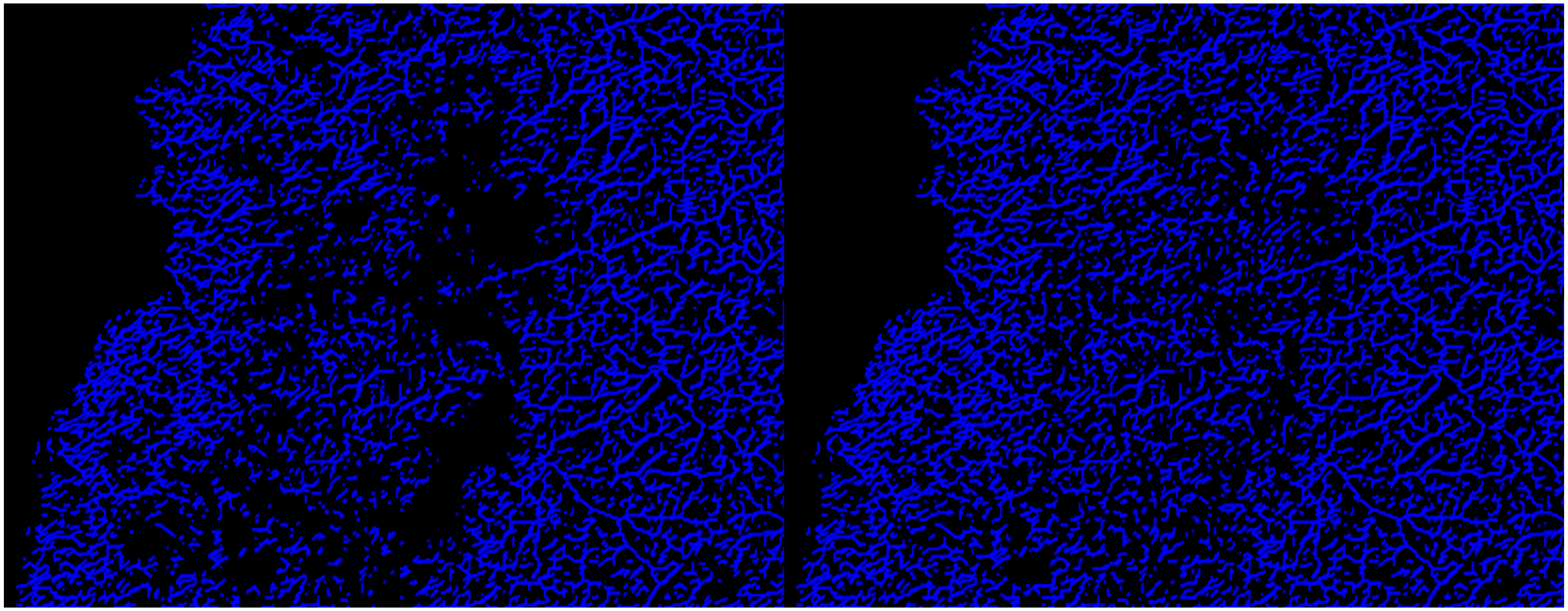


Figure 63: Comparison of drainage network extraction results produced in Runnel by the Peucker algorithm on raw terrain (left) and terrain preprocessed to resolve both flats and pits (right).

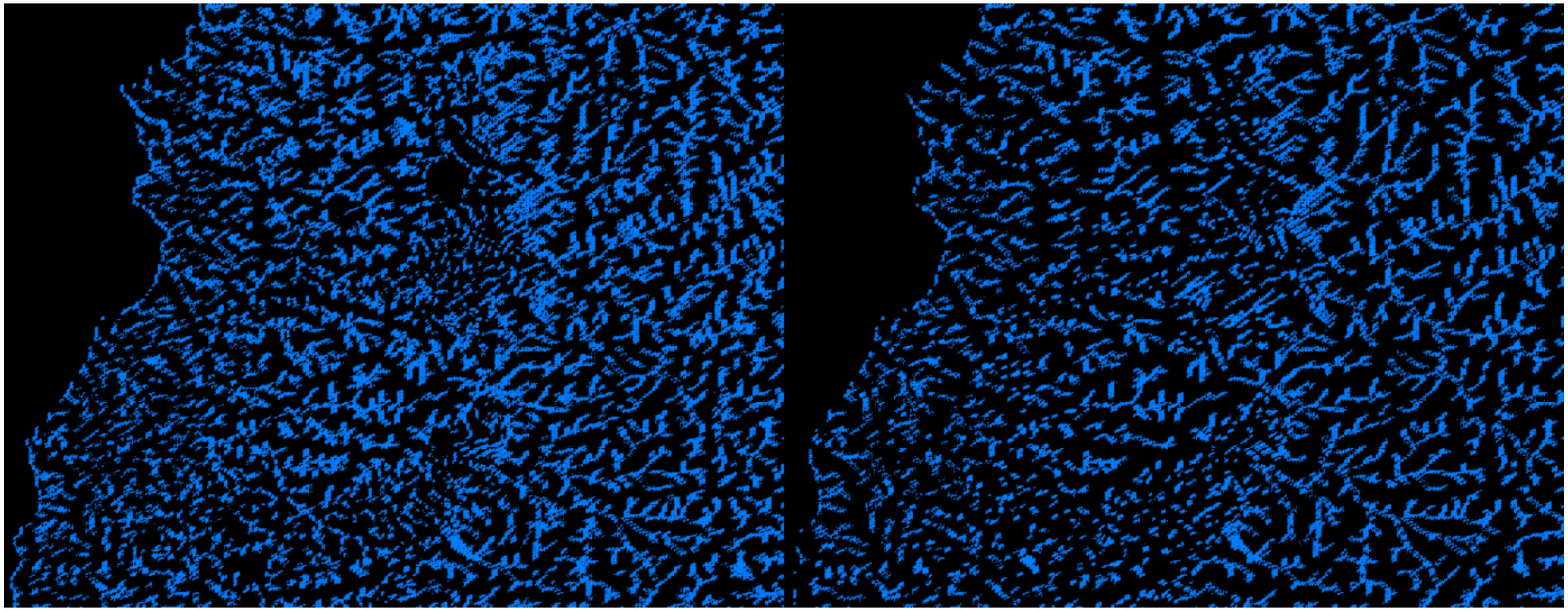


Figure 64: Comparison of drainage network extraction results produced in Runnel by the O'Callaghan algorithm on raw terrain (left) and terrain preprocessed to resolve both flats and pits (right).

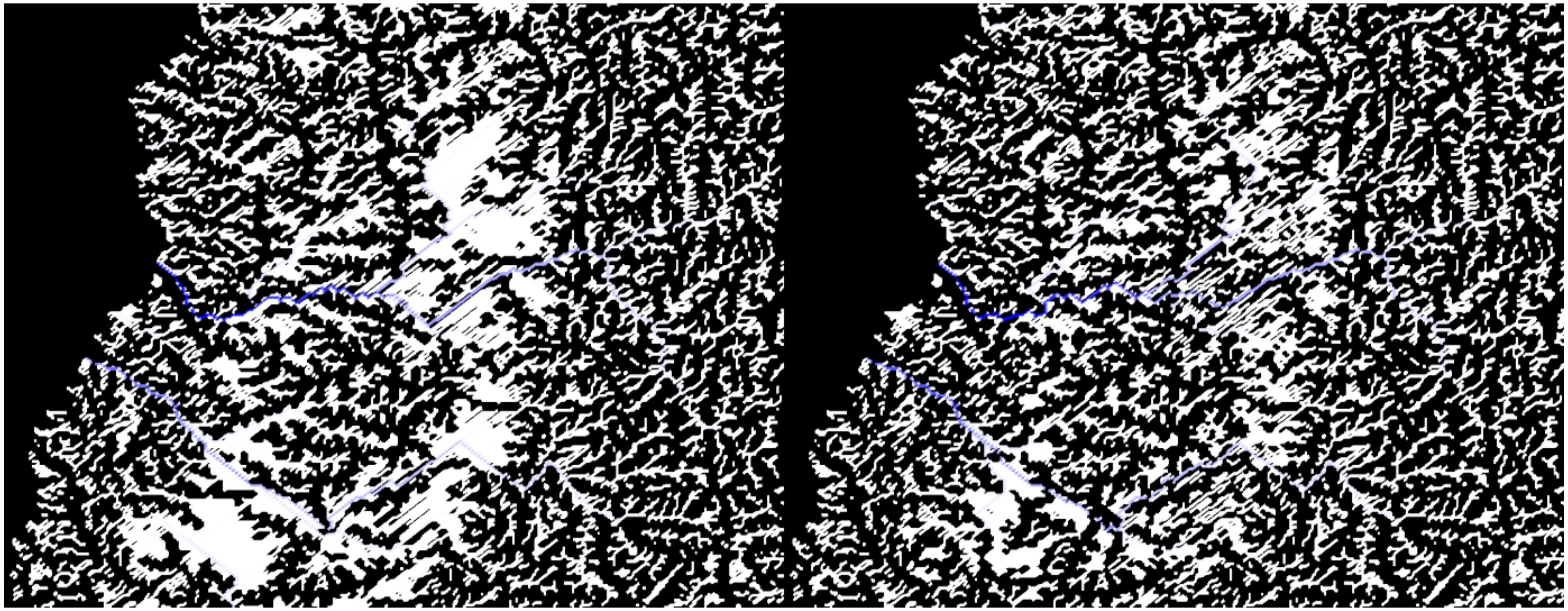


Figure 65: Comparison of drainage network extraction results produced in Runnel by the RWFlood algorithm on raw terrain (left) and terrain preprocessed to resolve both flats and pits (right).

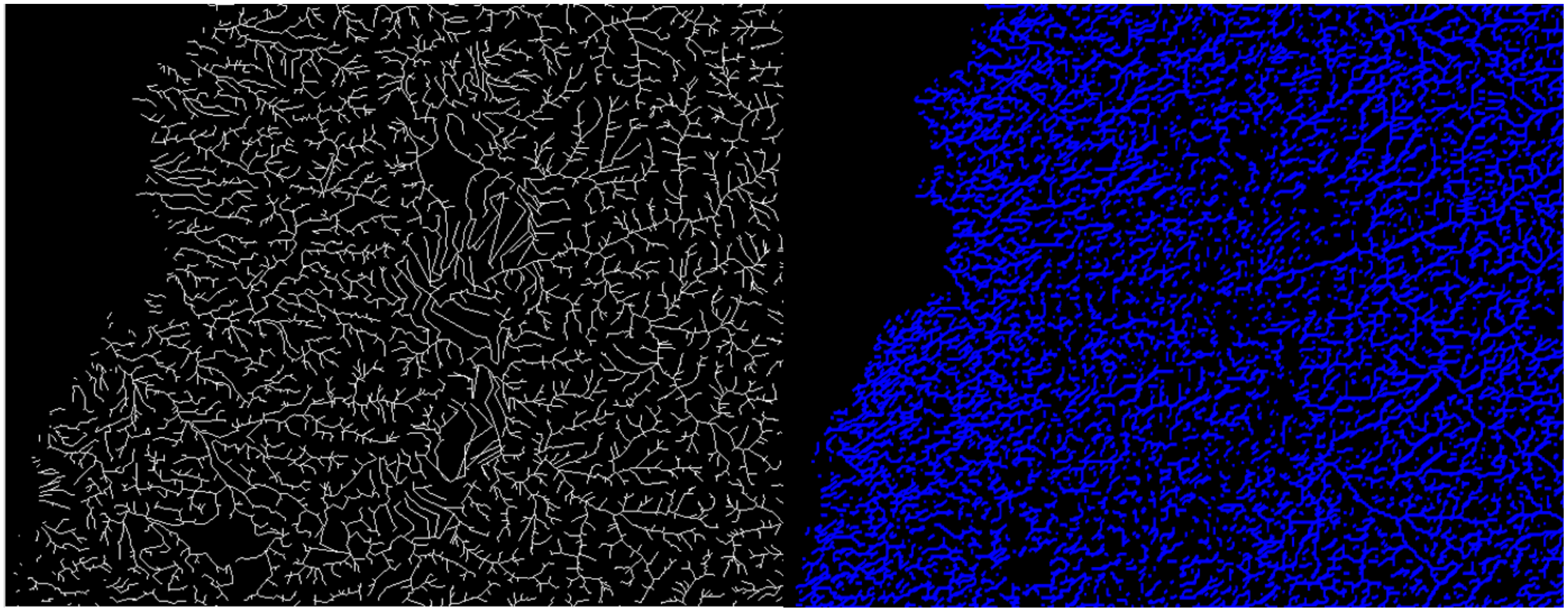


Figure 66: Comparison of drainage network extraction results produced by RiverTools (left) and the Peucker algorithm after applying the imposed gradients and depression breaching preprocessing algorithms in Runnel (right).

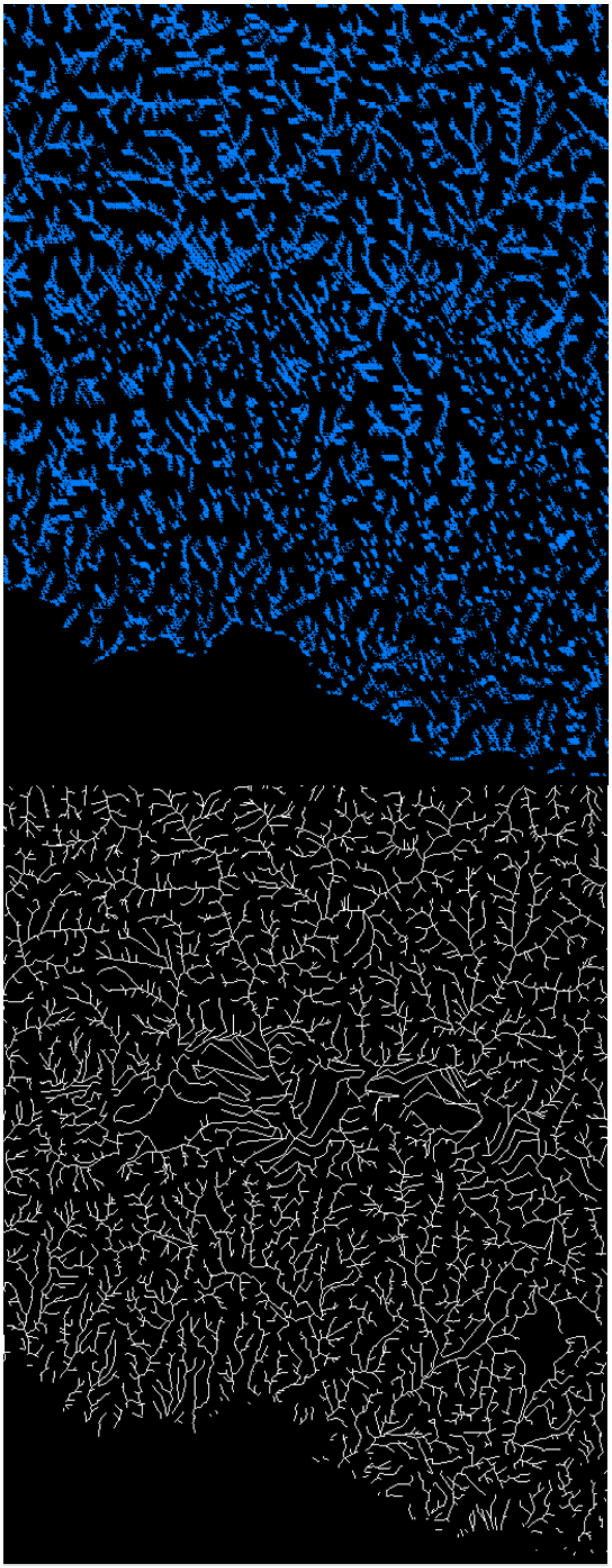


Figure 67: Comparison of drainage network extraction results produced by RiverTools (left) and the O'Callaghan algorithm after applying the imposed gradients and depression breaching preprocessing algorithms in Runnel (right).

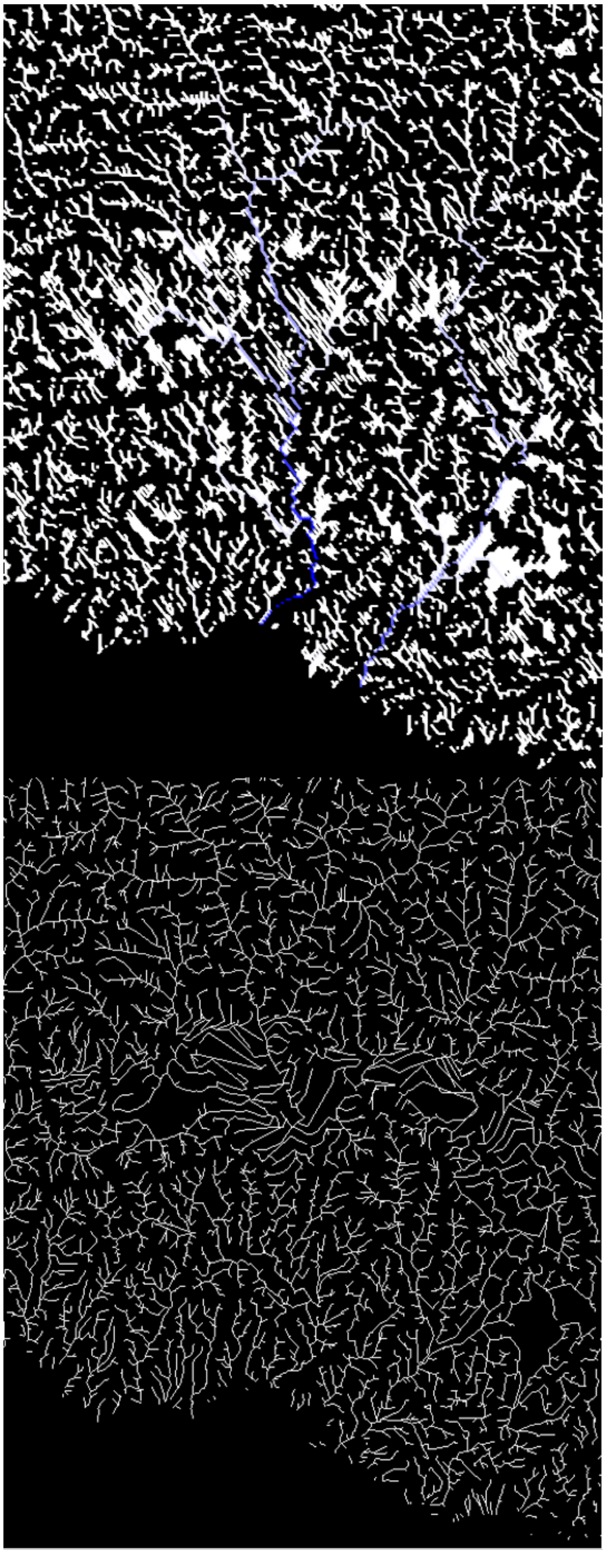


Figure 68: Comparison of drainage network extraction results produced by RiverTools (left) and the RWFlood algorithm after applying the imposed gradients and depression breaching preprocessing algorithms in Runnel (right).

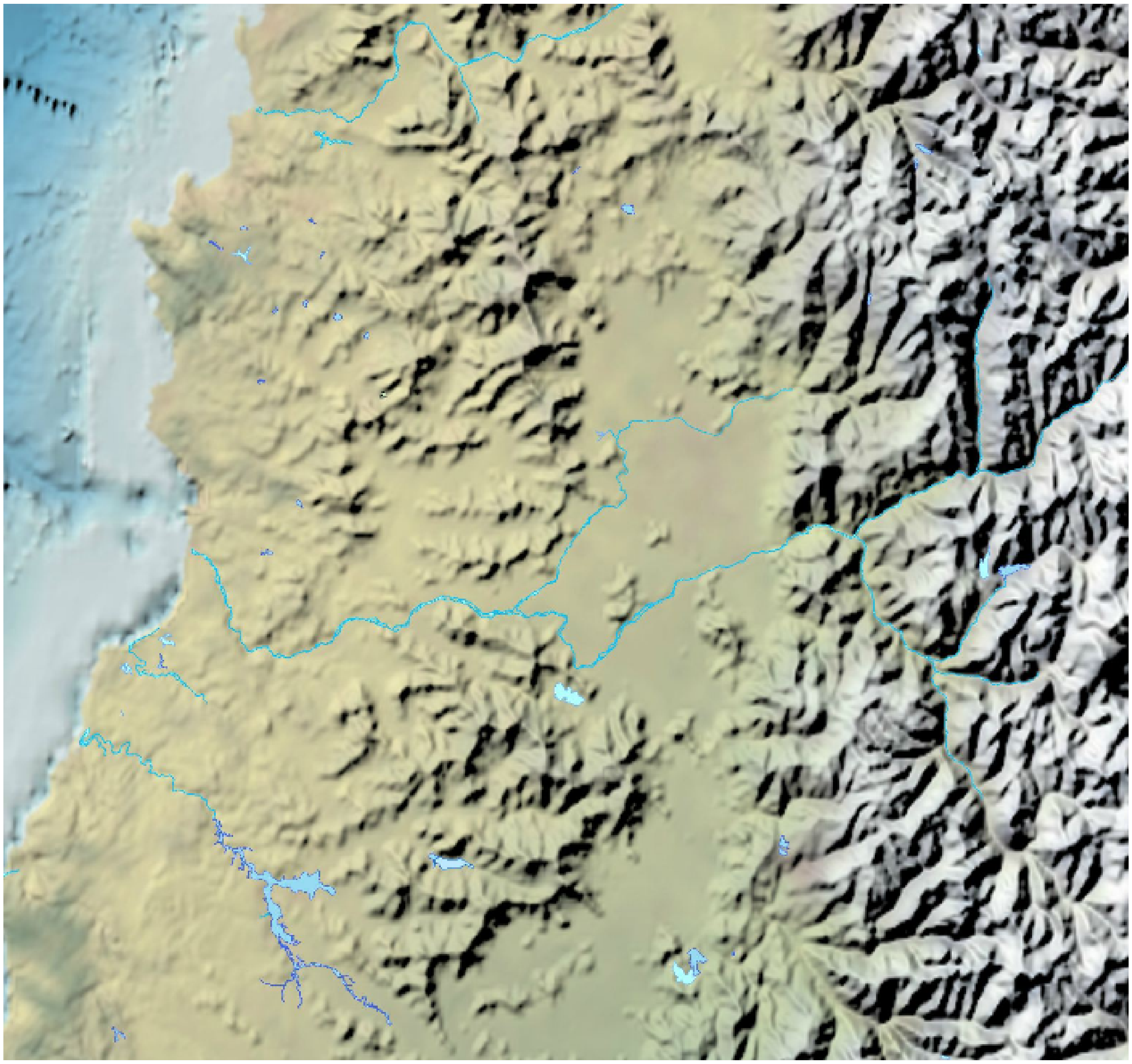


Figure 69: Map produced by the Military Geographic Institute of Chile, depicting main rivers located in central Chile within the boundaries of the corresponding DEM.

9.4 Strahler order calculation

In Figure 70 we present the results of Strahler order calculation on fully preprocessed terrain with the base mechanism provided by Runnel (left) and with our implementation of the Gleyzer algorithm (right), based on a Peucker drainage network. Figure 71 shows the results of the Gleyzer algorithm applied to a drainage network extracted by the newly implemented RWFlood algorithm.

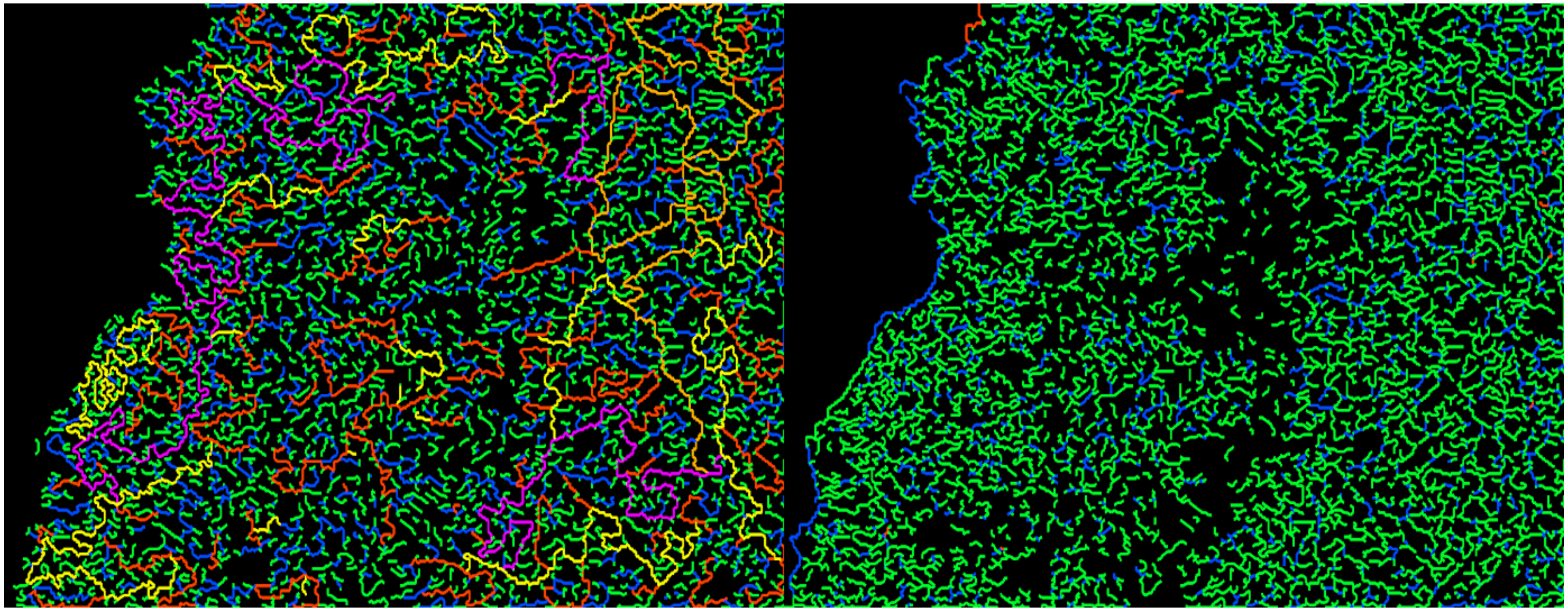


Figure 70: Comparison of Strahler order calculation results with the base mechanism provided by Runnel (left) and with our implementation of the Gleyzer algorithm (right) on fully preprocessed terrain with a Peucker network. Strahler orders are color-coded: green for order 1, blue for order 2, red for order 3, yellow for order 4, orange for order 5 and purple for order 6.

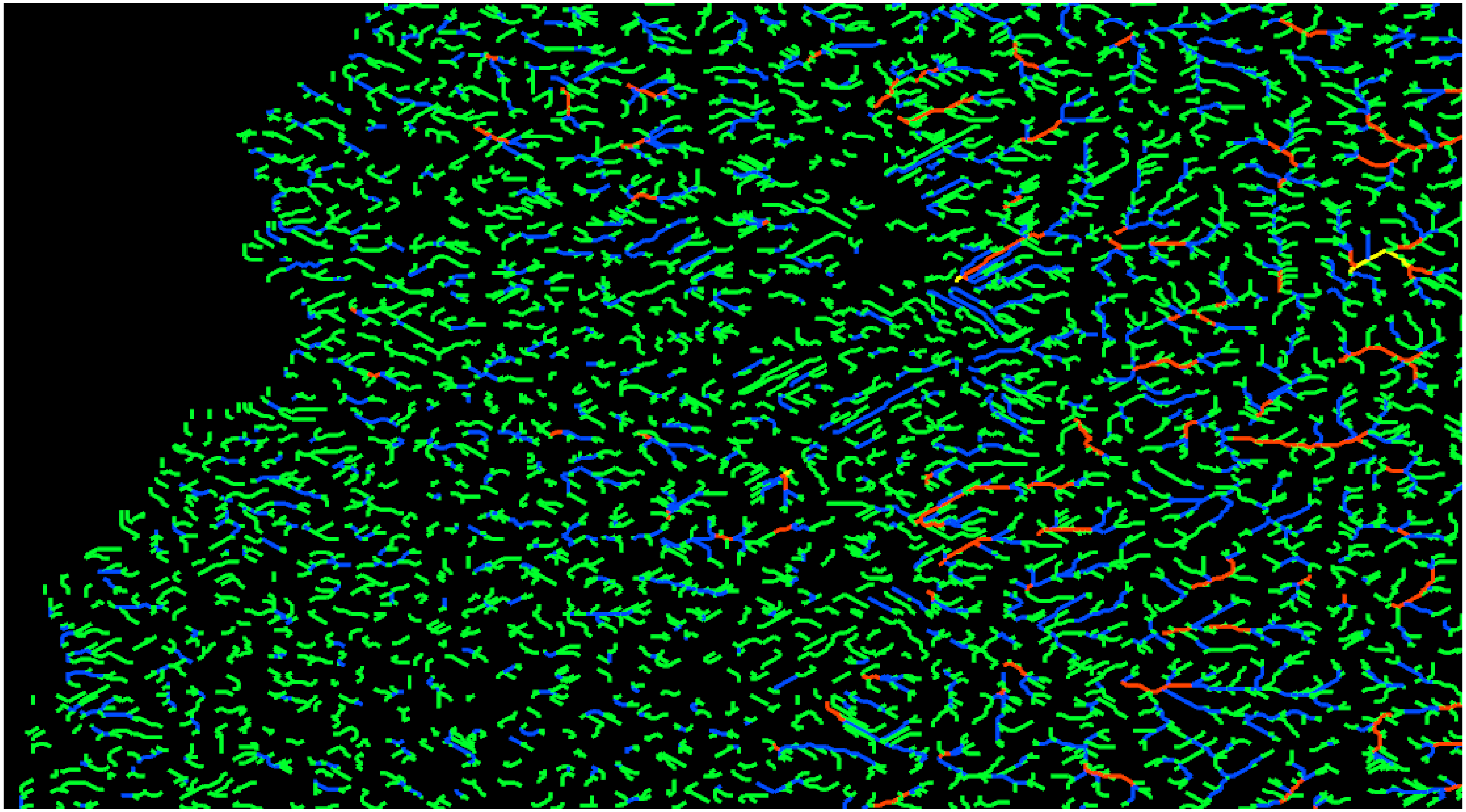


Figure 71: Strahler order calculation results with our implementation of the Gleyzer algorithm on fully preprocessed terrain with an RWFlood network. Strahler orders are color-coded: green for order 1, blue for order 2, red for order 3, yellow for order 4.

9.5 Terrace visualization

Figure 72 shows the results of the normal vector similarity algorithm applied on raw terrain. We set an angle threshold of 1 degree, and as our reference triangle, we select a perfectly flat triangle located in the ocean (leftmost region of DEM), allowing us to visualize all flat regions within the DEM. Figure 73 shows the results under the same conditions, applied to fully preprocessed terrain.

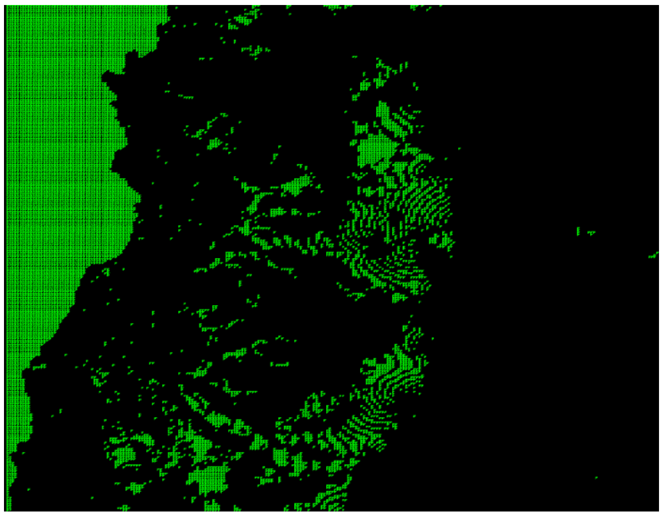


Figure 72: Results of the normal vector similarity algorithm for terrace visualization applied with an angle threshold of 1 degree on a perfectly flat reference triangle. The image shows all approximately flat zones within the DEM highlighted in green.

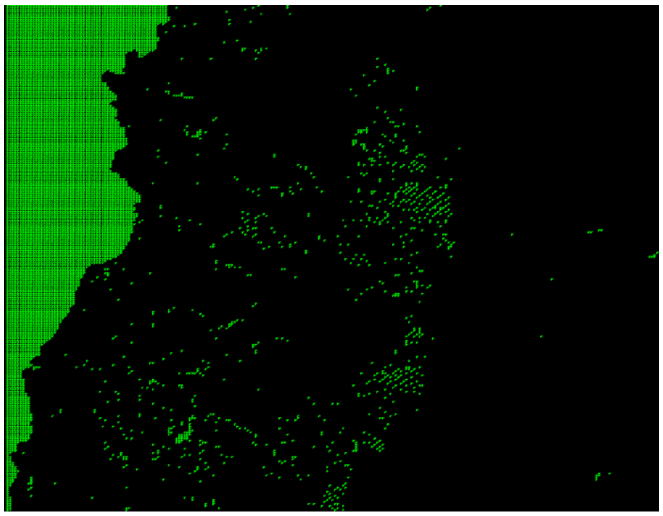


Figure 73: Results of the normal vector similarity algorithm for terrace visualization applied with an angle threshold of 1 degree on a perfectly flat reference triangle in the ocean, with terrain fully preprocessed to remove closed depressions and flats (on land).

9.6 High resolution DEM

Figures 74, 75 and 76 compare the tree network extracted in RiverTools with the results provided by the Peucker, O'Callaghan and RWFlood algorithms respectively in Runnel on the 4030x3080 Petorca province DEM. Figure 77 contains a reference map produced by the Military Geographic Institute of Chile depicting major streams located in the same area.

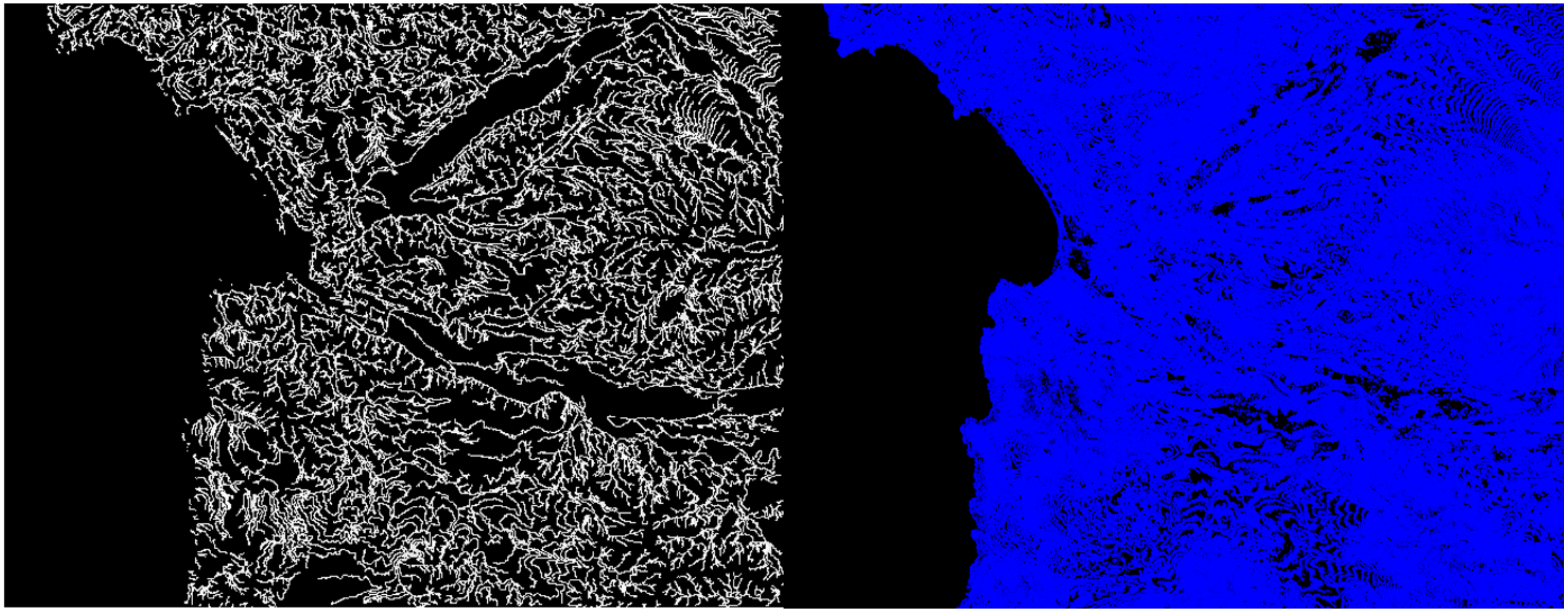


Figure 74: Comparison of drainage networks extracted by RiverTools (left), and by the Peucker algorithm in Runnel (right), on the 4030x3080 DEM covering a zone within the Petorca province.

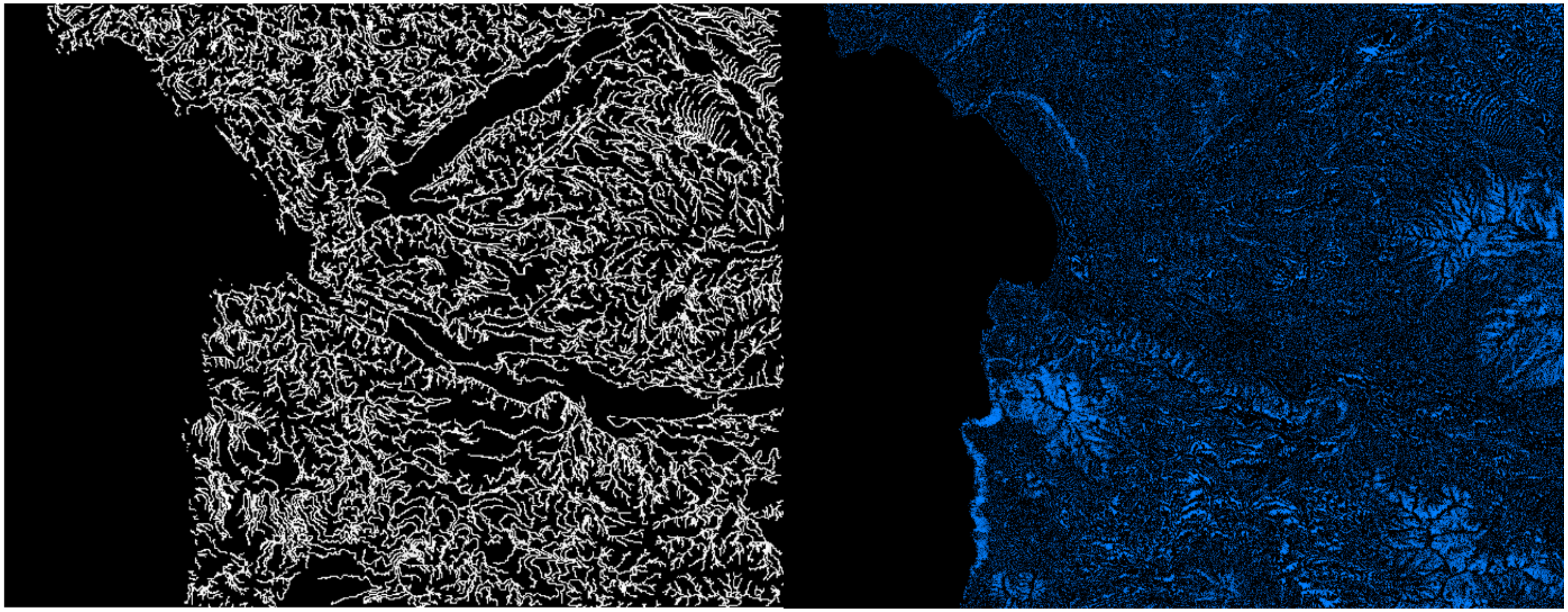


Figure 75: Comparison of drainage networks extracted by RiverTools (left), and by the O'Callaghan algorithm in Runnel (right), on the 4030x3080 DEM covering a zone within the Petorca province.

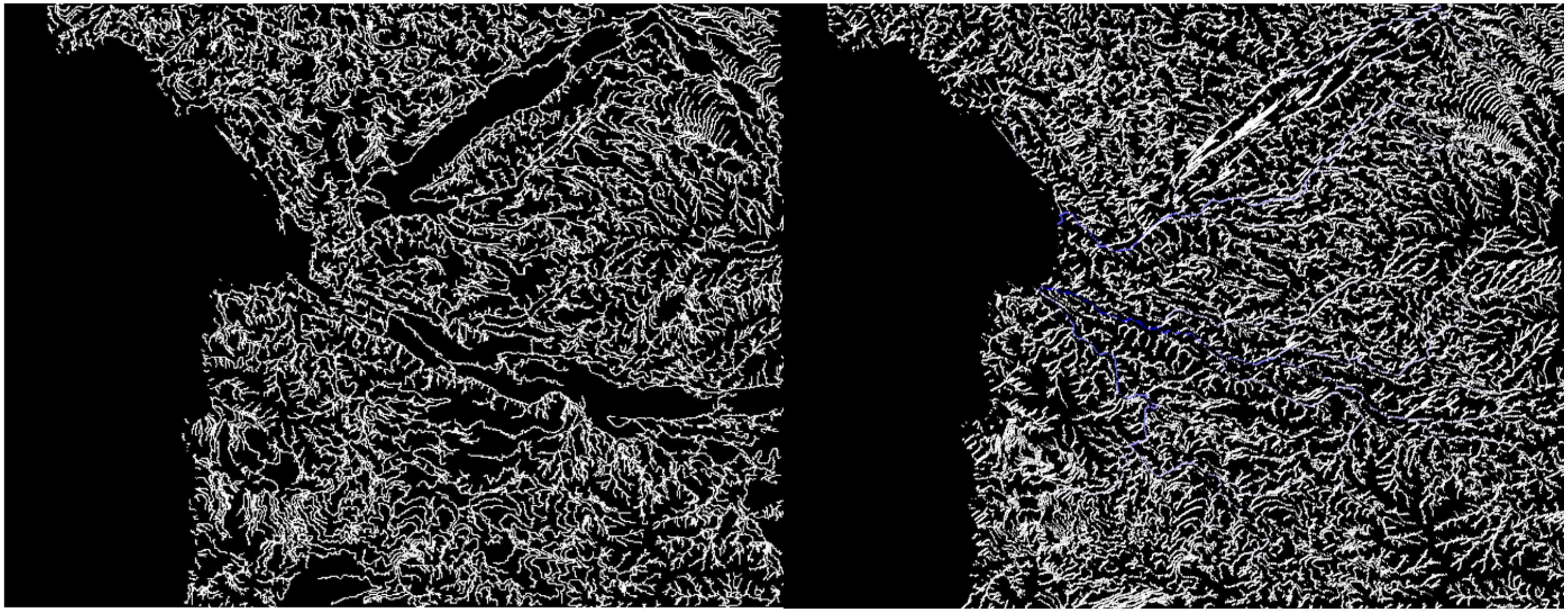


Figure 76: Comparison of drainage networks extracted by RiverTools (left), and by the RWFlood algorithm in Runnel (right), on the 4030x3080 DEM covering a zone within the Petorca province.

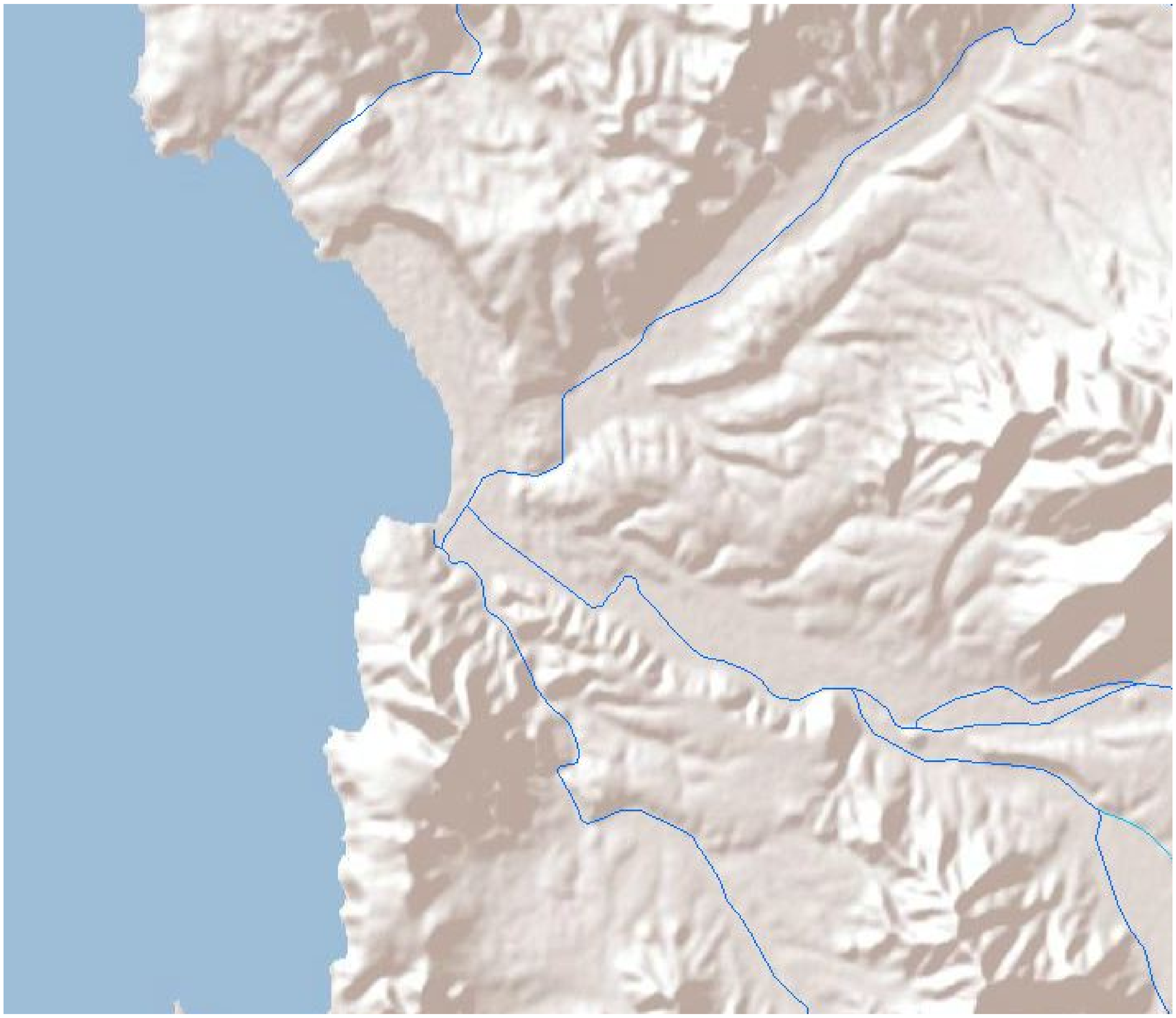


Figure 77: Map produced by the Military Geographic Institute of Chile, depicting major streams located in the area of Petorca province covered by the 4030x3080 DEM.

10 Bibliography

[1] Aquaveo — Water Modeling Solutions. http://www.aquaveo.com/software/wms-watershed-modeling-system-introduction. Retrieval date: 2015-11-16.

[2] ArcGIS — Main. http://www.arcgis.com. Retrieval date: 2015-11-16.

[3] ASTER: Advanced Spaceborne Thermal Emission and Reflection Radiometer. https://asterweb.jpl.nasa.gov/. Retrieval date: 2015-11-02.

[4] GRASS GIS - Home. http://grass.osgeo.org. Retrieval date: 2015-11-16.

[5] Instituto Geográfico Militar IGM. Maps obtained from WEBMAP IGM. http://www.igm.cl/. Retrieval date: 2016-04-07.

[6] Landsat. http://landsat.usgs.gov. Retrieval date: 2015-11-02.

[7] Rivix.com: Main Page. http://rivix.com. Retrieval date: 2015-11-16.

[8] Shuttle Radar Topography Mission. http://srtm.usgs.gov/. Retrieval date: 2015-11-02.

[9] Lars Arge, Laura Toma, and Jeffrey Scott Vitter. I/O-efficient algorithms for problems on grid-based terrains. *Journal of Experimental Algorithmics (JEA)*, 6:1, 2001.

[10] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 18–30. Springer, 2009.

[11] Sagi Dalyot. Digital terrain models, 2007. Technion — Israel Institute of Technology. http://tx.technion.ac.il/ dalyot/docs/Intro-DTM.pdf. Retrieval date: 2015-10-28.

[12] Guillermo De Bernardo, Sandra Álvarez-García, Nieves R Brisaboa, Gonzalo Navarro, and Oscar Pedreira. Compact querieable representations of raster data. In *International Symposium on String Processing and Information Retrieval*, pages 96–108. Springer, 2013.

[13] Jurgen Garbrecht and Lawrence W Martz. The assignment of drainage direction over flat surfaces in raster digital elevation models. *Journal of hydrology*, 193(1-4):204–213, 1997.

[14] Alexander Gleyzer, Michael Denisyuk, Alon Rimmer, and Yigal Salingar. A fast recursive gis algorithm for computing strahler stream order in braided and nonbraided networks. *Journal of the American Water Resources Association*, 40:937–946, 2004.

[15] Thiago L Gomes, Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Guilherme C Pena. Efficiently computing the drainage network on massive terrains using external memory flooding process. *GeoInformatica*, pages 1–22, 2015.

[16] Felipe Andrés González Maldonado. *Desarrollo metodológico para la construcción de un DEM de alta resolución y aplicación en el análisis morfoestructural de la Cordillera de la Costa entre los 32,25°S y 32,63°S.* 2015. Undergraduate Dissertation. Department of Geology, University of Chile.

[17] Richard John Huggett. *Fundamentals of geomorphology.* Routledge, 2007.

[18] Guy Joseph Jacobson. Succinct static data structures. 1988.

[19] Susan K Jenson and Julia O Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric engineering and remote sensing*, 54(11):1593–1600, 1988.

[20] Susana Ladra, José R Paramá, and Fernado Silva-Coira. Compact and queryable representation of raster datasets. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, page 15. ACM, 2016.

[21] Zhilin Li, Christopher Zhu, and Chris Gold. *Digital terrain modeling: principles and methodology*. CRC press, 2004.

[22] John B Lindsay and Irena F Creed. Removal of artifact depressions from digital elevation models: towards a minimum impact approach. *Hydrological processes*, 19(16):3113–3126, 2005.

[23] Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Guilherme C Pena. A new method for computing the drainage network based on raising the level of an ocean surrounding the terrain. In *Bridging the Geographic Information Sciences*, pages 391–407. Springer, 2012.

[24] Lawrence W Martz and Jurgen Garbrecht. The treatment of flat areas and depressions in automated drainage analysis of raster digital elevation models. *Hydrological processes*, 12(6):843–855, 1998.

[25] Lawrence W Martz and Jurgen Garbrecht. An outlet breaching algorithm for the treatment of closed depressions in a raster dem. *Computers & Geosciences*, 25(7):835–844, 1999.

[26] M Metz, H Mitasova, and RS Harmon. Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. *Hydrology and Earth System Sciences*, 15(2):667–678, 2011.

[27] Ian Donald Moore, RB Grayson, and AR Ladson. Digital terrain modelling: a review of hydrological, geomorphological, and biological applications. *Hydrological processes*, 5(1):3–30, 1991.

[28] John F O'Callaghan and David M Mark. The extraction of drainage networks from digital elevation data. *Computer vision, graphics, and image processing*, 28(3):323–344, 1984.

[29] Thomas K Peucker and David H Douglas. Detection of surface-specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and image processing*, 4(4):375–387, 1975.

[30] Ignacia Pérez. *Visualización de mallas de terreno e identificación de patrones de drenaje en cuencas*. 2014. Undergraduate Dissertation. Department of Computer Science, University of Chile.

[31] María Pía Rodríguez. *Evolución de la erosión y del relieve del antearco de Chile Central (33-34o.S) durante el neógeno mediante el análisis de minerales pesados detríticos y la geomorfología*. 2008. Master's Thesis. Department of Geology, University of Chile.

[32] Khalid Sayood. *Introduction to data compression (4th ed.)*. Newnes, 2012.

[33] Arthur N Strahler. Quantitative analysis of watershed geomorphology. *Civ. Eng*, 101:1258–1262, 1957.

[34] Laura Toma, Rajiv Wickremesinghe, Lars Arge, Jeffery S Chase, Jeffery Scott Vitter, Patrick N Halpin, and Dean Urban. Flow computation on massive grids. In *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, pages 82–87. ACM, 2001.

[35] Andrea Tribe. Automated recognition of valley lines and drainage networks from grid digital elevation models: a review and a new method. *Journal of Hydrology*, 139(1):263–293, 1992.

[36] CR Twidale. River patterns and their meaning. *Earth-Science Reviews*, 67(3):159–218, 2004.

[37] Marc van Kreveld. Digital Elevation Models: overview and selected TIN algorithms. *Course Notes for the CISM Advanced School on Algorithmic foundations on Geographical Information Systems, Utrecht University*, 1996.

[38] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[39] John P Wilson and John C Gallant. Digital terrain analysis. *Terrain analysis: Principles and applications*, pages 1–27, 2000.

[40] Emilie R Zernitz. Drainage patterns and their significance. *The Journal of Geology*, pages 498–521, 1932.

[41] Ling Zhang and Eric Guilbert. A study of variables characterizing drainage patterns in river networks. *ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 1:29–34, 2012.