Synthesis Project in Geomatics

GEO1101.2020 – AHN3

Khaled Alhoz 4560310 Kristof Kenesei 5142334
Manos Papageorgiou 5004845 Lisa Keurentjes 4557670
Maarten de Jong 4290933

June 2020

The work in this project was carried out in the:

3D geoinformation group
Department of Urbanism
Faculty of the Built Environment & Architecture
Delft University of Technology

Supervisors:	Dr. Hugo Ledoux
	Dr. Ravi Peters
Client:	ir. Jeroen Leusink (Het Waterschapshuis)

Contents

List of Figures

List of Tables

List of Algorithms

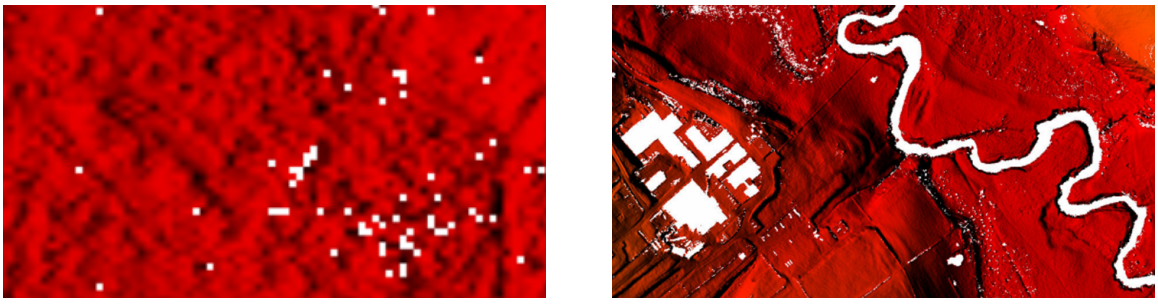
1 Introduction

The AHN is a point cloud with detailed and precise height data for the whole of the Netherlands. The height has been collected with helicopters and aircraft using laser technology, which results in the height of every square meter in the Netherlands being known to an accuracy of 5 centimeters. Automatic classification is also available on ground, buildings, vegetation, water bodies, and other street furniture. Besides the point cloud, you may also download the AHN as a raster file such as the ground level DTM, which is an interpolation of the ground filtering, and the DSM, which is a interpolation of all the points in the point cloud.

The AHN is a collaboration between the Water Boards, Provinces, and Rijkswaterstaat. These organizations have jointly taken the initiative to manufacture the AHN because they are jointly responsible for a safe Netherlands in the field of water system and flood defense management. The Waterschapshuis is the "owner" of the AHN data and therefore the client of this project. We work together with the Waterschapshuis to improve the rasters, so they can replace the current one with an improved version.

The AHN files have some problems; the aim of the project is to solve those and with that make the gridded DTM and DSM better. The first problem is that all the AHN tiles are collected and classified by different collectors, this results in that the points cloud and classification is not guaranteed to be the same quality. To decide which points we use for the gridded DTM and which points to use for the gridded DSM, we first need to check how "correct" the classification is. Furthermore, there are a few problems in the interpolation method itself. The first problem is in the method itself, at this moment only cells containing LiDAR points get a value, which results in missing pixels, as can be seen in Figure 1.1a. The missing pixels get a 'nodata' value in the raster, when you use the rasters then for making a model, you need to fill in those pixels by yourself, so we aim to solve the missing pixel problem.

Another issue with the interpolation method, is that it makes use of the points within a pixel and creates holes beneath buildings in the DTM. In the DTM and the DSM also the water bodies appear as holes, seeing water points are filtered out of the used points for interpolation. These two problems result in big holes in the raster, which is shown in Figure 1.1b.



(a) missing pixels

(b) empty water bodies and buildings

Figure 1.1: Problems of the AHN3

While fixing the problems with the AHN, we'll also get a scaling problem. The data set for the whole of the Netherlands is over approximately 14 TB of uncompressed point cloud data. That's a lot to download, therefore the AHN data is made available in so-called map sheets. The whole of the Netherlands is divided into rectangles, each of which has been given a unique number, which cover an area of 6.25 kilometers in the north-south direction and 5 kilometers in the east-west direction. As a result, the data set is divided into approximately 1100 small pieces, which are easier to download and use. A representation of these 1100 tiles can be seen in Figure 1.2.

Figure 1.2: The 1100 tiles

The scaling problem has two sides, first of all the compiling of all the big tiles, which are still around fourteen GB and how to process them efficiently. Secondly, the tiles need to be stitched back together, so that the edges of tiles are continuous.

1.1 Results

The outcome of the project has been successful from the interpolation perspective, where the final results have shown major improvements over the original rasters, and a multitude of possible other interpolation methods have shown potential. In the end, the algorithm used to create the gridded DTM was based on Startin's Laplace interpolation and for DSM a Python implementation of quadrant-based IDW was used. The Startin Laplace method gives statistically good results while running quickly on it's Rust base, though using quite some memory. The quadrant-based IDW has proven to be the best way to interpolate results which include the buildings, creating crisp edges without too many artefacts. Furthermore, the introduction of a polygon water flattening step was essential to prevent no-data values where water bodies prevented for accurate interpolation. The long-shot goal of being able to process all the tiles for the Netherlands was missed by a lot, eventually choosing to interpolate a series of tiles neighboring Delft to create a contiguous result set. Zuid-Holland has therefore only been partially completed, whereas the expectation was to be able to complete this area with ease in three days. An initial overview of the comparison for a single tile in Rotterdam (37HN1), can be seen in Figure 1.3 for the DTM results and Figure 1.4 for the DSM results.

The finalized code can be found in this GitHub repository under the MIT license, as all components used are free to use `https://github.com/tudelft3d/geo1101.2020.ahn3`. This repository contains the necessary documentation to get this project up and running on your local machine. This includes a variety of tools and datasets which have been used for processing steps outside of the main code. All the various interpolation methods that have been tested and implemented can be found for reference in another GitHub repository `https://github.com/khalhoz/geo1101-ahn3-GF-and-Interpolation`.

(a) DTM as currently available for download

(b) DTM as result of this project

Figure 1.3: Comparison of before and after results of DTM for tile 37HN1 (Rotterdam)

(a) DSM as currently available for download

(b) DSM as result of this project

Figure 1.4: Comparison of before and after results of DSM for tile 37HN1 (Rotterdam)

2 Pipeline

To tackle the challenge of creating an improved gridded DTM/DSM from AHN3 data, a processing 'pipeline' has been constructed. The next sections will describe each of the steps in this pipeline that are used to generate our improved DTM/DSM results. The range of steps necessary were agreed upon at the start of the project: ground filtering/pre-processing, interpolation, hole filling, and hydro-flattening. Within the pipeline these operations are performed successively in the order as we listed them here. Not strictly part of the pipeline itself, the scaling of the pipeline from single tile testing to multi-tile processing of AHN3 was an equally important task in this project owing to the extremely large volumes of data AHN3 is comprised of.

2.1 Ground Filtering / Pre-Processing Stage

To generate DTM rasters, we had to first remove non-ground points (points that are not part of the bare-earth surface of the earth) from the point cloud. AHN3 offers a well-classified point cloud from which we could easily separate out ground points based on the pre-existing classifications and use them as an input for our interpolation algorithm. A list depicting the first ten classifications of AHN3 can be seen in Table 2.1.

However, we decided to question the stock AHN3 classifications and conduct a series of tests to assess its quality, in order to make the final choice regarding whether to accept them or to use a ground filtering algorithm in our pipeline. This was done by comparing the stock classification to the results of ground filtering algorithms that are available in GIS packages, in representative testing tiles depicting various types of environments across the Netherlands. Luckily, many such packages are readily available publicly over the Internet. Some examples are the PDAL library which offers algorithms like the Progressive Morphological Filter (PMF) and the Simple Morphological Filter (SMRF), the CloudCompare software package which offers the cloth simulation filter (CSF) algorithm, and LAStools, whose modules such as the lasgound module (and lasground_new) can be used for ground filtering.

Our final set of testing tile consisted of 5 representative tiles we clipped from the AHN3 raw point cloud. In these 5 testing tiles, we tried to capture different sceneries that we will come across when processing wider areas, and which may represent challenges to our algorithms. The five locations that we chose

Code	Meaning
0	never classified
1	unclassified
2	ground
3	low vegetation
4	medium vegetation
5	high vegetation
6	building
7	low points(noise)
8	reserved
9	water

Table 2.1: The first 10 classification code numbers of AHN3 raw point clouds. More codes exist, but they are not listed here

Ledoux et al. [2020]

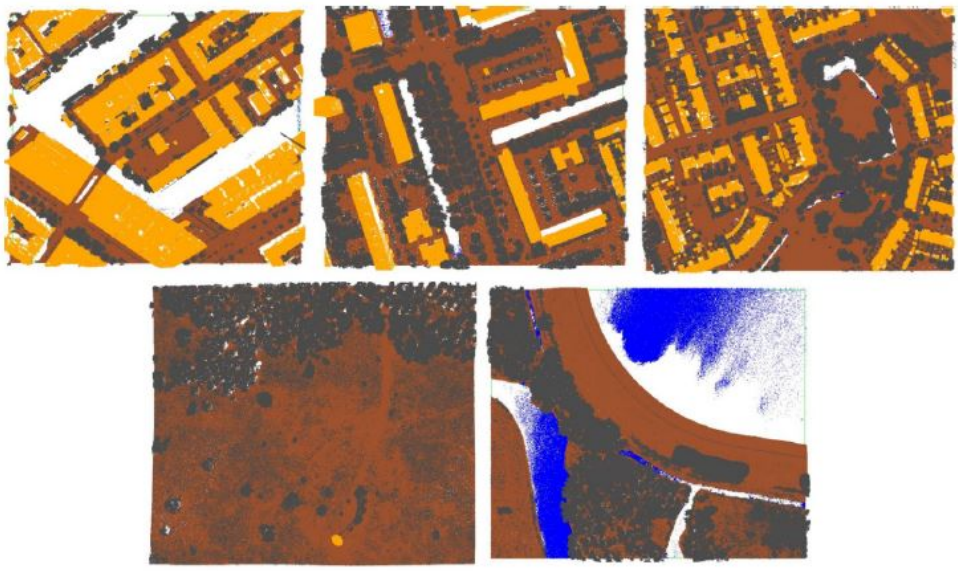


Figure 2.1: Data samples: top-left Amsterdam, top-middle Delft, top-right Groningen, bottom-left National Park Veluwezoom and bottom-right National Park De Biesbosch

are the city centre of Amsterdam (city centre), two less densely populated areas in the cities of Delft (city) and Groningen (city outskirts), a part of the National Park Veluwezoom (forests and heathland) and a part of the National Park De Biesbosch (willow forests, branching river system, wet grasslands and fields of reed). Bell et al. [2020], Zhang et al. [2016a] and Isenburg [2016].

The assessment of the quality of the AHN classification we conducted consists of the following steps:

- Filtering out points that belong to the following classes before applying some of the algorithms: "buildings", "low points" and "water". This led to better results.
- Applying the available ground filtering algorithms on the AHN raw point cloud samples.
- Examining the results and comparing them with AHN ground points.
- Writing a statistical and visual comparison report of the results.

The results of the assessment indicate that all these algorithms we tested, except for CSF, can provide us with a satisfactory classification in a general sense. However, their results each have various flaws relative to the stock classification in some subset of our testing tiles, as can be seen in Figure 2.1. For instance, LAStools's lasground module delivers good results in general, but with too many erroneous point classifications in urban areas like Amsterdam - where the stock classification evidently does not have these issues. On the other hand, both PDAL algorithms produce good quality results that can even rival the stock classification, but only when we remove points classified as water at the beginning of the PDAL pipeline.

PDAL is based on an internal pipeline; this is not the same pipeline we refer to in terms of our project in general. More information about this can be found on their website at PDAL. Pipelines in PDAL allow developers - us - to pass a single JSON configuration of the parameters and order of operations to execute on the data, in addition to the I/O configuration. In addition to the ground filtering algorithms we tested, we made use of the following filter operations in our PDAL pipelines, which we refer to in this report as pre-processing steps:

- filters.range: excludes some points that belong to certain classes from the whole process and achieve better results. It is also used again at the very end to store only the points classified as ground in the result.
- filters.assign: reclassifies all points to the same class.

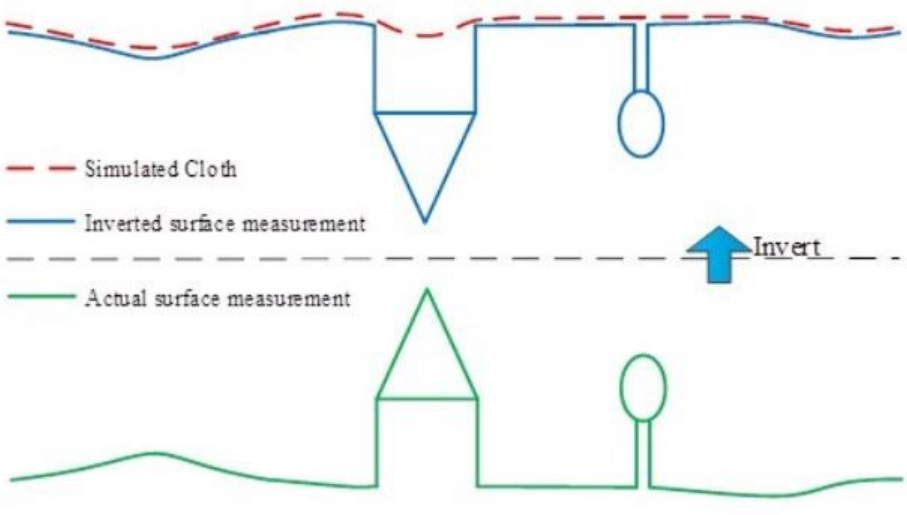


Figure 2.2: Overview of the CSF algorithm
Zhang et al. [2016b]

- filters.elm: performs noise filtering via the Extended Local Minimum (ELM) algorithm.
- filters.outlier: performs basic outlier filtering.

2.1.1 Ground Filtering Concepts

In this subsection, we provide a brief overview of the main concepts underlying the ground filtering algorithms we tested and compared with the stock classification.

CloudCompare's Cloth Simulation Filter (CSF)

According to the CSF algorithm, it is assumed that if we let a piece of cloth fall upon an upside-down terrain then the cloth will take the shape of the DTM and the shape of the DSM in the opposite case. Thus, with this algorithm, we extract the ground points from a point cloud by turning it upside down first and then dropping the cloth over it. Finally, the shape of the cloth can be determined and used as a base to classify the point cloud points into ground and non-ground points through an analysis made on the intersections between the points of the cloth and the point cloud Zhang et al. [2016b]

PDAL's Simple Morphological Filter (SMRF)

The workflow of the SMRF algorithm can be divided into four parts. Firstly, the minimum surface is generated by splitting the extent of the point cloud into cells and finding the lowest points within these cells. Then in the second part of the process, the cells that belong to bare earth are identified. This part is subdivided into four steps. The first step is to create a copy of the minimum surface. The second step is to create a vector of window sizes based on the supplied maximum which increases from one by one pixel to the ceiling of the maximum value divided by the cell size. The third step is to calculate an elevation threshold for every window size in the vector, create a new surface by applying a morphological opening on the minimum surface, add to the set of flagged ground cells any cell for which the difference between the minimum surface and the new surface we created is greater than the threshold we calculated and set the minimum surface equal to the new surface. The fourth and final step is to locate low outliers in the minimum surface. Moving on, the third part of the process is concerned with the creation of a provisional DEM by retaining the cells from the minimum surface that were identified as bare earth. Lastly, in the final part of the process, the points in the input point cloud

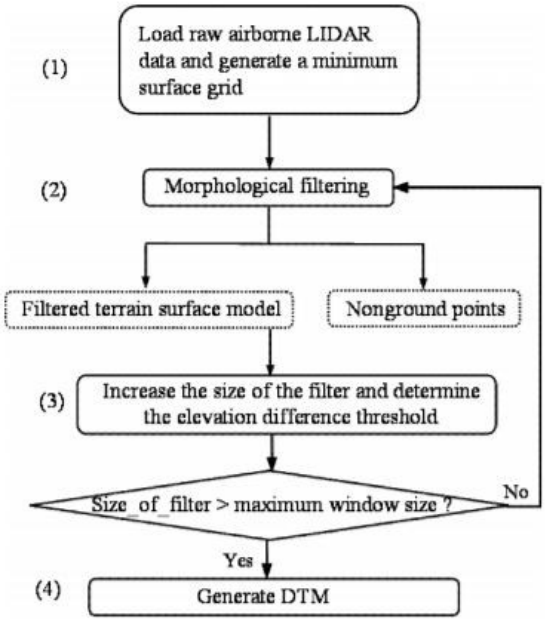


Figure 2.3: Framework of the progressive morphological filter
Zhang et al. [2003]

are classified into ground points based on a required vertical distance parameter and an optional scaling parameter Pingel et al. [2013]

PDAL's Progressive Morphological Filter (PMF)

To give a brief synopsis, the Progressive Morphological Filter (PMF) is an iterative process that creates an initial filtered surface by applying an opening operation with a window of a specified length on the point cloud. Non-ground features that are smaller than the length of the window are removed and the in the next iteration the length of the window is increased. To overcome the problem that the filtering process tends to incorrectly remove measurements at the top of high-relief terrain elevation difference thresholds based on elevation variations of the terrain, buildings and trees are used.

The framework of the PMF algorithm can be divided into four steps. The first step is to superimpose a grid over the point cloud, select the minimum elevation in each cell of the grid and generate the minimum surface. For steps two and three an iterative process begins where we apply the progressive morphological filter to the grid surface. Initially, we have an input filtering window and in each iteration the filter is applied on the resulting surface of the previous iteration. The filtering window is increased using elevation difference thresholds for every iteration. Finally, the DTMs are generated by removing the non-ground points Zhang et al. [2003].

2.1.2 Statistical Evaluation of Ground Filtering

To statistically evaluate how reliable the ground filtering algorithms and the stock AHN3 classification are, we used three small areas of 50x50 meters from the samples of Amsterdam, Delft and the National Park De Biesbosch, see Figure 2.4. We manually classified all the points in these three small areas and compared these (the true classification) with the ground points that the ground filtering algorithms give as a result when they are applied on these small sample tiles (predicted classification). The precision recall and F1-scores of the ground filtering algorithms were computed within the 3 small clipped areas. These values indicate how efficiently they can select ground points from the point cloud.

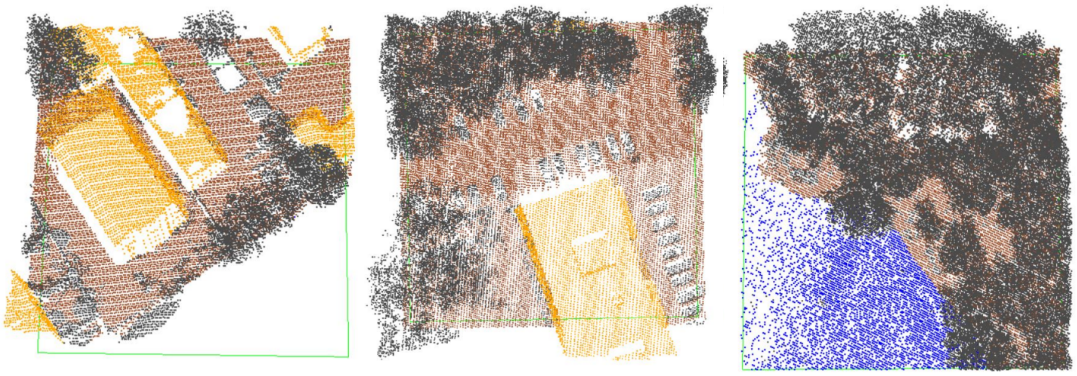


Figure 2.4: manually classified PointCloud used as Ground truth for statistical tests

```
{
    "type":"filters.elm"
},
{
    "type":"filters.outlier"
},
{
    "type":"filters.range",
    "limits":"Classification[2:2]"
},
```

Figure 2.5: Final pipeline for ground filtering using PDAL. We keep the points classified as ground by the AHN classification and remove potential outliers and noise points.

2.1.3 Ground Filtering Evaluation Results

In this subsection, we present the results of the visual and statistical evaluation of the PDAL results on which our decision regarding ground filtering was based on. The other two software packages were mostly excluded from this discussion based on the evidently much worse overall quality of their results as evidenced by the initial visual comparison.

The two available PDAL algorithms do not differ much in terms of the results they produce. Both approaches deliver similar results for most of the data samples we have but the second option comes with some flaws. It is worth noting that the data sample of Amsterdam was the most problematic since it has more objects and street furniture are present as well as ships and uniquely shaped buildings. The results of all ground filtering algorithms we tried produce the most erroneous classifications in this testing tile. Our final decision was based on an analysis we made on the results. We carried out a visual comparison of all the results as well as a statistical analysis by comparing them with some small areas we manually classified as ground and non-ground points. The F1-score was calculated for every ground filtering algorithm. Below we present our findings and we further discuss in more detail the flaws found in the results of the ground filtering algorithms when applied to the Amsterdam testing tile. The results of the complete set of ground filtering algorithms for all data samples can be found in the Appendix.

PDAL's SMRF algorithm delivered the best results according to point counts of all data samples we have, Table 2.2 shows only Amsterdam (the complete set of results are found in the Appendix). Four out of the five tiles had better results than the stock classifications, based on a quick visual inspection. However, although more points had been identified as ground for them, as shown in Figure 2.8, some small artefacts are visible even though we used a filter for outliers after the ground filtering algorithm. The Amsterdam tile suffered the most from this issue.

Location	Ground filtering algorithm	No. of ground points	Comments
Amsterdam	AHN3 classification	385651	--
	CloudCompare: Csf algorithm	303201	Less points are classified as ground, big holes are visible, the algorithm couldn't identify many ground points.
	LAStools: lasground	401356	More points are classified as ground but points that belong to boats and bridges are wrongly classified as ground.
	PDAL: filters.smrf	453767	Same as lasground tool but points from the roofs of some buildings are wrongly classified as ground as well (worse result).
	PDAL: filters.pmf	349188	Same artefacts as the smrf filter but with less points identified as ground.

Table 2.2: Amterdam data sample - point counts of each algorithm (results of all data samples can be found in the appendix)

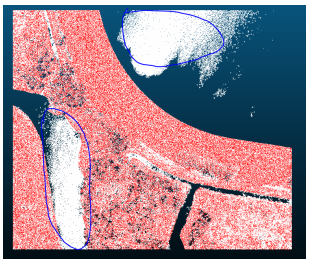
In the following figures we present some the artefacts we could not avoid producing by changing the ground filtering algorithm parametrisation, without introducing other issues in the process. Firstly, almost all water points in all cases were classified as ground points, as shown in Figure 2.6a. This challenge could be easily overcome by filtering them out before starting the algorithm (based on the stock classification, which identifies them accurately).

The most difficult challenge that we faced were ships and boats in the canals of Amsterdam, as seen in Figure 2.6b. Their points belong to two different stock classes, unclassified points and buildings. Filtering out those two classes was not a solution as we would have lost a lot of our potential ground points. It is something that we could also not solve no matter how we adjusted the parametrisation of the algorithm. These curcumstances influenced our final decision on the approach for the ground filtering part of our project to a great degree. A further challenge was that building points were classified as ground in many cases. Changing the parameters could reduce the amount of building points that were erroneously classified as ground, as shown in Figure 2.6d. However, to completely overcome this issue, we would have had to remove all building points in the pipeline, which was not an approach we agreed with.

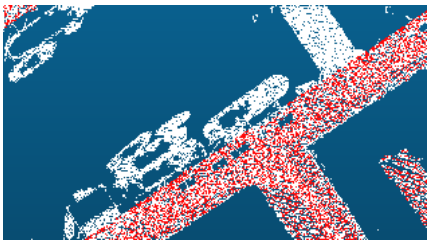
Another challenge was to decide whether we were going to consider bridges as ground points and include them in the interpolation algorithm for the generation of the DTM. The stock classification uses a specific, unique code for them (26) and evidently does not use them for DTM interpolation, as can be seen in Figure 2.7. Our final decision was not to consider them ground points either.

Lastly, it is also worth mentioning that in some cases we got much better results – locally – than the stock classification, like in the case of the National Park De Biesbosch testing tile, as shown in Figure 2.8. However, using two different ground filtering algorithms to have the best results everywhere is difficult to implement due to problems with the subsequent merging of the two sets of results, and we did not consider the added development complexity worth the slightly better potential quality of the results. Moreover, F1-scores of the ground filtering algorithms indicate that while more ground points were detected in these tiles by our PDAL configuration, this did not necessarily improve the statistical descriptors of the results significantly.

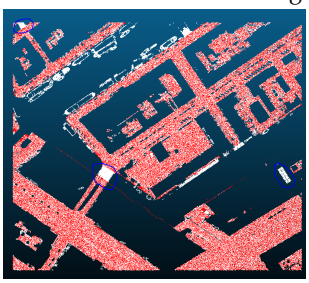
In fact, the F1-scores indicated that our best option would be to make use of the stock classification or - interestingly - the lasground module from LAStools. The ground filtering algorithms from PDAL turned



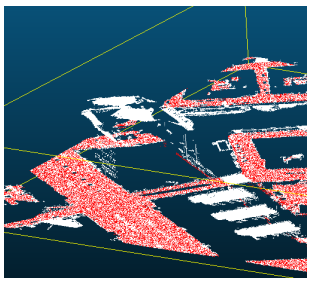
(a) Water bodies are classified as ground



(b) Boats classified as ground



(c) Overall result showing how an improvement was made (buildings artefacts removed) after many tries. However, ships, boats, buildings and bridges are still present.



(d) Buildings classified as ground, Red: AHN ground points.

Figure 2.6: White: PDAL's SMRF algorithm classification, Red: ANH classification

Figure 2.7: AHN3 raw point cloud (left) and DTM raster (right). Bridges are not included in the DTM raster.

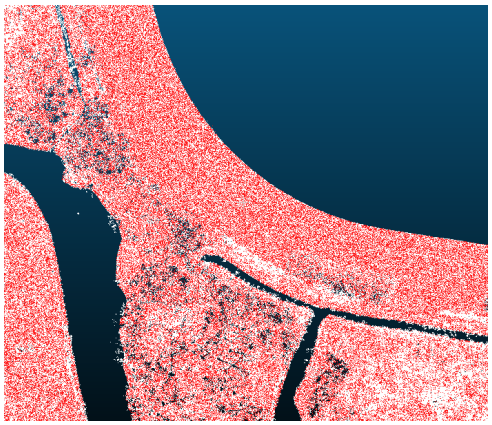


Figure 2.8: PDAL's SMRF algorithm result. Red: AHN ground points. 60,560 more points classified as ground.

Confusion matrix

	Actual ground points	Actual non-ground points
Predicted ground points	9089	119
Predicted non-ground points	31	15500

Confusion matrix

	Actual ground points	Actual non-ground points
Predicted ground points	11055	399
Predicted non-ground points	0	42561

Confusion matrix

	Actual ground points	Actual non-ground points
Predicted ground points	20321	268
Predicted non-ground points	0	32646

Figure 2.9: Confusion matrices of the AHN classification. Top: Amsterdam sample, middle: National Park De Biesbosch sample, bottom: Delft sample.

out to be relatively unreliable based on these metrics as they misclassify quite a lot of points as ground. For instance, in the case of the National Park De Biesbosch, shown in Figure 2.8, many vegetation points have been erroneously classified as ground. On the other hand, the AHN3 turned out to have a very good and trustworthy stock classification, as Figure 2.9 illustrates. Table 2.3 presents the F1-scores of all ground filtering algorithms according to the manually classified point cloud samples we created. This further decreased our confidence that it would be worth including PDAL-based ground filtering in the final pipeline.

2.1.4 Ground Filtering Decision

The results of our assessment left us with two options.

1. Either we use the stock AHN3 classification which in general is more reliable than any of our own results, if we first apply some improvements (pre-processing steps) like filtering out outliers and noise using PDAL.

Location	Method	F1-Score
Amsterdam	AHN3 classification	0.9918158009602793
	CloudCompare: Csf algorithm	0.45612311722331367
	LAStools: lasground	0.949598169141519
	PDAL: filters.smrf	0.8282780597191077
	PDAL: filters.pmf	0.7992817750416826
Delft	AHN3 classification	0.9934490344659007
	CloudCompare: Csf algorithm	0.007741061493057235
	LAStools: lasground	0.9843911058754233
	PDAL: filters.smrf	0.7306581453942113
	PDAL: filters.pmf	0.6991125288191264
National Park De Biesbosch	AHN3 classification	0.982273757163801
	CloudCompare: Csf algorithm	0.22917076368403808
	LAStools: lasground	0.8154150777354062
	PDAL: filters.smrf	0.2507851054284433
	PDAL: filters.pmf	0.2111231940344881

Table 2.3: F1-scores of ground filtering algorithms and AHN classification.

2. Or, we apply a ground filtering algorithm via PDAL and make some use of the stock classification along the way to remove, for example, water points from the whole process before starting the ground filtering algorithm.

While the above analysis had proven to us that the stock classification has reliable quality, especially for ground points (class 2), these ground points turned out to contain a great number of outliers and noise points. This is why the first option above includes the necessity to do a round of PDAL-based filtering. This is necessary to make sure we have a clean input for the interpolation stage, of which following section gives an overview.

After conducting our tests with the various available ground filtering algorithms, we decided that our best bet is the first. The results show that the differences are not significant enough to justify keeping a ground filtering algorithm in our PDAL pipeline, which would have also imposed significant additional computational complexity on the overall pipeline of the project. The stock classification has admirable overall quality, and it can certainly be trusted for the purposes of the subsequent operations (interpolation, hole-filling and hydro-flattening). Thus, the final result of this stage of the project is the PDAL pipeline configuration shown in Figure 2.5, which we refer to as pre-processing throughout the rest of this report.

2.2 Interpolation Stage

The stock Current Dutch Elevation (Actueel Hoogtebestand Nederland 3, AHN3) map uses the Squared-IDW interpolation method to produce gridded DTMs and DSMs at 0.5 metre horizontal and vertical scale. For the generation of the DTMs only points classified as ground (class 2) are used while for DSMs all points are used except those classified water (class 9) are included AHN [2020]. The DTMs/DSMs offered by AHN3 are freely available as open data in the GeoTIFF format, and PDOK's download facility is used for their distribution, (available at link). However, the generated rasters contain far too dense a distribution of artefacts, mostly holes (groups of no-data values) where buildings, vehicles, vegetation, and near water bodies were imaged. In general, we may say that these are mainly the result of using

Figure 2.10: A part of the AHN DTM tile M_25GN1 (Amsterdam's city center). The raster image contains artefacts (holes) where there are buildings and water bodies.

the Squared-IDW interpolation method which is by definition a discontinuous interpolant (owing to the underlying radial neighbour queries that it is based on). Figure Figure 2.10 shows examples of such artefacts in stock AHN3 rasters.

This leads us to the discussion of the main aim of this project, which is to improve the overall quality of the stock AHN3 rasters. We tested and compared the results of an ensemble of 6 interpolation implementations with the aim to fix various aspects of the current issues of AHN3's DTMs and DSMs. Furthermore, as per the request of our client, particular attention was paid to how building footprints, stationary water bodies, and large rivers are represented in the output rasters. In this section, we focus on providing an overview of the concepts underlying the interpolation methods we implemented, along with some details about their implementation and the discussion of the visual and statistical evaluation and the comparison of their results. Finally, we present our final decisions regarding which exact interpolation methods (and parametrisations) we chose to pass on to the scaling team of our group to be implemented in the server-based environment in a manner to allow the efficient (multithreaded) processing of the full set of AHN3 data.

To be able to select good candidates for our ensemble of interpolation methods, we carried out a comprehensive literature review. We identified known theoretical advantages and disadvantages of these interpolation methods, as shown in Figure 2.11 and Table 2.4. In the context of this project, we knew that the final candidate would be put to work on a point cloud that covers a large area (South Holland, potentially the whole Netherlands i.e. the entire AHN3 data set) which meant that efficiency had to be our main concern (beside quality). Therefore, we disregarded methods associated with high computational and development complexity, such as Kriging methods. Instead, we implemented and compared methods that are better in both these departments, or simply put, which are simpler and faster. In order to identify which one of them delivers the best results in a reasonable amount of time, we conducted timing tests, as well as visual and statistical analyses to serve as the basis for the comparison of their outputs. The next sections will introduce the methods we implemented in our testing environment.

Inverse Distance Weighting (IDW)

The inverse distance weighting interpolation method computes the value of unsampled locations (the interpolation points) using the distance-weighted values of nearby data points.

The weight $wi(x)$ assigned to each pi neighbor for an unsampled location x is calculated via the following equation:

$$wi = |xpi|^h \tag{2.1}$$

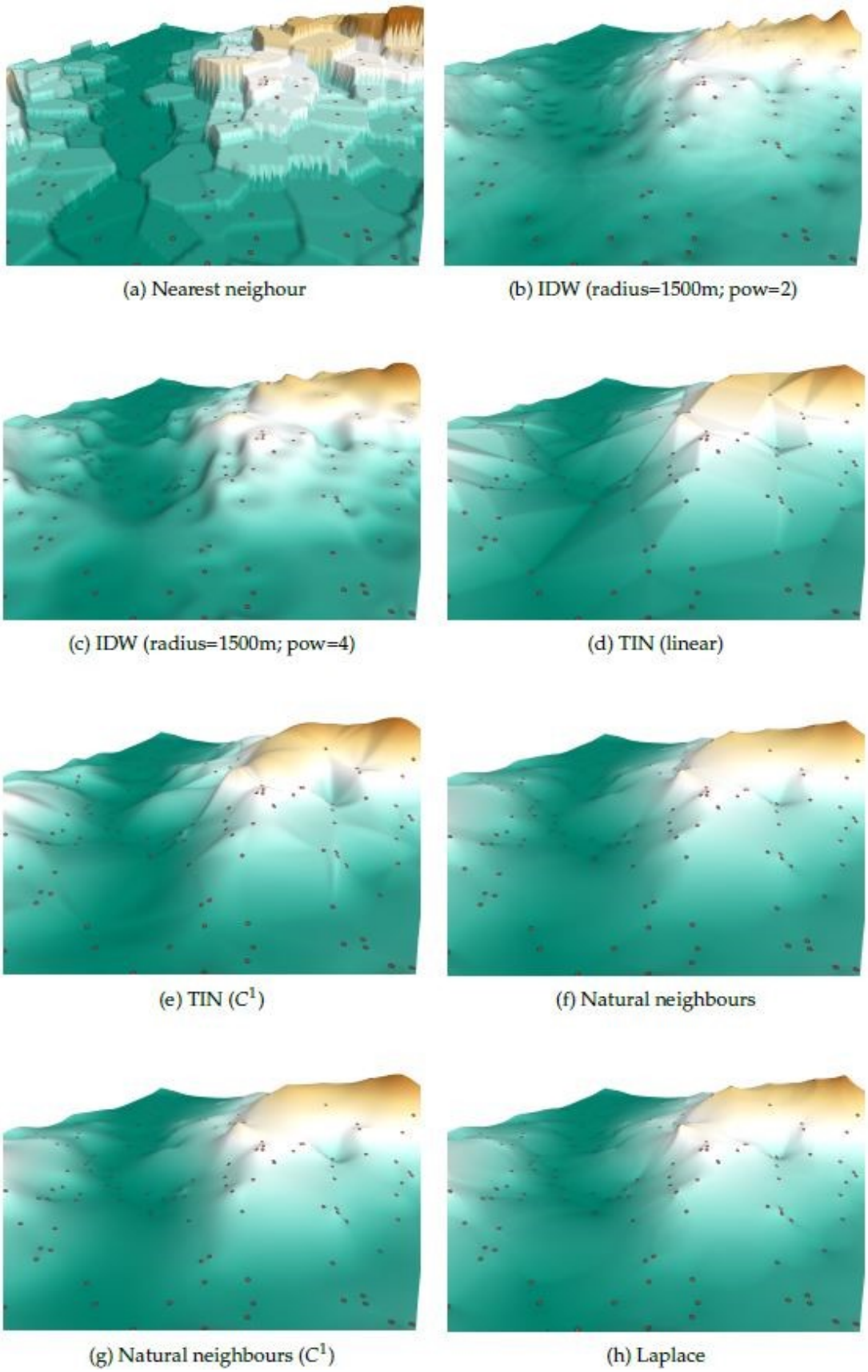


Figure 2.11: Visual comparison of the results of a few interpolation methods for the same dataset. The samples are shown on the surface (red dots).
Ledoux et al. [2020]

	exact	continuous	local	adaptable	efficient	automatic
global function	×	C^{2+}	×	–	–	×
splines	×	C^{2+}	depends	0	-	×
nearest neigh.	✓	×	✓	+	++	✓
IDW	✓	×	✓	-	0	×
TIN	✓	C^0	✓	+	++	✓
NNI	✓	C^0	✓	++	0	✓
NNI-c1	✓	C^1	✓	++	-	✓
Laplace	✓	C^0	✓	++	+	✓
bilinear	✓	C^0	✓	++	++	✓

Table 2.4: Overview of the characteristics of some interpolation methods in general. Ledoux et al. [2020]

where h is the power defined by the user, —xpi— is the distance between the unsampled location x and nearby data point pi.

The higher the power h is, the higher the weight is that points close to the unsampled location get. Most commonly, the nearby data points are those found in a query radius around the interpolation point. The size of the radius influences greatly the result of the interpolation. A very big radius leads to a smooth or flat result while a small radius results in a rough surface Ledoux et al. [2020]. The IDW interpolation method is typically used to generate rasters from dense point clouds with a homogeneous spatial distribution of data points. As already explained, where this assumption is violated, it may generate holes depending on the parametrisation. This is the result of the fact that uses a finite radius, which means that it will not yield values at the interpolation locations if there are no neighbours (nearby data points) within that radius. We commonly refer to this main type of IDW as ”radial IDW”.

Certain variants of the IDW method assign the interpolation weights simply using n closest neighbours, or n closest neighbours from cells of some subdivision of space centered on the interpolation point (most commonly quadrants). By definition, such variants do not generate holes as (apart from degenerate cases) they will always find closest neighbours irrespective of the inhomogeneneity of the distribution of input points. In this project we implemented such a variant, which we refer to as ”quadrant-based IDW”. We implemented two sub-variants, one fetches k-nearest neighbours, while the other is built on top of radial queries. Both variants iteratively query more and more neighbours (by fetching more neighbours and expanding the query radius, respectively), until a pre-defined minimum number of points per quadrant have been reached. To limit the computational complexity, the underlying kd-tree queries were made approximate, and iteration limits can also be set in the parametrisation. This means that in our implementation, it is only *under ideal conditions*, i.e. using a parametrisation suitable for the specific input point cloud (or a tile thereof), that the method can be regarded continuous.

Linear Interpolation in Triangulation (TIN)

This interpolation method, which we generally refer to as ”TIN-linear” interpolation in this report, is popular for terrain modeling applications. It is the simplest of the methods that are based on creating a Delaunay Triangulated Irregular Network (Delaunay TIN, also called DT) having as vertices the measured data points. Interpolation takes place within the triangles, the values are computed as the weighted sum of the elevation of the vertices. The weights are the barycentric coordinates of the unsampled location within the triangle. To compute the value of an unsampled location, we first identify which triangle it is contained by, and then we compute the barycentric coordinates we mentioned above, which we can then use in the weighted sum. The main disadvantage of TIN-linear interpolation is that it does not, in its simplest form, produce smooth surfaces. It can easily be made continuous at the edges and vertices, but it is not differentiable there. As all TIN-based methods, this method is unsuitable for

extrapolation and so it cannot yield values beyond the convex hull of the input point cloud. It is, however, entirely automatic (no parametrisation required), and is efficient. Moreover, it takes into account all data points and it can handle anisotropic data distributions and/or extreme variations in the density of the data points, Ledoux et al. [2020].

Laplace and Nearest Neighbour Interpolation (NNI)

These methods are identical in terms of a top-level understanding of how they work. They are both based on the Voronoi Diagrams (also called DT, commonly extracted from the DT of the data points) of the input point cloud, and hence they can be made continuous. Like TIN-linear interpolation, they do not generate holes owing to this continuity, and they generally produce smoother results than TIN-linear. Otherwise, all their properties except for performance match with those of TIN-linear interpolation. When a point is inserted, the DT and VD change locally, and the weights are based on this change in both Laplace and NNI. When a point is inserted, it creates a new Voronoi cell, which captures areas of the pre-existing Voronoi cells. In NNI, these "stolen" areas stolen from neighbouring cells are used as weights, because each cell is associated with a single data point that has an elevation. In Laplace interpolation, the lengths of the edges of the new VD cell are used as weights instead. This makes Laplace significantly cheaper computationally than NNI. Ledoux et al. [2020].

Constrained DT-based (CDT-based) TIN-linear Interpolation

This method is similar to TIN-linear interpolation with the single difference that its DT contains constraints. CDTs are triangulations in which a selection of input edges are kept and included as triangle edges, these are what we call constraints. Hence, CDTs make it possible to "force" external geometries into a TIN and keep them there throughout the interpolation stage. For the purposes of this project, this property was deemed interesting and potentially useful, as it can ensure that the boundaries of certain entities, such as buildings, are respected in a stricter manner in the output, generally resulting in sharper boundaries in the rasters. Under perfect conditions, another advantage of this method is that it can be used to create holes inside the TIN that represent objects, in turn allowing these objects to be interpolated in a different way, but to still be made an integral part of the TIN underlying the primary interpolation.

Overview of Implementation

Efficiency was key to the realisation of this project and thus although we implemented everything in Python at the top level, we strived to base each interpolation algorithm at least in part on binary software packages. Most of what was implemented as Python code, was also based on packages which are known to be optimised in terms of computational complexity, such as NumPy. The complete list of packages we used is as follows: CGAL, Fiona, Laspy, NumPy, OWSLib, PDAL, Rasterio, SciPy, Shapely, and startin. Below we list the interpolation methods we implemented along with details about and the software libraries/packages used, and some general details about the implementation. Further details can be found in the documentation of the relevant code repository. The input, for the purposes of the testing code framework, was always assumed to be a subtile of the AHN3 with known extents.

- Radial IDW Method with Fallback Kernel
  - Name of method in code framework: PDAL-IDW.
  - Entirely based on a PDAL pipeline (binary, C++-based), wrapped in Python.
  - A Python interpolation iteration is not involved.
  - It is indirectly based on GDAL's IDW implementation.

  - Fallback kernel: wherever radial IDW fails to interpolate, the program looks for pre-existing pixel values in surrounding pixels in a kernel with a predefined size centered on the pixel that is currently being interpolated.
  - Limited selection of parameters involved, performance is good.

- Quadrant-Based IDW Method (IDWquad)
  - It is based on SciPy's C-based KD-tree implementation, but uses our own Python iteration for the interpolation itself.
  - Uses approximate KD-tree queries, hence its performance is acceptable.
  - Uses NumPy array-based efficient slicing to find a set minimum number of points per quadrant around the location of interpolation.
  - As already explained above, both k-nearest neighbour incrementing, and radial search expansion sub-variants were implemented.
  - Its parametrization is complex and needs to be fine-tuned to keep the performance within an acceptable range.

- TIN-Linear and Laplace Methods
  - Names of methods in code framework: startin-TINlinear, startin-Laplace.
  - DT construction and interpolation via startin (binary, Rust-based).
  - The interpolation iteration is written in Python.
  - The startin DT and Laplace interpolant have exceptionally good performance.
  - No parameters involved.

- Natural Neighbor Interpolation (NNI) Method
  - Name of method in code framework: CGAL-NN.
  - Python bindings of CGAL (binary, C++-based) used to construct DT/VD and to interpolate.
  - The interpolation iteration is written in Python.
  - The CGAL DT and NNI interpolant have mediocre performance.
  - No parameters involved.

- Linear Interpolation in Constrained Delaunay Triangulation
  - Name of method in code framework: CGAL-CDT.
  - Python bindings of CGAL (binary, C++-based) used to construct CDT, and to insert the constraints.
  - The interpolation, apart from locating triangles in the CDT, is based on our Python code.
  - The constraint edges are extracted from polygons that are loaded from disk and through a WFS service via our vector tiling Python code.
  - No parameters involved. Constraint insertion and Python-based interpolation make it slower than the rest of the methods.

2.2.1 Testing Framework and DTM/DSM Interpolation

Overall, the interpolation task was divided into two sub-tasks, DTM interpolation and DSM interpolation. The programming framework of the testing environment was built in such a way that it can serve both DTM and DSM generation through all the above implementations via a single entry point, configurable using command line arguments. As a result, both DTM and DSM interpolation testing through all interpolation methods could follow the exact same workflow. The pre-processing operation, described in the previous section, was also implemented as a PDAL pipeline in this framework, so that it is an optional part of the main testing pipeline. Post-processing, which will be introduced in the next section, was also built into this framework, hence the entire processing chain can be launched through a single command line command with the appropriate configuration of arguments. This helped streamline our testing procedures. Furthermore, the implementation also offers native multiprocessing, which allowed our testing team to not only run the full testing pipeline (pre-processing + interpolation + post-processing) as a single operation, but to do so for multiple testing tiles depicting various types of environments (most commonly the 5 tiles we have introduced in the previous section), using parallel processing. This allowed them to see what the effects of a change in their testing configuration is, in combination with all other operations, and in all environments, at the same time - without the need to segment their testing workflows into testing different pipeline stages and environments at different times. The complete testing code framework and documentation are available in our code repository.

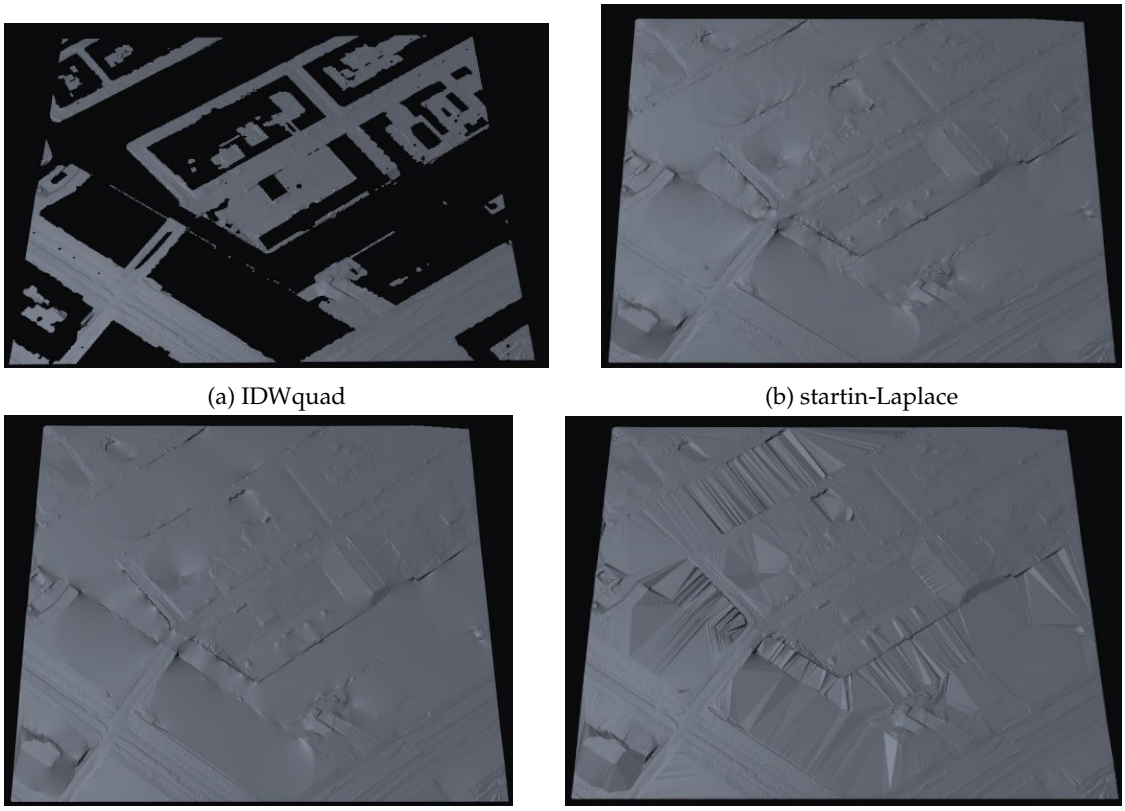
The main difference between DTM (Digital Terrain Model) and DSM (Digital Surface Model) configurations is that DTM interpolation excludes all points that are not classified as ground points. As a result of this, the interpolation problem changes entirely (due to the resulting large gaps in the input point cloud), meaning that it is not unreasonable to expect different solutions to be optimal for these two types of rasters. All other differences are in how we proposed to post-process them, which will be explained in later sections. Although the output file type and pixel size is user-configurable in the testing environment, all outputs shown here are GeoTIFF files with the same pixel size as the AHN3 rasters (0.5 metre horizontally and vertically). To visualise the results in 2D and 3D we used QGIS, ArcMAP and Blender (via the BlenderGIS add-on).

For the statistical analysis and comparison of the results we wrote our own Python scripts, mainly based on the NumPy, Laspy and Rasterio packages. We computed the vertical differences between the resulting raster images and the input point cloud, and using this, the Root Mean Square Error (RMSE) and Mean Absolute Error (MAE) distribution in the outputs of each interpolation method. This provided statistical constraints to our evaluation procedure, which was otherwise primarily based on timing and the visual assessment of correctness. The same set of testing tiles was mostly used that we have already introduced in the section describing ground filtering/pre-processing.

DTM - Visual Analysis of Interpolation Results

To generate the DTMs, as we already explained in the ground-filtering/pre-processing section, we decided to rely on the stock AHN3 classification. The interpolation algorithms were preceded by the application of the "filters.elm" and "filters.outliers" pre-processing PDAL filters in each case. These PDAL filters ensure that the input point cloud for DTM generation has no outliers or noise points. In our implementation, PDAL can pass the results of these operations directly to NumPy, so there is no I/O operation involved as a result of applying pre-processing. In this section we present the outputs of our final configurations of each method, on the following three data samples: Amsterdam, Delft, and the national park of De Biesbosch, in Figures 2.12, 2.14 and 2.13. All other final testing results (including 2D raster images) can be found in the Appendix.

The PDAL-based radial IDW interpolation method (with fallback kernel) is not generally continuous. The interpolated pixel gets its value from points within a search radius of the interpolation point, where the radius, inverse distance power, etc. can be configured by the user. While the fallback kernel can patch in small holes where there is enough neighbourhood information to work with, this is not the case for larger holes, which radial IDW often generates. As a result of this, the output of this method has holes. In Figures 2.12a, 2.14a and 2.13a we can see that these holes are found where buildings and



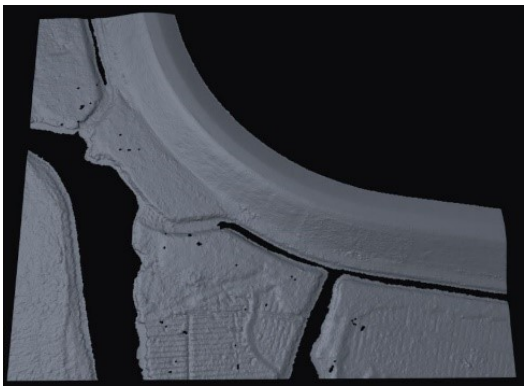
(a) IDWquad

(b) startin-Laplace

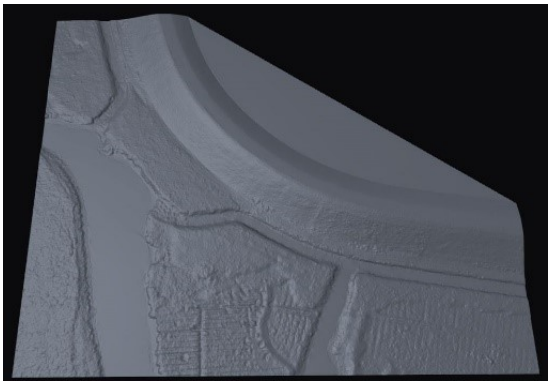
(c) CGAL-NNI.

(d) startin-TINlinear.

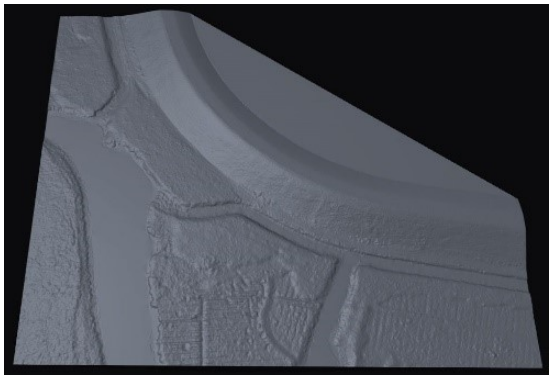
Figure 2.12: Amsterdam DTM; 3D visalisation.



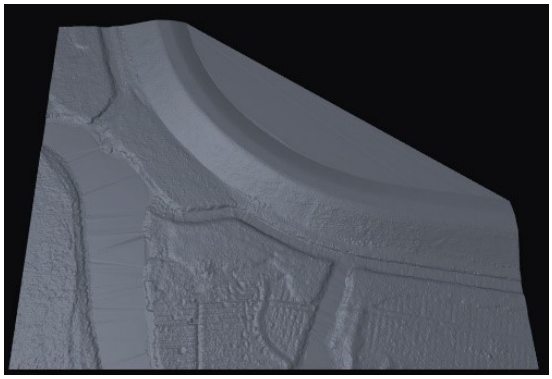
(a) IDWquad



(b) startin-Laplace

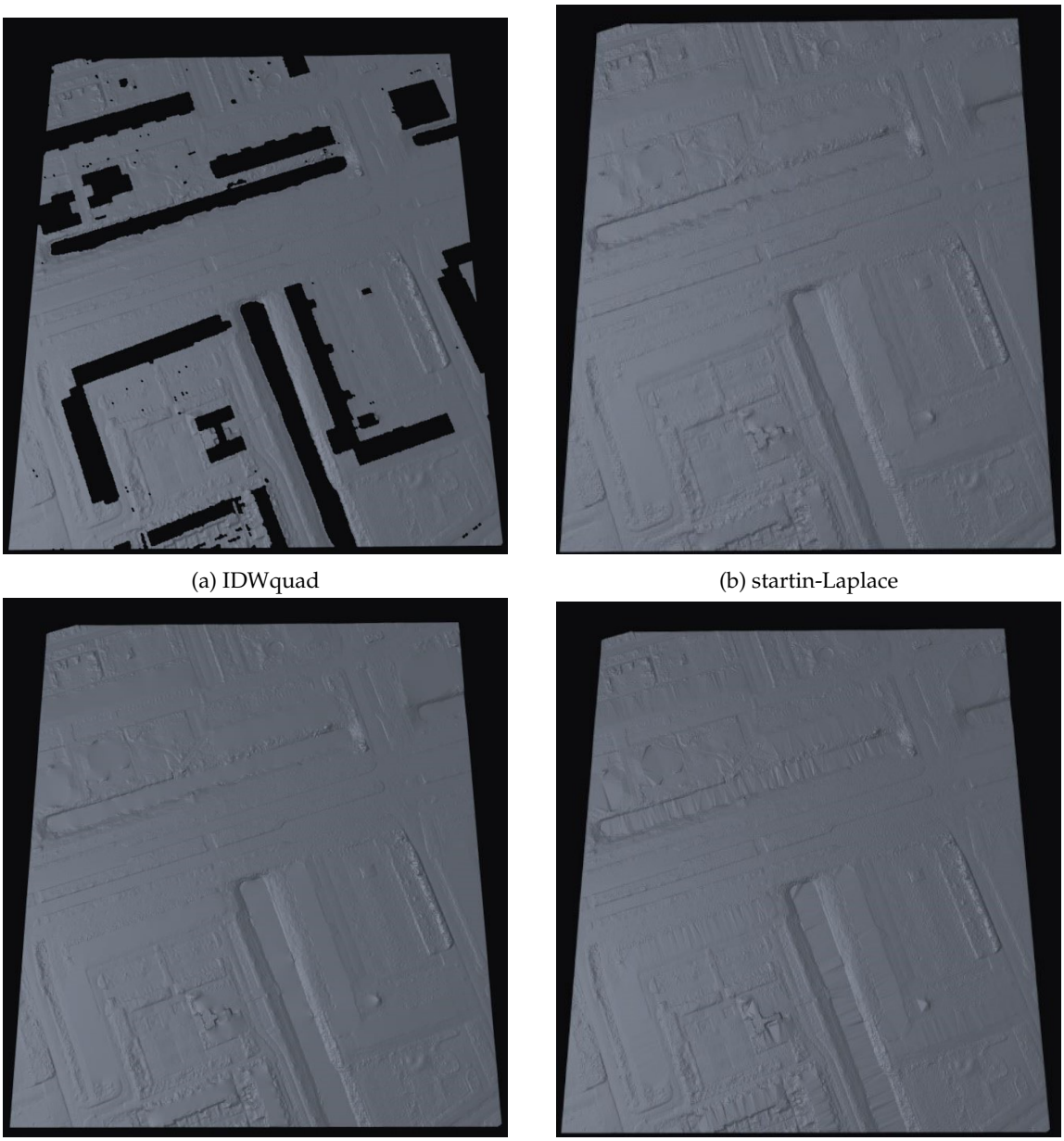


(c) CGAL-NNI.



(d) startin-TINlinear.

Figure 2.13: De Biesbosch DTM; 3D visualisations.



(a) IDWquad

(b) startin-Laplace

(c) CGAL-NNI.

(d) startin-TINlinear.

Figure 2.14: Delft DTM; 3D visualisations.

water bodies are found. Hence, we excluded PDAL-IDW from further consideration (and development), because it shows no potential to produce raster that surpass the quality of the stock rasters of AHN3.

In general, our results show that quadrant-based IDW interpolation is far more continuous than simple radial IDW. It is conceptually possible to set up the parametrisation to provide perfect continuity, but in practice it would be extremely difficult to write a program that can intelligently self-configure the parametrisation to achieve full continuity, not to mention the toll this might take on performance. Hence in practice, we needed to keep the parametrisation unchanged across tiles and therefore a compromise needs to be made that worked best on our testing tiles overall. Since it needed to provide optimum results both in terms of quality and performance, our IDWquad results also include holes wherever the algorithm was not allowed to complete enough iterations to collect enough points per quadrant to interpolate (mainly due to the iteration limits we needed to set).

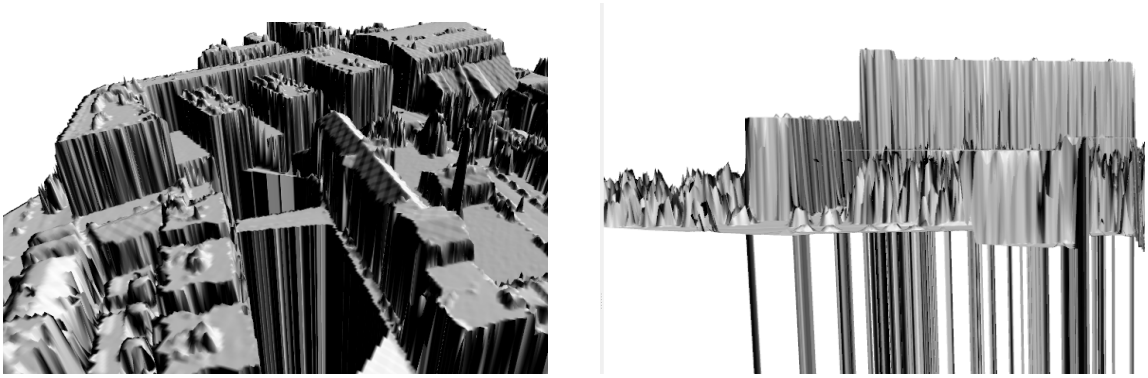
Our final CDT-based (CGAL-CDT) outputs used the boundaries of BAG building polygons and BGG water polygons as constraints in the triangulation. Even though this method delivered good results overall and helped produce sharper building and water boundaries in the rasters (see the next subsection about the DSM results), it failed to create true holes in the triangulation which would have been the aim for DTMs. As already mentioned, our desire was to use the boundaries of buildings and water bodies to introduce holes into the triangulation that could then be be separately worked on later, using alternative methods of interpolation. While this should have worked in theory, we still found valid triangles inside our CGAL-CDT holes which contained enough non-constraint vertices to perform interpolation. This is a result of the fact that we cannot afford (in terms of computational complexity) to remove Lidar points from within the polygons that are used for the hole creation, which unfortunately nucleate too many valid triangles inside the holes for this approach to be useful. Hence, like PDAL-IDW, CGAL-CDT was eventually excluded from further consideration in terms of DTM generation.

On the other hand, all triangulation-based methods (TIN-linear, Laplace and NNI) are indeed continuous interpolants in practice, because they use a full tessellation of the convex hull of the input points. The only exception to this rule are areas outside the convex hull around the borders, which we proposed to solve by merging overlapping tiles (see later in the scaling-related sections). As can be seen in our results, these methods indeed ”fill holes” automatically and therefore are more desirable to us, owing to the fact that the biggest problem with the current AHN3 DTMs is that they contain an unreasonable amount of holes. However, when using TIN-based continuous interpolation to fill large holes, the interpolated surface may visually exhibit the shapes of the underlying TIN surface (i.e. colour-graded triangles, in effect), especially simple TIN-linear triangulation. Meanwhile, NNI and Laplace interpolation methods produce smoother surfaces, as shown in Figures 2.12b, 2.14b and 2.13b 2.13c. Due to this, our initial thoughts based on this visual comparison was to use either the NNI or Laplace interpolation for the generation of DTMs. However, further analyses were needed before we could make an educated decision about the best candidate method and parametrisation to pass on to the scaling team.

DSM - Discussion of the visual results

While continuous interpolation methods delivered better results when generating DTMs, the opposite goes for DSMs. The input point cloud for DSM generation includes all manmade and natural objects present on the terrain. Thus, we needed a method that can handle sharp edges and generates as few as possible artifacts in an environment with complex-shaped objects. We are presenting the outputs generated with the same testing tiles as above to facilitate easy visual comparison. The results for all data samples and interpolation methods are found in the Appendix.

It is also important to note that the input point cloud for DSM generation is still pre-processed first using filters from PDAL even though it is not only ground points that are kept. The same filters (“filters.elm” and “filters.outliers”) are applied to ensure that any potential outliers or noise points are removed, as shown in Figure 2.20. Unfortunately, the raw AHN3 point cloud contains outliers that may in turn cause interpolation artefacts, so this pre-processing is essential. This is more severe an issue for DSM generation, because it appears that keeping only ground points already serves as a good way of getting rid of them, which is not an option for DSM generation. For instance, in the Amsterdam sample some



(a) Artefacts around river polygons (b) Further issues with CGAL-CDT results

Figure 2.15: Artefacts caused by the CDT interpolation.

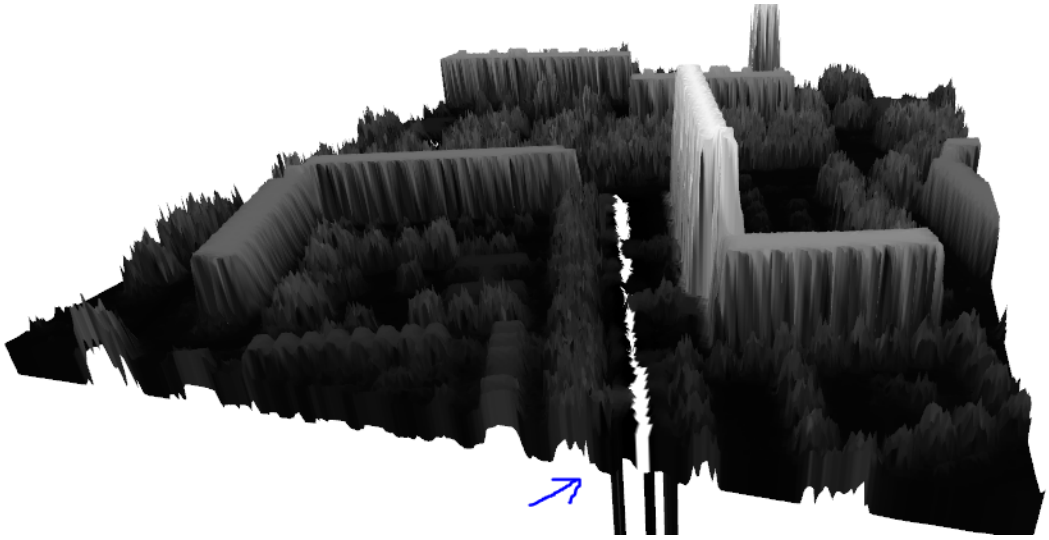


Figure 2.16: Holes present in DSM generated by IDW

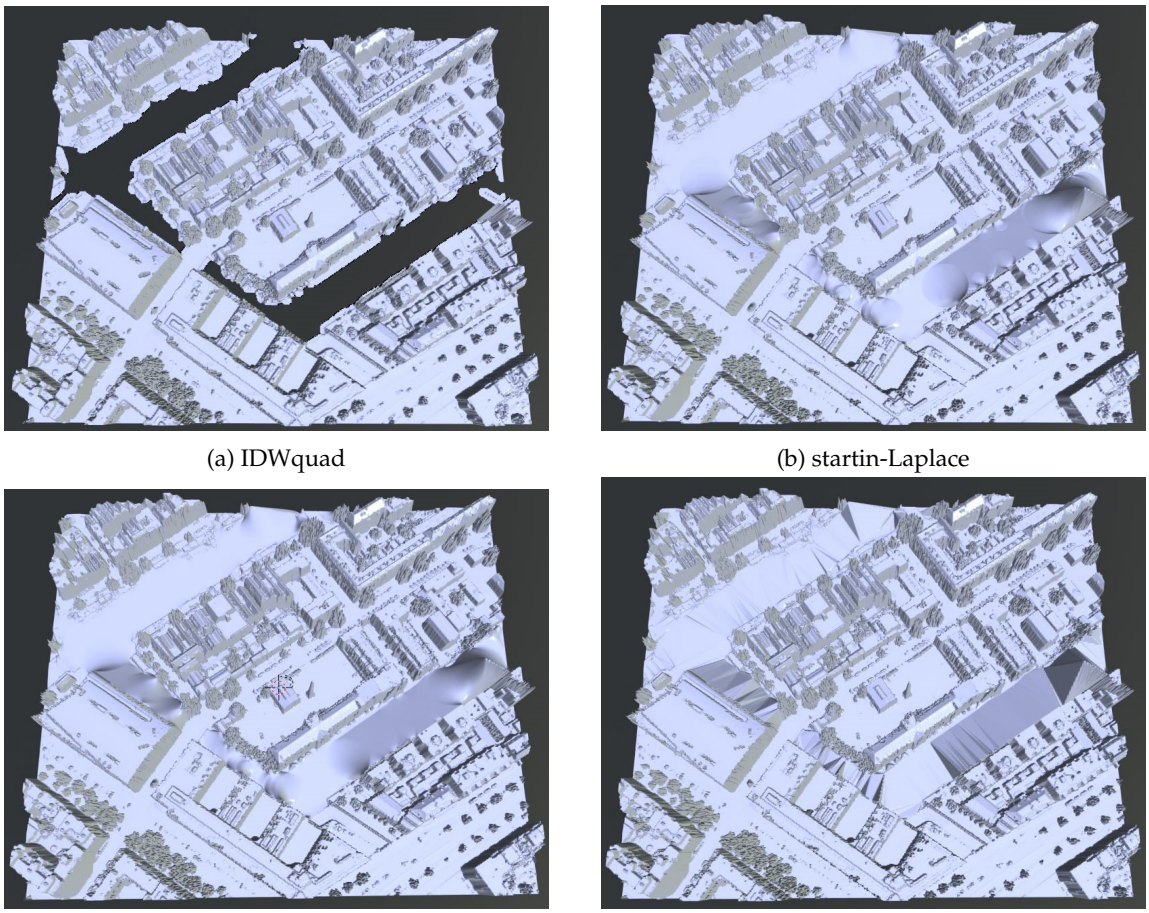
points floating in mid-air can be found, which we assume that they belong to a crane that was present during data collection, this is shown in Figure 2.21.

The CDT-based TIN-linear method (CGAL-CDT) was expected to produce sharp edges for DSM when using the boundaries of building and water polygons as constraints. While this improved the visual appearance of buildings in the outputs overall, using the boundaries of water bodies degraded rather than improved the quality of the appearance of water bodies in the outputs. While this problem could have possibly been solved in the post-processing stage, CGAL-CDT produced another type of artefacts around some buildings that had too few Lidar reflections or too low elevations, as shown in Figure 2.15. Combined with the high computational complexity of the method, these issues justified excluding it from further consideration in terms of DSM generation too.

PDAL-IDW was excluded for the same reason as the one mentioned in the DTM discussion section. Despite the presence of non-building points, it still generates far too many holes. The only difference is that, since building points are included, holes are bigger and more common around water bodies, as shown in Figure 2.16.

To summarise, out of the six interpolation methods that we tested, the following ones produced the most satisfactory results:

- IDWquad: radius = 1, power (exponent) = 2, minimum number of points to find per quadrant = 1, query radius: 3, tolerance = 2, and max. no. of iterations = 4 (parametrization explained in detail



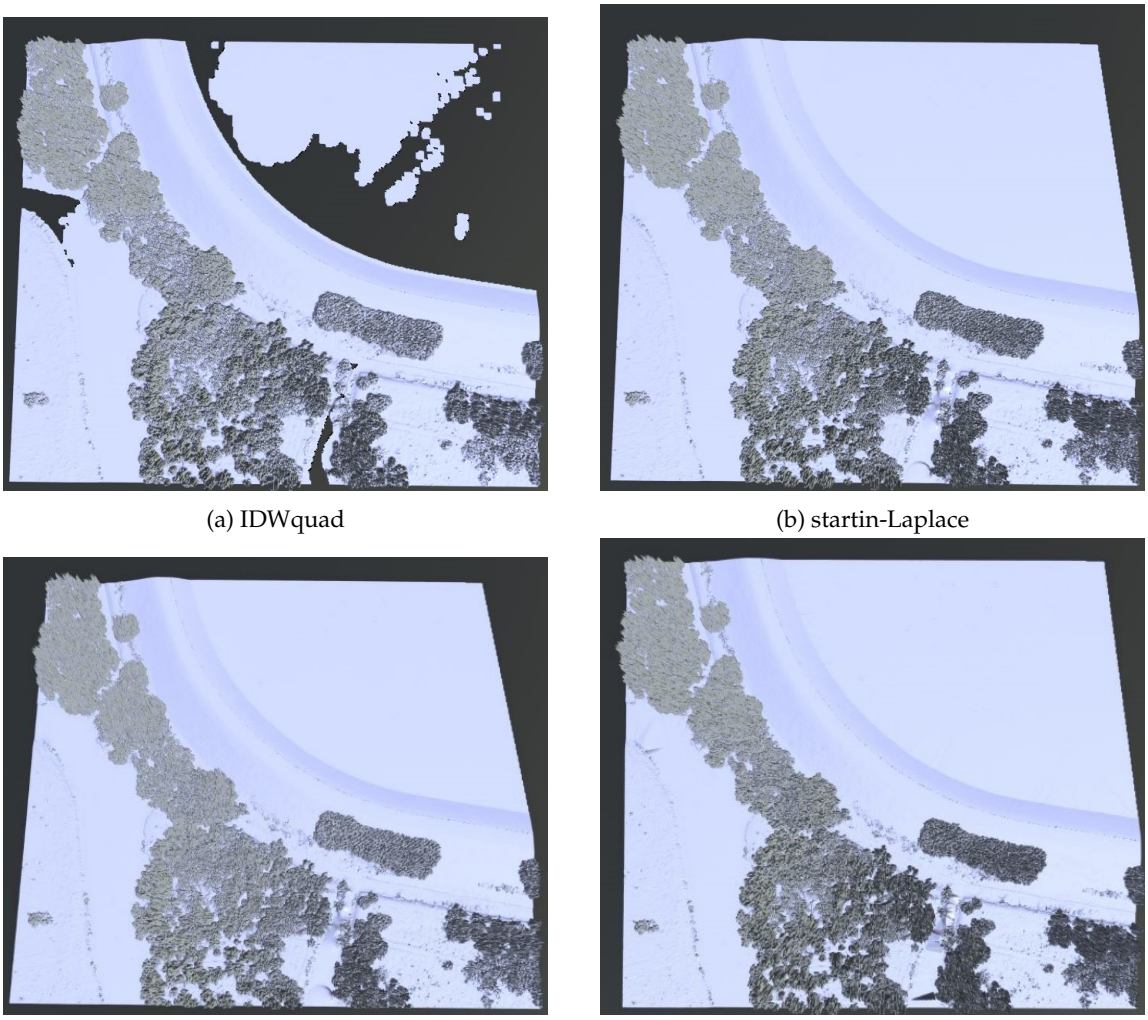
(a) IDWquad

(b) startin-Laplace

(c) CGAL-NNI.

(d) startin-TINlinear.

Figure 2.17: Amsterdam DSM; 3D visual results.



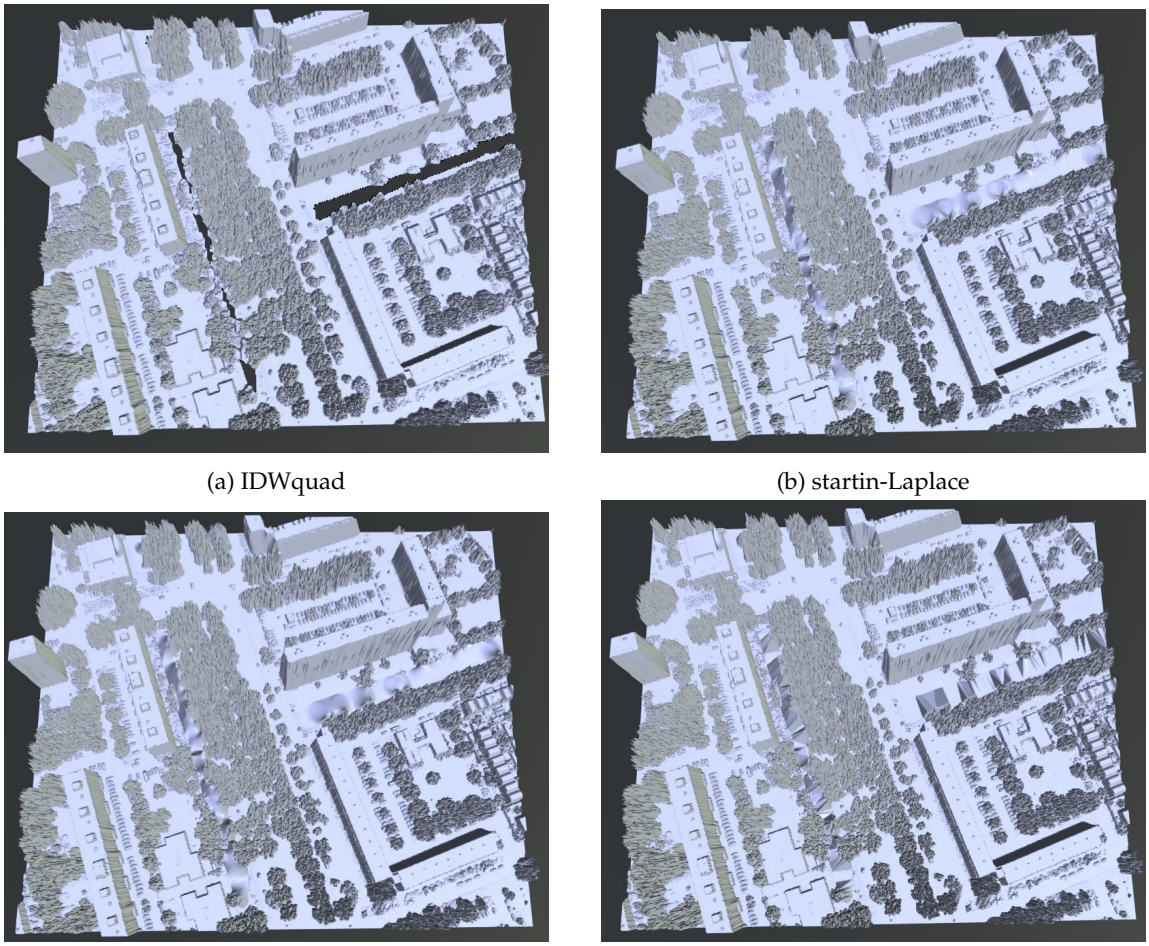
(a) IDWquad

(b) startin-Laplace

(c) CGAL-NNI.

(d) startin-TINlinear.

Figure 2.18: Biesbosch DSM; 3D visual results.



(a) IDWquad

(b) startin-Laplace

(c) CGAL-NNI.

(d) startin-TINlinear.

Figure 2.19: Delft DSM; 3D visual results.

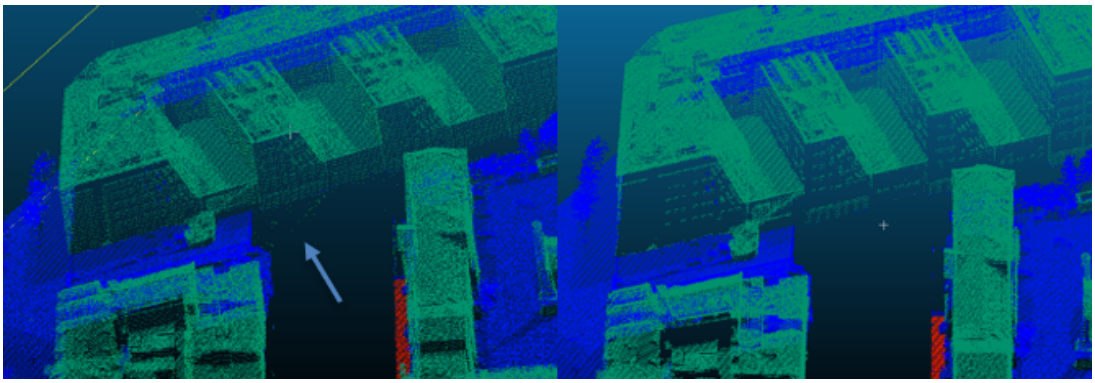


Figure 2.20: Removing outliers using PDAL's filters before generating the DSMs.

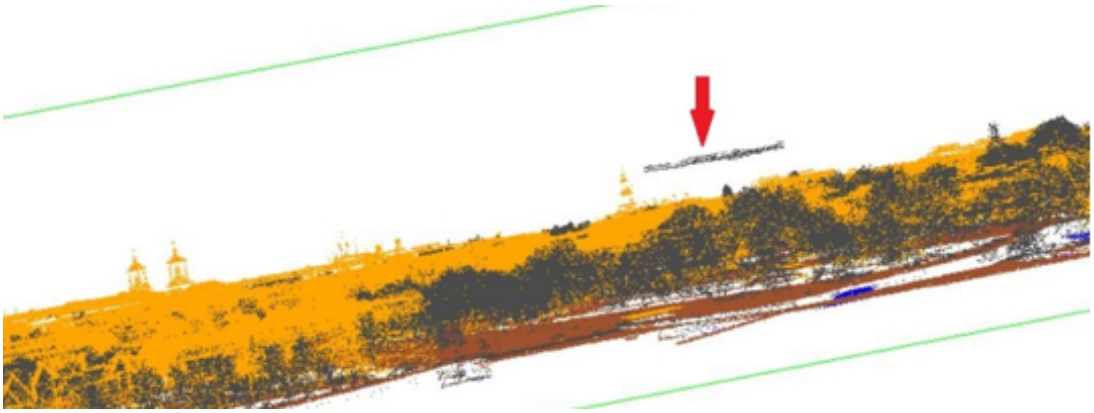


Figure 2.21: Points in the raw point cloud that cause artefacts. Most likely a crane was present.

on the GitHub repository)

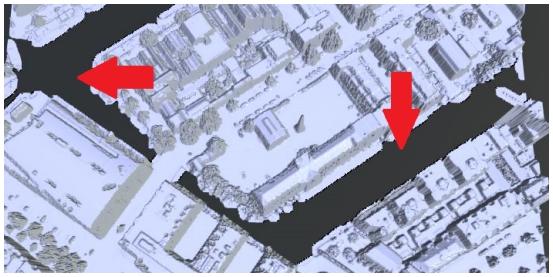
- startin-Laplace
- startin-TINlinear
- CGAL-NN

Overall, the above four methods provided us with good quality DSMs. However, all their outputs came with some artefacts around the areas where buildings are located directly adjacent to water bodies (canals or rivers for example) and where trees extend over water bodies, as we show in Figures 2.22 and 2.23. Out Laplace and NNI interpolation methods created slopes connecting buildings and tree canopies to the opposite sides of the adjacent rivers and canals, while TIN-linear interpolation produced odd, triangle-shaped artefacts. Meanwhile, the quadrant-based IDW method exhibited the disadvantage that with the parametrisations we found to be stable, it often generated undesirable amounts and sizes of holes, especially where water bodies are located (which constitute gaps in the Lidar data even when not only ground points are kept, owing to their optical properties). This can be seen in Figure 2.22a. However, these issues can mostly be resolved in post-processing, as will be explained in the next section - hence, none of these methods were excluded on this basis. Since all four methods show equally good visual quality otherwise, making a final choice required statistical analysis.

DTM & DSM Post-processing

While this section will not yet go into any depth regarding the exact algorithms implemented, we will now present some basic details about our post-processing steps as these were used comprehensively when testing and evaluating the interpolation methods, and choosing the final candidate. Two types of post-processing steps were implemented in our testing environment with the intention of using them in our production environment (which the scaling team developed). We termed the first method ”basic flattening”, as this algorithm burns the areas of polygons into the output raster (via rasterisation) at a constant elevation. The elevation value of a polygon is the median of the interpolated values at its vertices. We named the second method ”pixel patching”, because it is a simple method that uses a moving kernel to fill in small isolated groups of missing pixels in the output. The values are assigned based on the median of the 8 neighbors of the pixel in the moving kernel. The third method (hydro-flattening) is a proof-of-concept implementation that we did not intent to pass on to the scaling team; it server demonstration purposes only. The full set of details about these algorithms and their implementations are explained in Section 2.3.

As mentioned above, the basic flattening algorithm can flatten polygons to an elevation that blends into the surrounding environment. This resulted in better-looking water bodies in the output overall. Furthermore, it also made most of the DSM artefacts disappear. Therefore, even though the DTM and DSM results of our continuous (TIN-based) methods do not contain holes, this post-processing step,



(a) Holes in the outputs of the quadrant-based IDW interpolation.

(b) Artefacts caused by the continuous interpolation methods. The top of the buildings and trees are connected to the other side of the canals.

Figure 2.22: Artefacts caused by the DSM interpolation methods

which was originally meant to fill holes, also proved useful for these methods. We do note however, that interpolating at the vertex location provides inaccurate results when there are other objects (e.g. trees) overlapping with the boundaries, which may corrupt the computation of the overall elevation of the polygon. Furthermore, there are two other limitations of the method: firstly, large polygons (e.g. long rivers) are not generally associated with a constant elevation. Secondly, it is not, in its current form, suitable for building polygon flattening, because due to the imperfect lining up of the polygons with the Lidar representation of buildings, the heights would inveitably be interpolated from a mixture of building points and other (e.g. ground) points. A suitable method would be to use Lidar points within the extents of the polygons instead, but this would be extremely complex computationally and therefore buildings are better interpolated directly without the use of post-processing. Furthermore, the pixel patching algorithm was also used for all methods as it has a low computational complexity, but a very positive visual impact on the results.

Statistical Comparison

Owing to their success, all statistical analysis (for both DTMs and DSMs) used outputs with post-processing (basic-flattening and pixel patching). This following section gives a detailed account of the procedure and results of the statistical analysis.

To choose the best interpolation methods for the generation of DTMs and DSMs, we statistically compared the results of all interpolation methods with the initial input point cloud we used to generate them. More specifically, we calculated the vertical differences between point cloud point elevations and the values of the raster pixel into which they fall based on their lateral coordinates. The result of these operations are rasters describing local interpolation accuracy (which we included in the Appendix) and statistical data, which we show here in tables.

From the resulting rasters, we discovered that the largest vertical differences can be found in the DSM results, where trees are located close to building boundaries. This was expected because there are a large quantities of points on the façades of buildings, and trees are found with the same lateral coordinates at these locations, but with various heights (Z coordinates). This means that this disagreement does not necessarily constitute a problem despite the large errors.

Metrics used to measure accuracy like Mean Average Error (MAE) and Root Mean Square Error (RMSE) gave us further constraints regarding the correctness of the results. The only difference between them is that the RMSE value is larger than the MAE value when all the prediction error comes from a single or a few test samples, Janet [2016]. Tables 2.5 and 2.6 show the MAE and RMSE values of all results of all interpolation methods, except the PDAL-IDW method of the framework, where post-processing was not implemented as we excluded the method from further consideration before implementing post-processing.

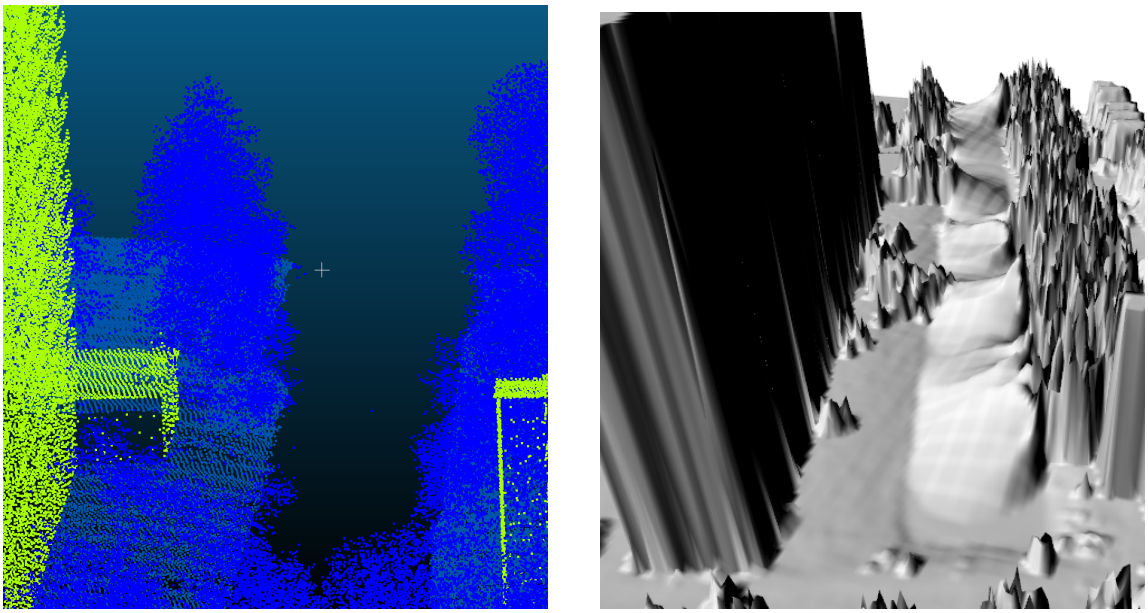


Figure 2.23: Artefacts caused by continuous interpolation methods, tree canopies are connected to the other side of the canals.

The MAE values were calculated using the following equation:

$$MAE = \frac{\sum_{i=1}^{n} |yi - xi|}{n} \tag{2.2}$$

where n is the number of pixels in the raster image and yi - xi is vertical difference between the points of the raw point cloud and the value of their corresponding pixel.

The RMSE values were calculated using the following equation:

$$RMSE = \sqrt{\frac{1}{n} \frac{\sum_{i=1}^{n} |yi - xi|}{n}} \tag{2.3}$$

where n is the number of pixels in the raster and fi - oi is vertical difference between the points and the value of their corresponding pixel.

Looking at the values of the RMSE table we can see that the Laplace interpolation method has the lowest RMSE values when generating DTMs while the IDWquad method performs best when generating DSMs. The TIN-linear interpolation gave the largest values in most cases. Looking at the values of the MAE table we can see that the NNI and IDWquad interpolation methods have the lowest MAE values when generating DTMs and IDWquad interpolation method also has the lowest MAE values when generating DSMs. The TIN-linear interpolation once again gave the largest values in most cases. Comparing the MAE and RMSE values, the RMSE values are much higher than the MAE values, especially where tall buildings are present (in Delft, for instance). This was expected because as we noted before, the largest vertical differences (errors) come from a limited number of pixels that are close to trees and that are around the boundaries, especially corners, of buildings. The RMSE and MAE values prove to us that our results are satisfactory for our project's aims. The surfaces we create fit the raw point cloud reasonably well.

MAE											
DTM						DSM					
IDW	Laplace	NN	TIN	IDWquad	CDT	IDW	Laplace	NN	TIN	IDWquad	CDT
Amsterdam											
0.074	0.079	0.061	0.064	0.075	0.082	-	6.107	6.364	6.996	4.908	6.994
Delft											
0.049	0.042	0.042	0.045	0.04	0.045	-	10.786	11.528	13.195	7.177	13.192
Groningen											
0.069	0.056	0.056	0.058	0.052	0.057	-	5.271	5.65	6.497	3.82	6.497
National Park Veluwezoom											
0.166	0.117	0.107	0.109	0.114	0.12	-	3.919	4.226	4.977	2.988	4.977
National Park De Biesbosch											
0.192	0.123	0.1	0.102	0.119	0.126	-	6.296	6.743	7.757	4.626	7.76

Table 2.5: MAE values of interpolation results

RMSE											
DTM						DSM					
IDW	Laplace	NN	TIN	IDWquad	CDT	IDW	Laplace	NN	TIN	IDWquad	CDT
Amsterdam											
0.207	0.186	0.196	0.213	0.315	0.328	-	24.612	25.422	27.297	18.047	27.274
Delft											
0.095	0.072	0.073	0.077	0.084	0.084	-	50.29	52.286	56.138	22.066	56.303
Groningen											
0.135	0.107	0.107	0.109	0.102	0.11	-	14.776	15.833	18.19	10.545	18.188
National Park Veluwezoom											
0.744	0.152	0.153	0.155	0.51	0.424	-	11.531	12.441	14.584	8.779	14.584
National Park De Biesbosch											
0.36	0.156	0.157	0.159	0.529	0.532	-	17.329	18.446	20.97	12.657	20.96

Table 2.6: RMSE values of the interpolation results.

2.2.2 Interpolation Decisions

For interpolation, we implemented six methods with vastly different characteristics. Most methods produced reasonable results that are evidently better than the current DTM/DSM both in terms of visual and statistical analysis. The main differences between the methods were related to the continuity of the output terrain and surface models. While continuous methods such as NNI and Laplace generate smooth terrains and surfaces, which is desirable for DTMs, they produced artefacts when tested with DSM generation. On the other hand, the IDWquad method and its variations create holes where water bodies are located, but which can easily be filled using our post-processing steps. Below, we explain how the final decision regarding DTM and DSM generation was made.

DTM

The discussion of visual analysis concluded that using one of the continuous methods would be more suitable for DTM purposes. Laplace, in particular, is very efficient computationally, and using continuous methods allow continuous surfaces to be produced and then enhanced via post-processing, which we though is a better approach than filling holes, which could probably not be executed with a 100% effectiveness. As the TIN-linear method produces undesired angular shapes where large holes are present, which is an issue specifically for buildings (as they are not flattened in the post-processing stage), our attention was drawn to Laplace and NNI that produce smoother results. Furthermore, the statistical analysis proved that these two methods deliver the best fits to the point cloud results on all our testing tiles. MAE showed that NNI produces slightly better results in three of the data samples, but the difference was negligible. However, Laplace was ranked as the top candidate as it produces slightly better results using all our samples according to the RMSE values, as shown in Tables 2.5 and 2.6. To further tip the balance towards Laplace, our timing tests have shown that it is by far the best performer in terms of computational complexity. Hence, the final decision was to use Laplace interpolation for DTM generation and the implementation was passed on to the scaling team to proceed with integrating it into their server-based environment.

DSM

Making a decision about the most appropriate method to use for DSM generation was difficult. Continuous methods caused artefacts around water bodies whereas IDWquad generated holes at the locations of water bodies, as previously explained. As post-processing solved all these issues, the decision was down to further visual and statistical analysis. The visual results narrowed down the options to TIN-linear, Laplace, NNI and IDWquad with equal suitability. On the other hand, the statistical results concluded that IDWquad delivers the best-fit result in both RMSE and MAE tests, as can be seen in Tables 2.5 and 2.6. Therefore, our final decision was to use quadrant-based IDW interpolation with the final parametrisation determined by the testing team for DSM generation, and so the IDWquad implementation was also passed on to the scaling team to proceed with integrating it into their server-based environment.

2.3 Hole Filling / Hydro-Flattening

In terms of filling holes, our initial concept was that we would implement a Constrained Delaunay Triangulation (CDT) via CGAL by inserting polygon boundaries as constraints into the triangulation of the pre-processed Lidar points. While the implementation was successful and the constraints indeed formed closed loops (defining holes) inside the CDT, the holes were still triangulated by CGAL. This was in part the result of the input polygons (such as lakes and canals) not coinciding perfectly with the Lidar extents of the holes – some Lidar points inevitably fell within the polygons, causing them to be triangulated. The constraint-holes would have represented an elegant solution to perform hole-filling, because it would have allowed the holes to be present in the triangulation directly (only possible via

CDT). Hole filling could then have taken place within the triangulation itself. However, removing the Lidar points that were within these holes would have been too complex computationally, hence we opted for a different approach.

An important reminder here is that all triangulation-based primary interpolation methods are continuous within the convex hull of the points making up the TIN. CDT would have introduced real holes into the TIN, but as noted above, this is not the solution used in the production configuration of our code. In the case of such continuous methods, the term "hole filling" is not descriptive of the real operation at work, hence we internally refer to this stage as "basic flattening", because it is similar to hydro-flattening, but it is simpler in the sense that the polygons are simply flattened to a constant elevation, rather than modelled as surfaces with constant slope directions. Instead of using the CDT workflow, the production code instead takes the startin-TIN constructed during the primary interpolation phase and interpolates at the input polygon vertices in it. The median of the obtained elevation values is treated as a constant elevation value for the whole extent of the polygon, which is then burned into the output raster as part of an efficient rasterio operation. For the final DSM choice (IDWquad), such a TIN is not constructed during primary interpolation, hence a surrogate TIN is built from scratch at the beginning of the post-processing stage, from the pixel centres af the semi-final output raster to be edited by the post-processing code.

It is worth mentioning how the vector geometries are imported. Since the source data sets (BBG and BAG) cover the whole extent of The Netherlands but each subtile processing operation concerns a small bounding box, we needed to implement a vector tiling algorithm to select those polygons which are within the bounding box of a given tile, and crop those which intersect the bounding box boundaries. This allowed us to process much larger vector data sets, which much better performance. However, the roughly 12 Gb size of the BAG data set still exceeded the code's capabilities, hence an alternative approach was selected. The tiling of this large data set is done externally, as the code obtains the relevant polygons via WFS requests from the 3D BAG service provided on the servers of TUDelft's 3D geoinformation research group. This solution allows both the post-processing stage, and the CGAL-based CDT algorithm (described earlier in this report) to benefit from the building polygons contained in BAG, optionally. Although BAG footprints are not used in any part of our server-based production environment, we perfected this code as it is suitable for any local or WFS-based vector data sets (not just BBG and BAG, which we used), which could be useful for future work.

When the pre-existing startin-TIN is re-used from primary interpolation (not possible if primary interpolation not involving startin was used), this post-processing approach produces visually appealing results with a computational complexity that is at least a factor of two less than that of the primary interpolation. When IDWquad primary interpolation is used, this constitutes a performance bottleneck, which was partially resolved in the server-based production version of the code. As the visualisations in the previous sections had shown, the smooth, flat surfaces resulting from the flattening of the input polygons blend into the surrounding parts of the rasters.

2.3.1 Pixel patching

The final production configuration of our code uses the above "basic flattening" algorithm in conjunction with the PDAL-based pre-processing step and the startin-based Laplace interpolation (for DTM generation) and IDWquad (for DSM generation) as described in earlier sections. The Laplace-based configuration produces no holes or gaps, but it can produce blank (no-data) areas around the tile borders wherever pixels fall outside the convex hull of the startin-TIN. These areas are the only areas, where the terms "hole filling" is still appropriate for the post-processing step referred to as "basic flattening" otherwise. In these areas, the algorithm is indeed giving pixels values that were originally no-data pixels, mainly representing large water surfaces where Lidar imaging could not obtain values, or where PDAL-based pre-processing removed points.

In these areas, polygons are burned into an environment of empty and non-empty pixels. This may leave no-data pixels, or groups of no-data pixels between the burned-in polygon and the rest if the interpolated tile, or other burned-in polygons. These small gaps are subsequently filled in a separate step we call "pixel patching". In this step, the program scans the raster pixels once, patching in these

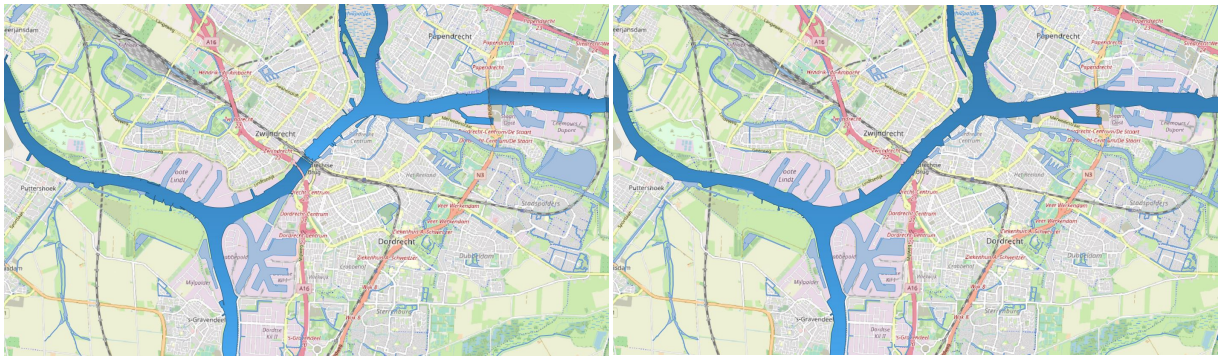
empty cells. Only those empty cells are interpolated in such a way, which have at lest a pre-defined number of neighbours. The interpolation itself uses a simple median kernel, that is, the median value of the 8 neighbours' values is the interpolant. The method was inspired by the idea behind the fallback kernel method in PDAL's IDW implementation. Furthermore, IDWquad does often generate holes, as we have explained its generalised parametrisation does not guarantee continuity. This means that in the case of our final DSM production configuration, both basic flattening and pixel patching perform hole filling where they work in areas containing no-data pixels.

2.3.2 Hydro-flattening

The polygons used in the production configuration of our code for the above basic hole filling operation are all the water polygons in BBG that are not flagged as rivers. The flattening of rivers follows the hydro-flattening concept as outlined in our initial proposal, meaning that they are flattened separately, after the above hole filling phase.

The first step was to skeletonize the river polygons of the BBG. First, specific rivers were selected based on their names, stored as attributes in the BBG data set. But as can be seen in Figure Figure 2.24a rivers can consist of multiple small polygons in this data set, so to get a skeleton over the whole river body we needed to combine these parts into a single river polygon. Having not been able to find a solution in Python, we resorted to applying QGIS buffer and dissolve operations instead. By first buffering the river polygons by a small amount, we guaranteed that they overlap. This made it possible to dissolve them, and the the buffer operation could then be reversed. The results are shown in Figure (Figure 2.24). The risk associated with this method is that the water bodies may lose some details and/or small holes, but in our particular case, this did not represent a practical obstacle. Skeletonization could then take place.

For this operation the "skeletonize" function from Scikit Geometry Python module was used. Due to the size and complexity of the water bodies, the skeletonization process was rather slow, hence we performed polygon boundary line simplification to speed up the process. The Scikit simplification function unfortunately also ran into memory issues due to the size and complexity of the BBG polygons, hence this operation was also executed in QGIS.



(a) Rivers consisting of seperate polygons (b) Dissolved river polygons

Figure 2.24: Combining river polygons

While the above script generates a network of river skeletons that are generally well-behaved, this pruning approach cannot eliminate all branches, small branches are still present in the output. The skeletons will be used in the next step as a linear profile of segments of the river network, hence small skeleton branches are unacceptable. We solved this issue via QGIS's native network analysis tools. Start and end points were selected for segments of the river network, and the shortest paths in the network between these points were computed. This yielded a new representation of the river skeletons that is entirely free of unwanted branches.

This series of semi-automatic steps is followed by the main part of the hydro-flattening algorithm. The following steps are taken:

1. The river polygons and skeletons are imported the same way as other water polygons in the hole filling step above.
2. Perpendicular lines are cast on the skeleton at its vertices. These are of an arbitrary length, long enough to certainly intersect the shorelines.
3. Cross-sections are constructed from these long skeleton-perpendicular lines. This is done by identifying the closest intersection with the river polygons of these lines in both directions from the spine.
4. Each cross-section is associated with an elevation. The elevation is obtained by interpolating at the two relevant shore points, and the relevant spine vertex. Like in the "basic flattening" stage, the interpolation is done in the TIN inherited from primary interpolation.
5. The cross-section elevations now form an elevation profile along the length of the skeleton. These are refined to form a series that decreases monotonically (with constant elevations allowed). This is done by first removing all values that increase after the first elevation in the series, and then interpolating linearly where they were removed.
6. The water polygons are rasterised to form a mask. The mask is used to identify those pixels, which are covered by the polygons. These are each interpolated (or re-interpolated) in the next step to become part of the hydro-flattened surface.
7. The actual interpolation takes place. The previous and next cross-sections relevant to each water pixel (pixel covered by a river water body) are identified, and the value of the pixel is set by inverse distance weighting, using the elevations of the previous and next cross-sections, and the two distances from them.

The last step is crucial. It effectively means that we take the river pixel, find which previous and next cross section it falls between, and then associate an elevation with it that is somewhere between the elevation of these two cross sections. Theoretically, this guarantees that the elevation will always decrease downstream under normal circumstances. For this, we need the cross-sections to be associated with monotously decreasing elevations in the downstream direction, which is guaranteed by the third step above. When identifying the previous and next cross-section relevant to a water pixel, the program not only searches for the two closest cross-sections, but also ensures that the interpolation point (the river pixel) indeed geometrically falls between them. It does this by "casting rays" to the closest point on each cross-section and checking whether the ray intersects any other cross-sections along the length of the way. If it does, the cross-section is excluded from further consideration because the river pixel is separated from it by another cross-section and therefore it can be neither the previous, not the next cross-section that the program is looking for. Part of this procedure is illustrated in Figure 2.25.

Computing the location of the nearest point on each cross-section that is tested, is an operation with a high computational complexity, which is one aspect of our implementation that we would recommend for further work. One possible alternative worth exploring would be to keep the cross-sections long (extending beyond the first intersections with the river polygons), and casting rays to some diagnostic points along its length, instead of the nearest point specifically (which is expensive to compute).

In theory, this should work perfectly, and it does work quite well in practice where the rivers are relatively straight, and the spine vertices are relatively sparse. However, dense spine vertices, especially in river bends, may result in intersecting cross-sections, which will trick the interpolation mechanism into drawing a small area with locally reversed flow direction. These are generally cone-shaped artefacts. In multi-channel settings (consider e.g. the islands in Rotterdam), the spine of the shorter channel was split off so that it is interpolated almost like a separate river. This ensures that the algorithm does not run into logical problems at these locations, but further work to ensure cross-channel continuity would be required. Furthermore, the intersections at river-channels may cause many artefacts to appear in

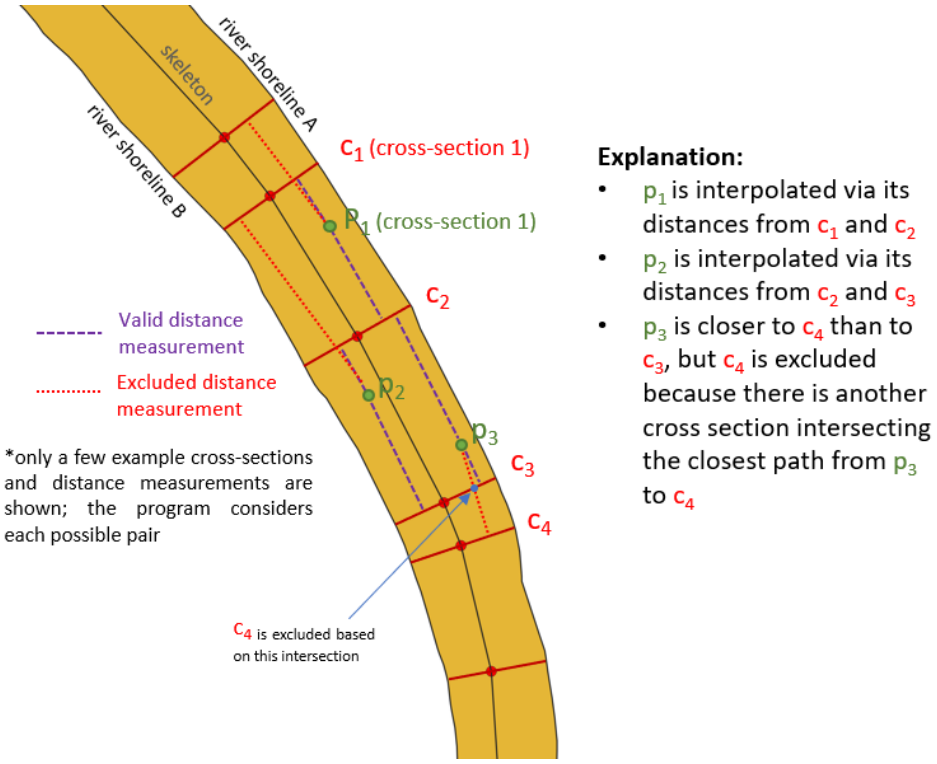


Figure 2.25: Illustration of hydro-flattening algorithm.

most cases. This is the result of a great number of potentially intersecting cross-sections being generated in these areas, which do not have consistent orientations or elevation values. This may subdivide the surface of the river locally, into parts that have conflicting elevations, and slope directions.

We suggest further research into fixing these issues. We think that it I would be better to generate the skeletons in such a way that they allow the algorithm to perform better, than to make significant changes to the structure of the algorithm itself. In particular, the placement of skeleton vertices in river bends should be done in a way that prevents the generation of intersecting cross-sections. It requires further testing and research to decide whether the issues related to the areas where the rivers fork into channels could be resolved in the same way.

The algorithm is reasonably robust in the sense that it can handle odd river shapes (such as small offshoots, dock channels, hydro-power plants, etc.) without any issues, assuming that the continuity and placement of the skeleton complies with some pre-conditions, such as that it runs within the river channel, and that it does not itself branch at any place. This robustness is the result of the algorithm's flexible approach to finding the previous and the next cross-sections. However, there are some circumstances under which this may fail quite badly. For instance, long dock channels extending into the convex part of large, sharp river bends may contain water pixels that are closer to cross-sections other than the ones which they belong to (based on where they are connected to the river) without intersecting other cross-sections, so that the algorithm could exclude it from further consideration, as it does for well-behaved river shapes.

2.4 Scaling

Scaling performs two main tasks within the pipeline. One of these tasks is related to the managing of the bulk of data; splitting large tiles into smaller ones, bringing them back together, and managing the task queue. The other task is based around reducing merging artefacts when the smaller tiles are merged back into a large tile, as well as artefacts between neighboring tiles.

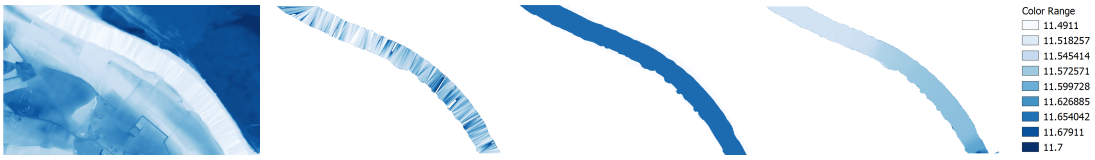


Figure 2.26: As this arrangement of figures shows, the absence of post-processing, basic flattening, and hydro-flattening produce vastly different river surfaces. Without any post-processing, the river surface is generally comprised of sliver triangles, which are what the triangulation-based primary interpolation leaves where it encounters holes that are densely surrounded by valid points. Basic flattening on the other hand burns the entire river polygon into the raster with a constant elevation. For small water body polygons, this is a reasonable approach because their elevation does not normally change vary much, but for long rivers, the same assumption does not hold. Hydro-flattening constructs a slightly sloping surface based on the on one additional supplementary geometry, which is the skeleton of the river. The slope is consistent along the length of the river, which is more realistic in terms of visualisation and any possible further processing it may be needed for that involves rivers.

The main aim for scaling is to be able to interpolate the tiles for Zuid-Holland, with a possible expansion to the Netherlands. The aim of Zuid-Holland sounded feasible at first, but during processing it was discovered that some algorithms took longer to run than initially estimated. This resulted in only a part of Zuid-Holland having been completed (ca. 30/115 tiles). Considering that the goal of this project is to improve the AHN3 gridded DTM/DSM, the amount of tiles that is completed is not the most important aspect of this process.

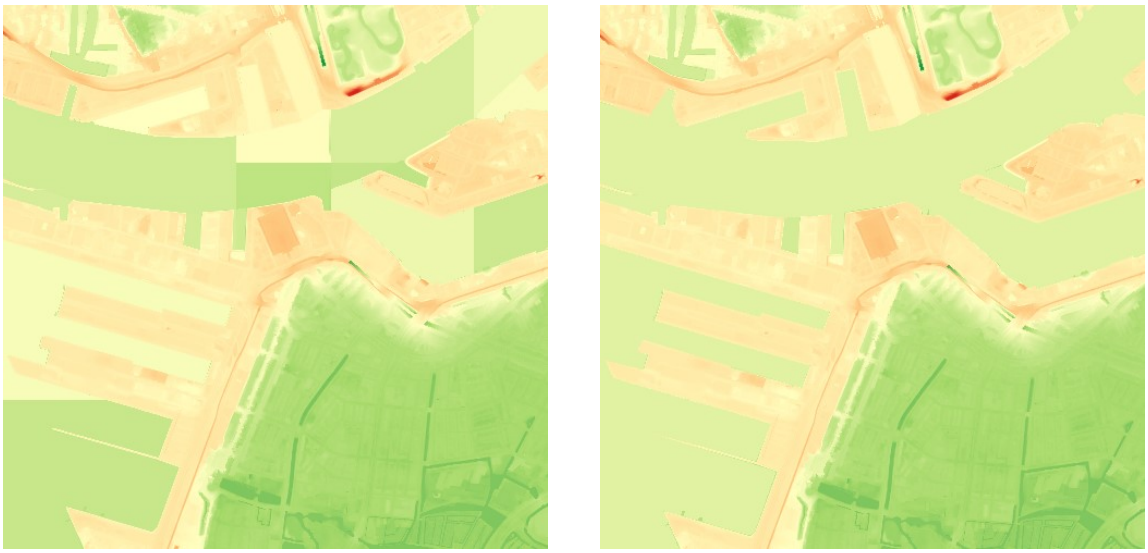
2.4.1 Managing Data

As mentioned, one of the challenges of this project is the sheer amount of data that is present in the Lidar point clouds. Considering that each tile is 5x6.25km many millions of Lidar points are contained in each file, giving it an uncompressed size upwards of 14GB. To manage this amount of data, various different choices are made to create manageable chunks of data.

The first choice made is to split a single AHN3 tile into multiple subtiles, thus reducing the 14GB size to 20 tiles (4x5) of 800MB. Using 20 subtiles in a 4x5 grid is a choice based on reducing the subtiles to a little less than 1GB, as this would be able to be processed in a reasonable amount of time. An added bonus is that it makes each subtile square, which has real value besides being aesthetically pleasing. This step also includes creating a buffer around each subtile, which ensures that when interpolating is it possible to accurately interpolate the values near the boundaries of the subtile. The buffer chosen for the final runs is 25m which leaves more than enough Lidar points available outside the border of the final raster.

Part of the splitting of the main tile into subtiles is to accurately be able to bring these back together once the interpolation has completed and the raster output is ready. This is done by initially storing the output of each successful interpolation to disk within a tile's own processing folder. Once it has been detected that all interpolations for a tile have succeeded the rasters can be merged. After this merging has completed a few artefacts will remain related to water bodies, these are subsequently removed by a homogenization step. These artefacts are caused by the interpolation of the water to differ per subtile, as can be seen in Figure 2.27a. The homogenization step uses the completed (merged) raster and the water body polygons to give all connected water bodies a value equal to the mean of the sections. This will ensure that the various water bodies from the different subtiles within a single tile are homogenized, which can be seen in Figure 2.27b However, there is currently no homogenization implemented within this section that homogenizes the water bodies bordering between the full tiles.

Another aspect to managing this data is to have a framework which is flexible and able to issue and complete tasks as they are created depending on various triggers. This is done by using a main entry point for the application that reads the LAZ files that are contained in a folder and subsequently creates



(a) Water body patchwork pre-homogenization (b) Water body coherence post-homogenization

Figure 2.27: Comparison of unhomogenized vs. homogenized waterbodies

the required tasks for the files found. The entry point initially creates a few tasks to split the tiles into subtiles and places these tasks in a queue. Subsequently it will create processes that will pick up a task from this queue, process it, report the result, and repeat this indefinitely. Possible tasks have been subdivided into a few categories, namely: splitting of AHN3 tile, interpolation, merging of completed rasters, and downsampling of merged raster. By using this configuration as many processing units can be created as the machine can handle, with the limitation being the amount of RAM available. During processing the RAM usage varies depending on the task. For a machine running Ubuntu 18.04 LTS a maximum usage of 20GB per process was recorded, whereas for another machine running Windows 10 Pro 64-bit a maximum usage of 10GB was recorded. Where this difference lies is currently not known, but it is essential when determining how many processes are started to ensure RAM is available when necessary. An overview of how this queuing structure is composed can be seen in Figure 2.28.

2.4.2 Reducing Artefacts

As mentioned, artefacts are introduced by the splitting of the tiles, and in Figure 2.27 it has been highlighted how some of these issues are tackled. These artefacts are related to a post-processing step introduced to flatten the water bodies that are present in the tile. A related type of artefact that may occur is related to the post-processing step in which the building polygons are flattened, where similar issues occur as with the water flattening.

For the building polygons the solution is based on the buffer that is used around each subtile. Using this buffer it is possible to have most of a building polygon within the interpolated area, thus allowing for the median value of this building to be the same for different tiles. This is because the tiles on both sides of the building polygon have the complete overview of the values within that polygon, and thus are able to provide a near exact same median value for that polygon. Therefore, if the polygon is cut somewhere due to the tiling, the values on both sides of the cut are close to eachother.

Another way that artefacting was reduced is by trying to keep subtiles as large as possible. Intially a 3x3 grid was attempted for interpolation, thus reducing the amount of joins between tiles from 31 to 12, compared to a 4x5 grid. However, this was computationally too intensive to be feasible to execute. The main reason for this is that it would reduce the amount of threads that could be run in parallel from five threads to two, thus having a large impact on how much of Zuid-Holland could be processed.

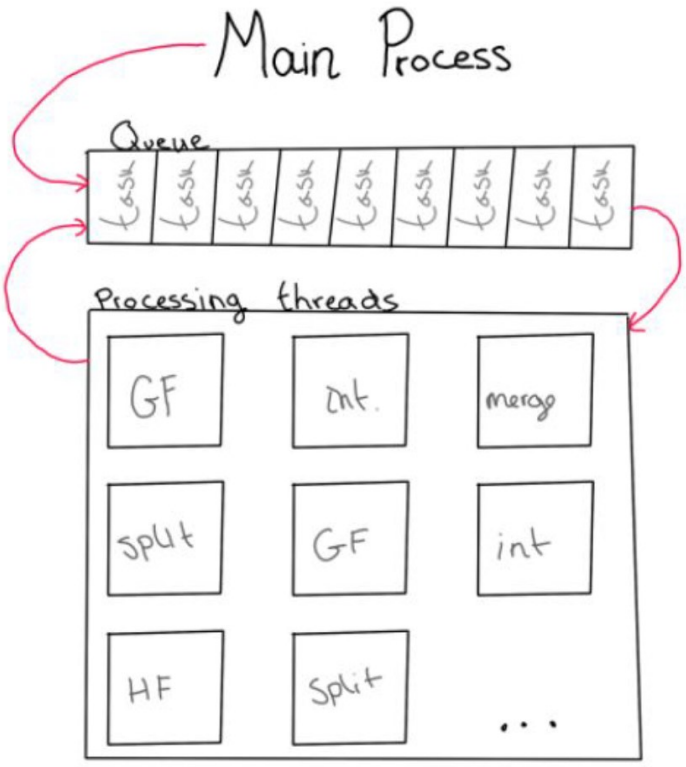


Figure 2.28: Figure showing how the threads use a queue to put and get tasks

3 Results and Recommendations

As shortly mentioned in Section 1.1 the result of this project is a successfully improved gridded DTM and DSM. The main goals of removing missing pixels and providing values for all raster cells have been achieved for both the DTM as well as the DSM. Furthermore, proof of concepts have been created to show how hydroflattening works and a simplified version of such has been applied on the full-scale project. In the following sections the results will be discussed in more depth, followed by recommendations on what could be improved in the current workflow, and finally what future work we envision could be done using this project as a base.

3.1 Results

The main goal of the project to improve the gridded DTM/DSM currently provided has been achieved. As mentioned there are no more missing pixels and no-data values have been removed in various ways. Especially in the DTM this difference can be noticed, where the result has gone from a hole-ridden Swiss cheese to being as smooth as the Dutch countryside, as can be seen in Figure 3.1. For the DSM the result is slightly more subtle because the building footprints have not been removed, thus creating less holes that require post-processing to fill in. However, water is still not given a value in the DSM and various missing pixels due to interpolation choices have been resolved in the updated DSM version, as can be seen in Figure 3.2. Both figures depict the raster with 0.5m cell size, but 5m versions have also been created as these are also available for download on the PDOK website.

However, despite these nice results there are also a few areas in which the results are currently less optimal than the current solution. This occurs in locations near the coast of the Netherlands or in places where there are large waterbodies. Here, the program attempts to flatten the water body using the polygons it has, but it can only flatten the tiles that have actually been created. Empty tiles are not interpolated, thus it is not possible to flatten them using this method, resulting in the effect that can be seen in Figure 3.3.

Furthermore, the results for the scalability of this project are reasonable as well. The framework is able to run for a while, but shows some memory leaking during this process and is therefore often forced to restart around the 24 hour mark. It was also not possible to run the entirety of Zuid-Holland, let alone the Netherlands, in less than a week, which was initially expected to be possible. This is mainly due to the algorithm used for interpolating the DSM that took longer than expected. While it is possible to speed up this algorithm by implementing it in a language such as C++ or Rust, being able to run a large amount of data in a short amount of time is not a criteria for the success of this project.

An indication of how long it takes to process a single subtile (1/20th of an AHN3 tile) has been included in Figure 3.4. The results in the figure have been based on a five different sample from various subtiles. Varying complexities of the content of the tile greatly affect the speed at which the algorithms are able to process the tile. Furthermore, an overview of how many, and which, tiles have been completed can be seen in Figure 3.5. Around 30 tiles of the 115 for Zuid-Holland have been completed at the time of writing. More tiles are still being processed for handover to Het Waterschapshuis to show that the results show promising results over many different types of tiles and environments.

As mentioned previously, the code can be downloaded from the following GitHub repository: `https://github.com/tudelft3d/geo1101.2020.ahn3`. All interpolation methods are found in another GitHub repository: `https://github.com/khalhoz/geo1101-ahn3-GF-and-Interpolation`. Furthermore, the latest collection of completed results can be downloaded from: `https://stack.dejong.cc/s/oIYdMo7uB1KI5jL`.

(a) ”Swiss cheese” DTM with many holes and no-data values for buildings and water

(b) ”Countryside” DTM with no holes or no-data values are any location

Figure 3.1: Comparison of before and after results of DTM for tile 37HN1 (Rotterdam)

(a) DSM as currently available for download

(b) DSM as result of this project

Figure 3.2: Comparison of before and after results of DSM for tile 37HN1 (Rotterdam)

Figure 3.3: Image showing how this DTM along the coast is only partially water flattened

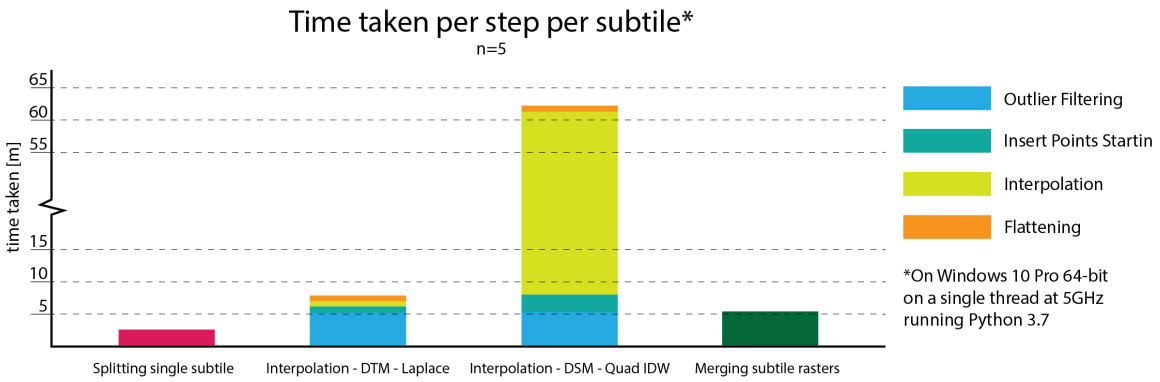


Figure 3.4: Figure showing time taken for various processing steps for a single subtile

3.2 Recommendations

During the course of the project different situations have been encountered in which critical decisions were made which affected the results shown here. Some of these decisions resulted in improved results, while other decisions may have created new issues. The latter is what will be discussed here per section, where each section will elaborate on what recommendations there are for improving these results.

3.2.1 Ground Filtering/Pre-processing

As far as ground filtering and pre-processing is concerned, we mentioned in one of the previous sections that for the DTM we trusted the AHN classification. As input for our interpolation algorithms we used

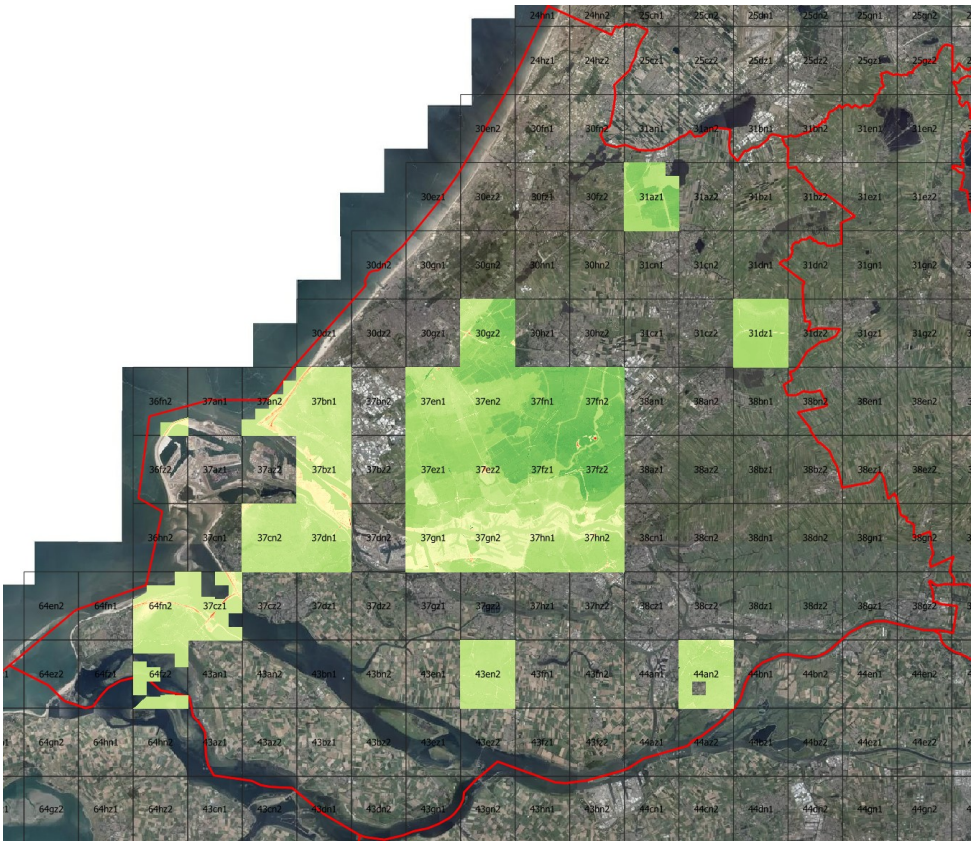


Figure 3.5: Figure showing an overview of all the tiles that have been completed at the time of writing

the points classified as ground points by the AHN after first applying an outlier and low noise filter on the raw point cloud. This happens to be our final decision on how to approach ground filtering because we did not manage to find a better ground filtering algorithm. Moving forward, we recommend testing and experimenting with more ground filtering algorithms. During this project we only tested four. It is also possible that mistakes were made with the parameters or that we did not test as much different parameters combinations as we should. However, even though we could find some better combinations of parameters or test more ground filtering algorithm, that would lead to better results, we would still not be able to overcome some problems like boats. Removing some classes of points from the whole process is a bad approach, thus preprocessing the raw point cloud is the only solution. For example, we could make use of the river polygons of the BBG to clip the raw point cloud and remove all points in the rivers and canals (water, bridges and boats/ships points). This would allow us to make use of other ground filtering that suffered from these points and identify more ground points without having flaws.

For DSM, as a pre-processing step, we removed water points from the raw point cloud before continuing with interpolation. However, as noted in the interpolation section, the results have some flaws caused by points that could not be identified as outliers and be removed by the filters we applied on the raw point cloud. These points belonged to a temporary object, a crane, that happen to be there during data collection and caused a spike. After examining the raw point cloud, we also noticed that the body of the crane was missing and only the top part of it was present. A special algorithm must be written to remove points like that since common outlier filters cannot distinguish them.

3.2.2 Interpolation

For interpolation, the focus at the beginning of the project was mainly on getting as many algorithms operational as we can, which was good in terms of having more options. However, less attention was

paid to studying and analysing the quality and the nature of the input data (AHN point cloud). Moreover, having many methods/possibilities caused a dispersion and loss of time. So instead of focusing on two or three suitable and realistic solutions, much time was spent on testing the algorithms and getting them to work. In future similar projects, it would deliver better results and be more efficient if an algorithm can be developed according to the input data quality/nature.

One of the flaws of our interpolation methods is that artefacts are created around rivers/canals when producing the DSM. This happens because of two reasons; 1- rivers/canals create holes in the point cloud since they contain no points in most cases and 2- trees extend over water bodies. An improvement could be made to tackle these two artefacts and reduce them. For instance, an algorithm could be written that fills these holes with water points before interpolation takes place; or an algorithm that isolates tree points from the data and interpolates them separately.

The basic-flattening algorithm we developed is also applied on the interpolation results for both DSM and DTM as post processing. This algorithm estimates the elevation of the water body using a Laplace interpolant at the locations of each polygon vertices. This method is not entirely accurate for a couple of reasons. First, the polygons data set used does not align perfectly with AHN point cloud. This means that some water-body-polygons shift a bit off the AHN waterpoints. As a result, some of the points that belong to the bank of the water body will be counted in, which is really problematic at canals where the elevation of the water is much lower than the elevation of the bank. Moreover, the points of trees extended over waterbodies can also be counted in, and these have also much higher elevation value than the water-body itself. Second, Laplace interpolant was not chosen as it causes some artefacts around water bodies, so it is not optimal to base an algorithm on the accuracy of it. For future work, we recommended finding a different approach to estimate the elevation value of waterbodies.

3.2.3 Hole Filling

With hole filling one of the main recommendations is to find a method in which the polygons can be obtained and matched more accurately. Currently it is assumed that the polygons align well enough with the lidar data to prevent real issues from occurring, but this may not be the case for all the polygon sets, or in countries other than the Netherlands. If the case occurs in which a polygon does not align, then data will be removed in the wrong locations. This may currently occur due to temporal differences, or because of general misalignment.

Furthermore, the polygons uses have been downloaded from external locations and processed by hand. This introduces more simplification as the polygons have been adjusted to be connected with large tolerances as well as removing small branches from them. If, mainly water polygons, are able to be created in an automated fashion using the data present, this would improve the results. This is because when using a medial axis transform, or similar technique, the polygons created will always match the lidar data that is present, hereby reducing the risk of creating new artefacts due to mismatched data.

3.2.4 Scaling

For scaling the main focus during the project time has been to create a framework within which the different algorithms can be run, as well as managing the files that act as in- and output. However, the portion of scaling related to removal of artefacts received less attention, thus having been less optimized. This can be seen in how scaling introduces small artefacts between the subtiles as well as how the water flattening that has been performed isn't continuous between full-sized tile borders. In the existing DSM and DTM tiles this wasn't an issue as water was not interpolated, thus not creating any new artefacts that require resolving. One of the recommendations is therefore that more attention could be spent on ensuring that the data from one tile to another is more continuous.

Furthermore, a hydro-flattening algorithm has been developed which is able to accurately create a downhill flow of a river. This algorithm was is not yet suited to manage the borders between tiles

at a scale of the Netherlands. Therefore, it would be recommended that this algorithm be further optimized and mostly be integrated into a scaling rework which takes into account tile borders and how water flows between these tiles.

Lastly most of the scaling implementation was saved for later on in the project when a final interpolation method was chosen. However, this process did not adhere to the initial planning and thus the implementation was moved forward a few times. What is recommended here is that within scaling an initial interpolation method is taken when creating the framework, which will allow for basic integration testing, and later on replacing this method with the finalized method. Using this approach some issues may have been highlighted earlier on in the process, thus allowing for them to be addressed in a timely manner.

3.3 Future work

Within a project it remains challenging to be able to complete all the different possibilities that are encountered during the execution. In the case of this project various different algorithms have been briefly touched on, but some have not been developed completely. These various algorithms would be ideal for further research and in depth analysis to create possible new workflows that are more efficient or provide results with higher quality. Some of the opportunities for future work will be highlighted and elaborated on briefly.

One of the first opportunities for future work is based on the constrained Delaunay triangulation (CDT) which has been briefly explored. This approach relied on inserting building and water polygons as constraints for a Delaunay triangulation, creating a triangulation which takes into account how the areas within these polygons should be formed. This reduces the amount of artefacting within the polygons due to interpolation with limited data. Future work based on this approach could include more in depth analysis and further development of this approach, as currently the approach is deemed non-useful as it takes a long time to run, and produces unpredictable artefacts. Improvements would therefore include efficiency improvements and focus on reduction of artefacts within and around the tile edges.

Another opportunity for future work is related to continued development on the true hydro-flattening algorithm. Currently a proof-of-concept has been created showing how hydro-flattening would work when using a river and it's skeleton. In this way the water height flows from high to low along the entirety of the river, thus accurately representing how the river water height is in the real world. This approach relies on a number of different prerequisites, such as having the skeleton of a river and ensuring that it is a clean polyline. Future work could focus on two different aspects of this algorithm, namely: creation of the river skeleton and improvement of hydro-flattening algorithm. The current river skeleton has been created by a number of manual steps done in QGIS involving merging different polygons, creating the skeleton using a skeleton algorithm, pruning branches, and finding a continous path. This process can be fully automated and doing so would benefit the level of autonomous execution the hydro-flattening algorithm can function at. The second portion is finding methods to improve the speed and accuracy of the algorithm. Currently the algorithm runs different sections of a river piece by piece. An improvement here would be more integration between different pieces to ensure that the water bodies are continuous over the entire length of the river from beginning to end.

For scaling, future work could include creating a dockerized version of the program, allowing for limitation of memory and cpu usage in a simple way. Mostly, however, improvements around further artefact reduction and sharing of data between different tiles could prove to be useful. This is related to ensuring that data that is shared along tile borders is ensured to be the same on both sides of a tile. In this case there are different features that can be shared, such as ensuring that the heights of building polygons are the same between tiles, and the same goes for water bodies. Furthermore, further flexbility in choosing which algorithm to use for various different steps (e.g. choosing different algorithm for DSM and DTM depending on the environment) would also be preferable.

A Results ground filtering algorithms

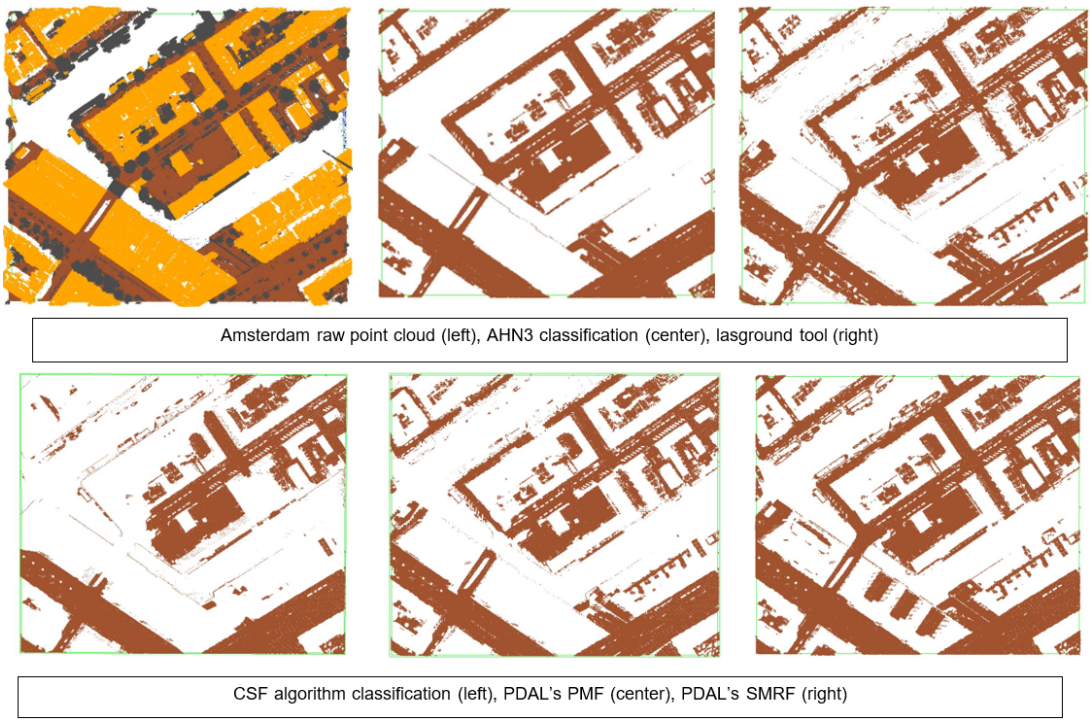


Figure A.1: Amsterdam

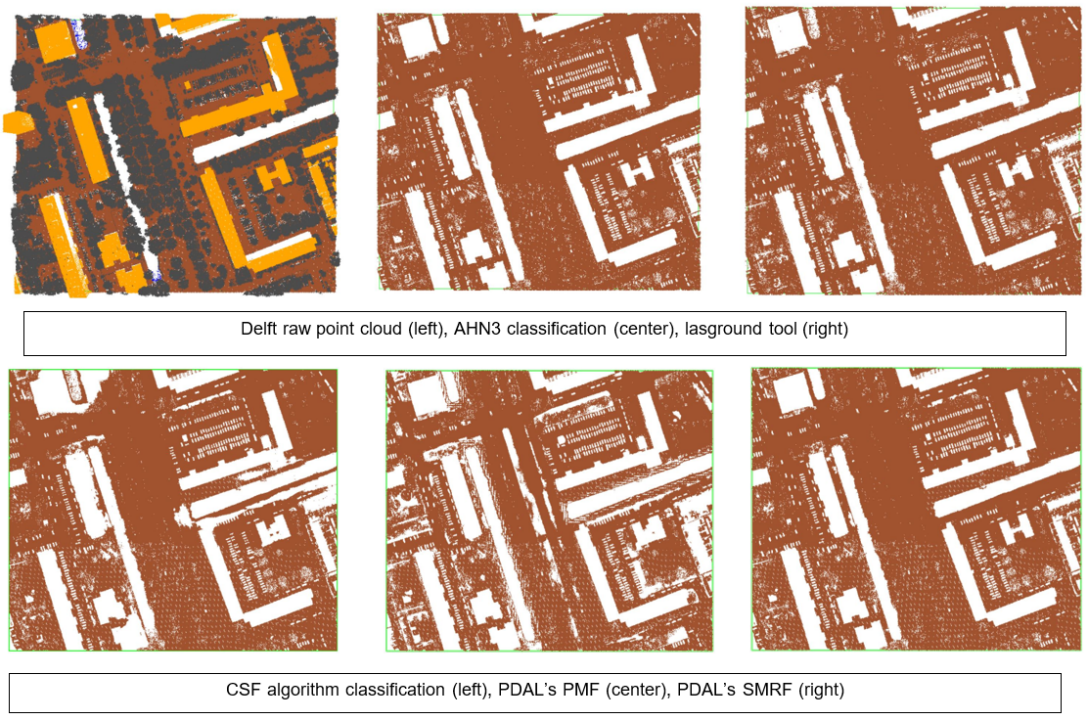


Figure A.2: Delft

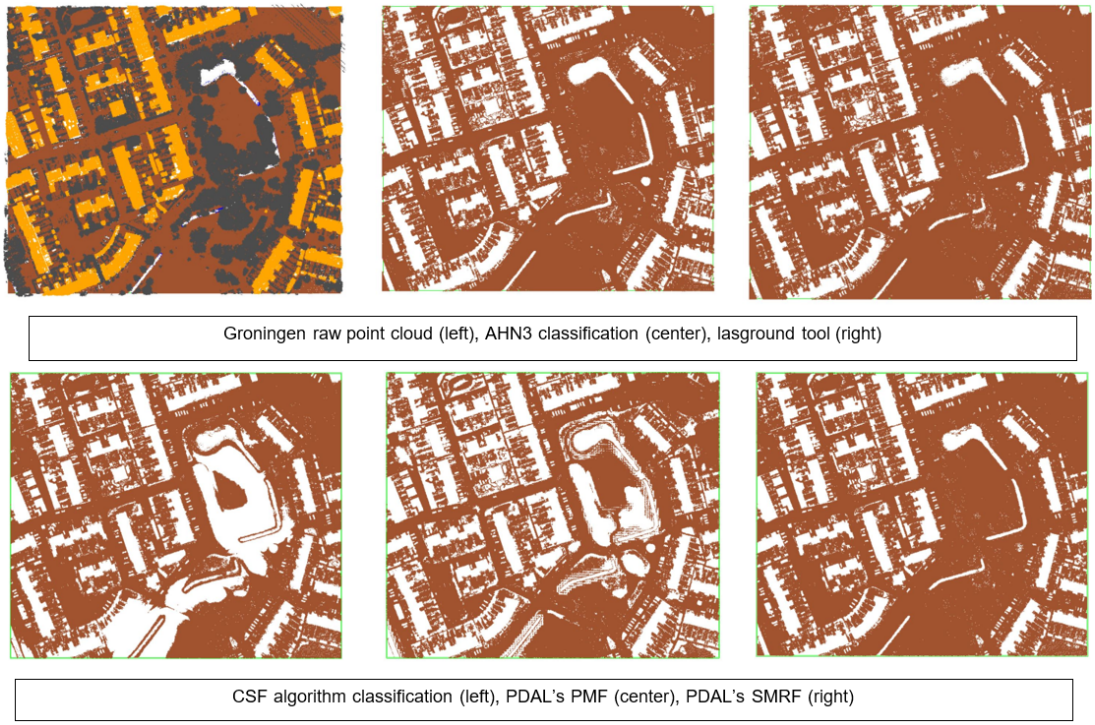


Figure A.3: Groningen

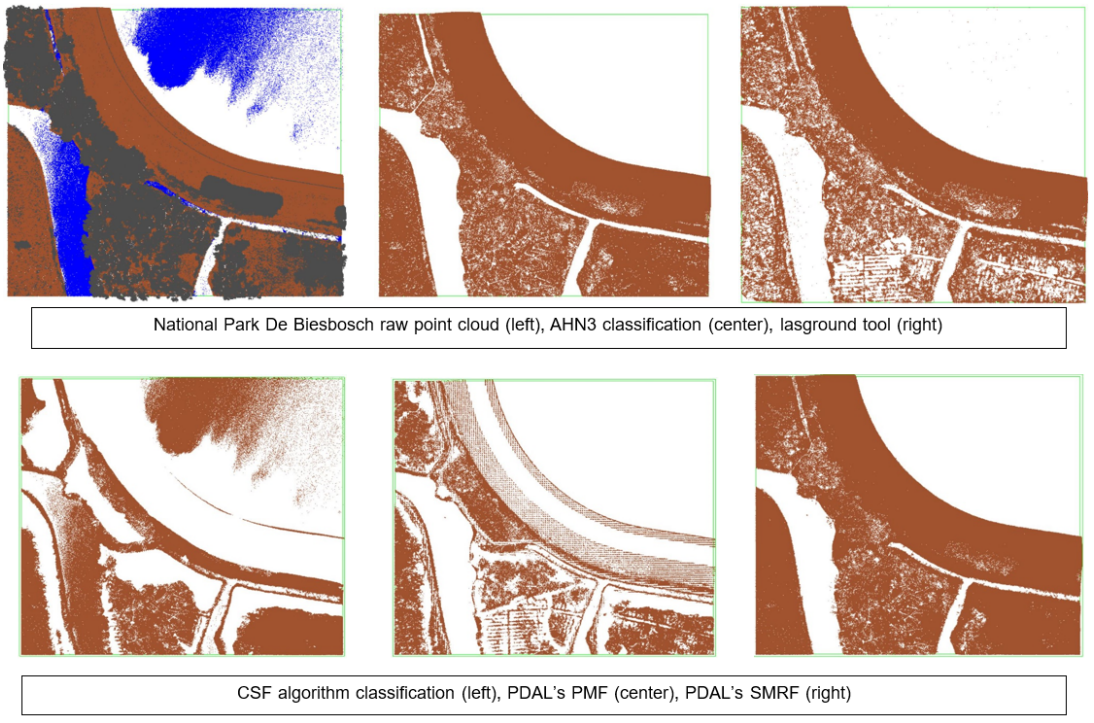
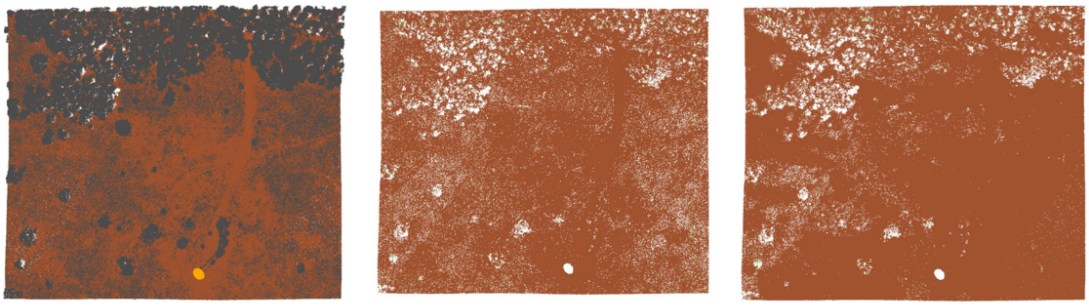
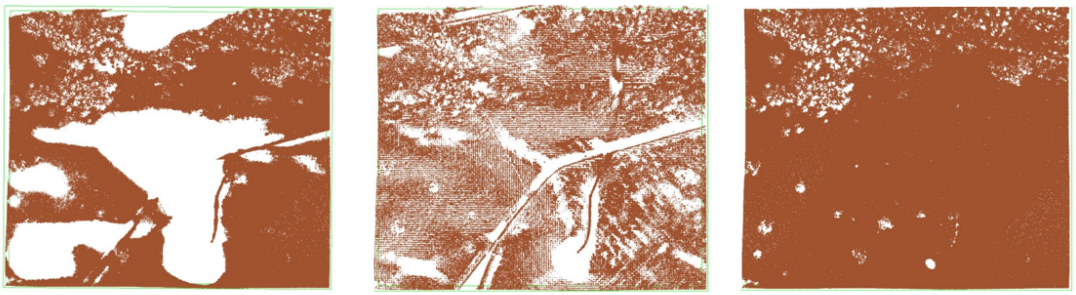


Figure A.4: National Park De Biesbosch



National Park Veluwezoom raw point cloud (left), AHN3 classification (center), lasground tool (right)



CSF algorithm classification (left), PDAL's PMF (center), PDAL's SMRF (right)

Figure A.5: National Park Veluwezoom

Location	Ground filtering algorithm	No. of ground points	Comments
Amsterdam	AHN3 classification	385651	--
	CloudCompare: Csf algorithm	303201	Less points are classified as ground, big holes are visible, the algorithm couldn't identify many ground points.
	LAStools: lasground	401356	More points are classified as ground but points that belong to boats and bridges are wrongly classified as ground.
	PDAL: filters.smrf	453767	Same as lasground tool but points from the roofs of some buildings are wrongly classified as ground as well (worse result).
	PDAL: filters.pmf	349188	Same artefacts as the smrf filter but with less points identified as ground.
Delft	AHN3 classification	614333	--
	CloudCompare: Csf algorithm	593126	The result looks the same but less points are identified as ground.
	LAStools: lasground	617281	The result looks the same but more points are identified as ground.
	PDAL: filters.smrf	628786	The result looks the same but more points are identified as ground.
	PDAL: filters.pmf	533611	Similar result as the smrf filter but with less points identified as ground.
Groningen	AHN3 classification	795751	--
	CloudCompare: Csf algorithm	713450	The result looks the same but less points are identified as ground. Some points in some canals are wrongly classified as ground.
	LAStools: lasground	838133	The result looks the same, but more points are identified as ground. Some points in some canals are wrongly classified as ground.

	PDAL: filters.smrf	854810	The result looks the same, but more points are identified as ground. Some points in some canals are wrongly classified as ground.
	PDAL: filters.pmf	699698	Less point identified as ground. There are large holes.
National Park Veluwezoom	AHN3 classification	817329	--
	CloudCompare: Csf algorithm	646117	Big holes are visible. Less points are classified as ground. The algorithm could not identify a lot of ground points.
	LAStools: lasground	945196	The result looks the same. More points have been classified as ground.
	PDAL: filters.smrf	1000126	The result looks the same. More points have been classified as ground.
	PDAL: filters.pmf	473013	Large holes, a lot of points missing.
National Park De Biesbosch	AHN3 classification	567210	--
	CloudCompare: Csf algorithm	367299	Big holes are visible. Less points are classified as ground. The algorithm could not identify a lot of ground points.
	LAStools: lasground	540870	A lot more points have been classified as ground. Water and trees points have been wrongly classified as ground. The result is better is they are removed but it has fewer ground points and many holes.
	PDAL: filters.smrf	627770	A lot more points have been classified as ground. Water points have been wrongly classified as ground. After removing them the result looks better but some points, probably tree points, are wrongly classified as ground and they are visible (inside the lake).
	PDAL: filters.pmf	243671	Large holes, a lot of points missing.

B DTM & DSM complete visualizations

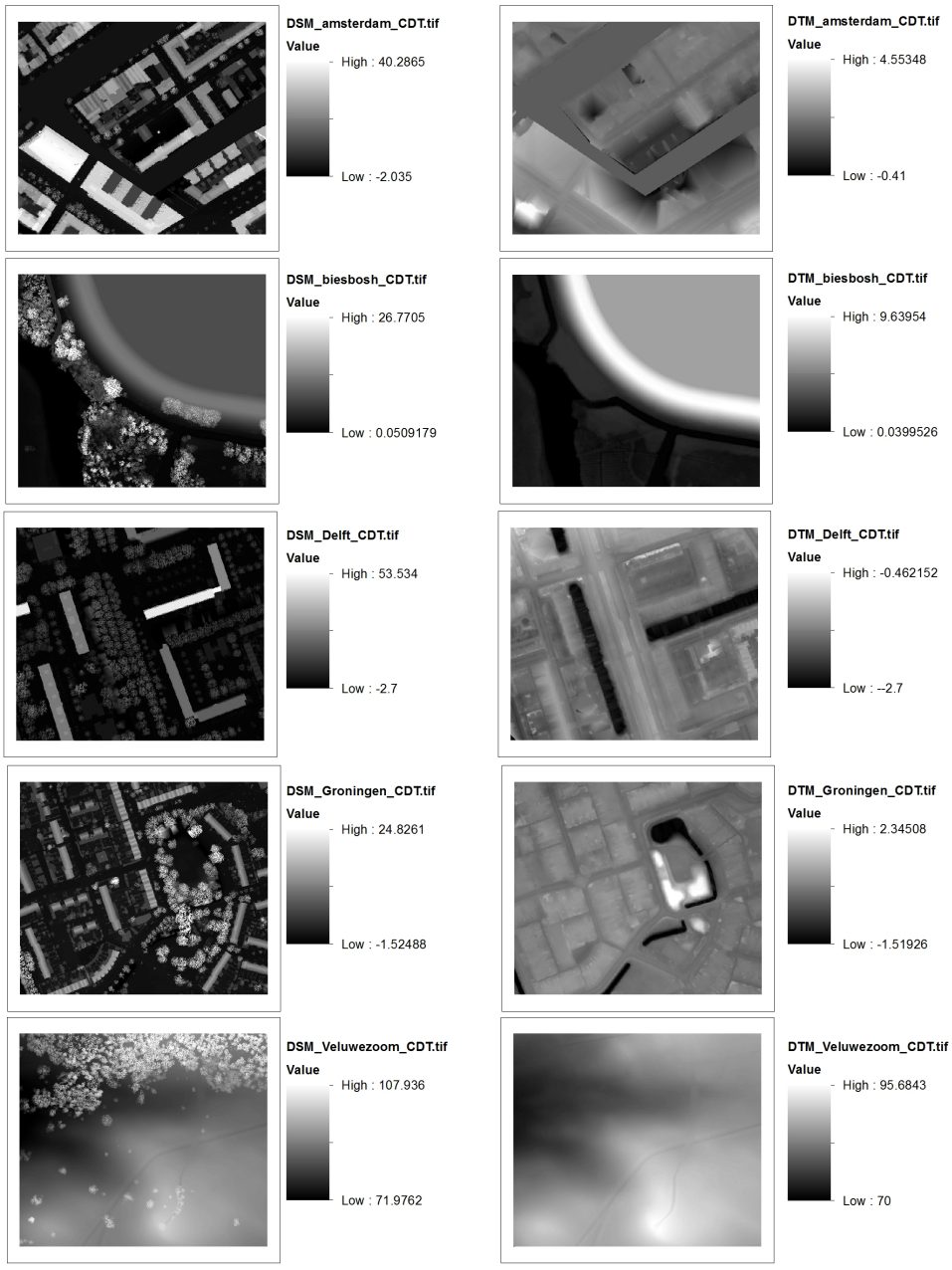


Figure B.1: Constrained Delaunay Triangulation (CDT) TINlinear

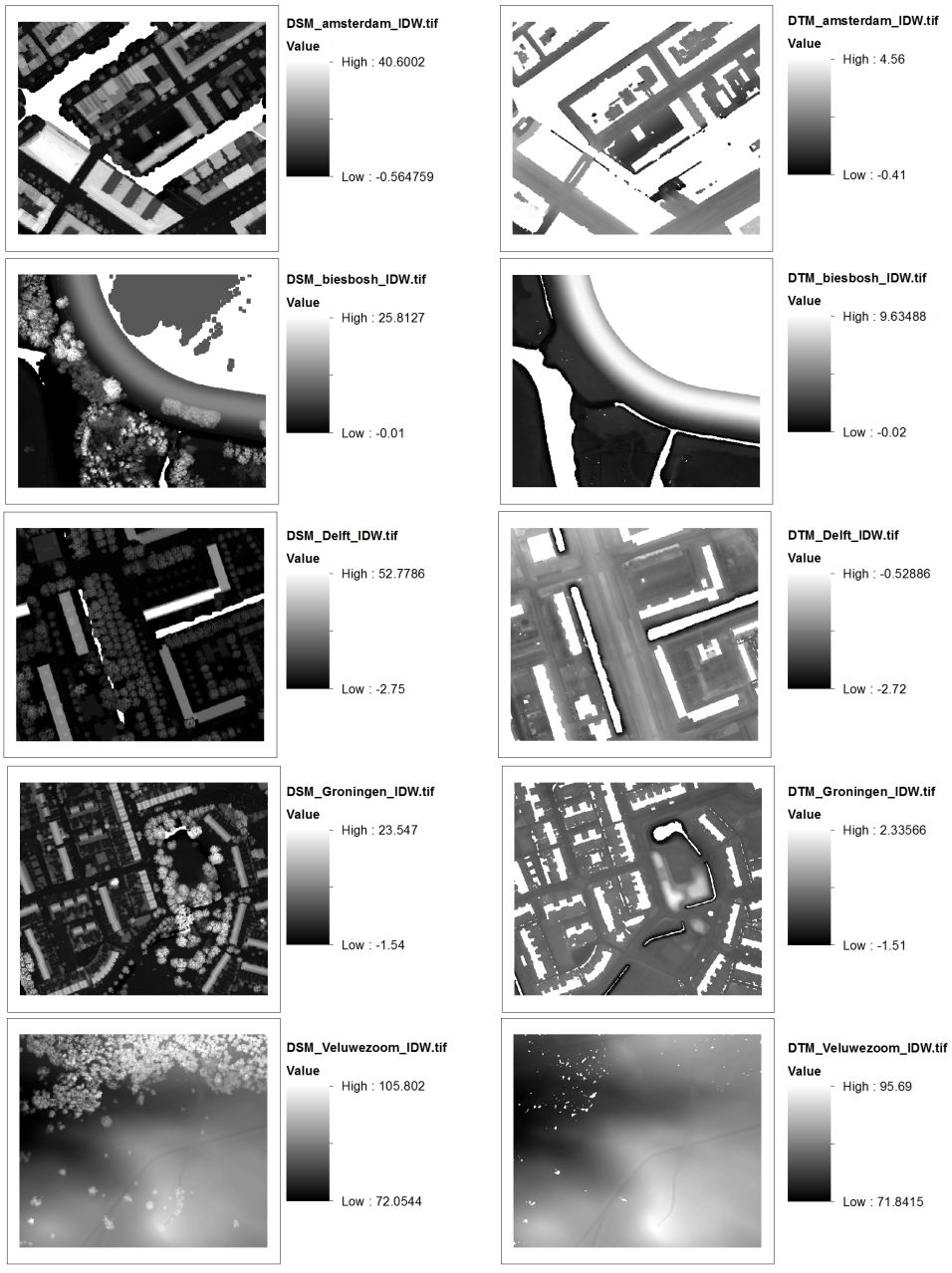


Figure B.2: PDAL-IDW

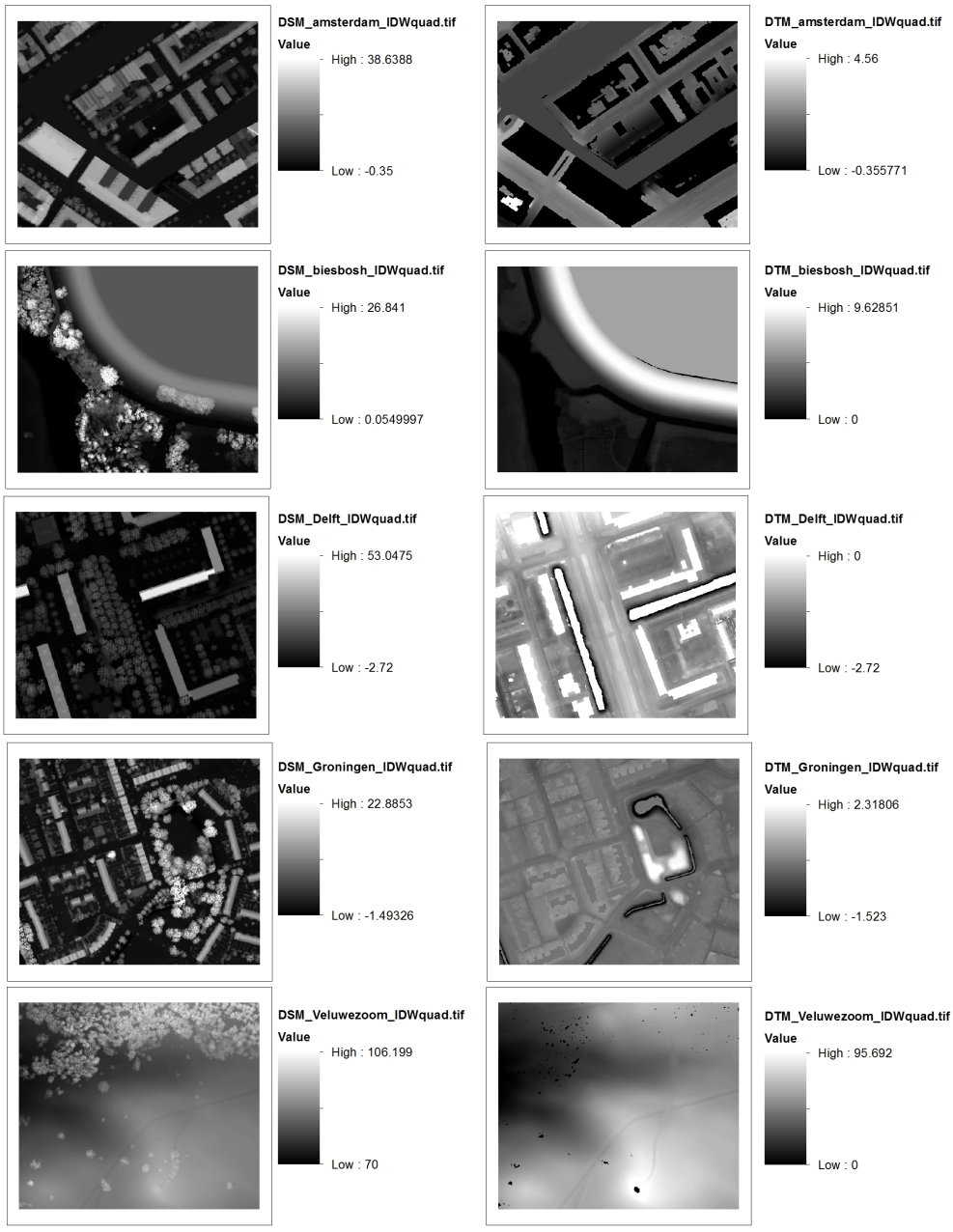


Figure B.3: IDWquad - radial

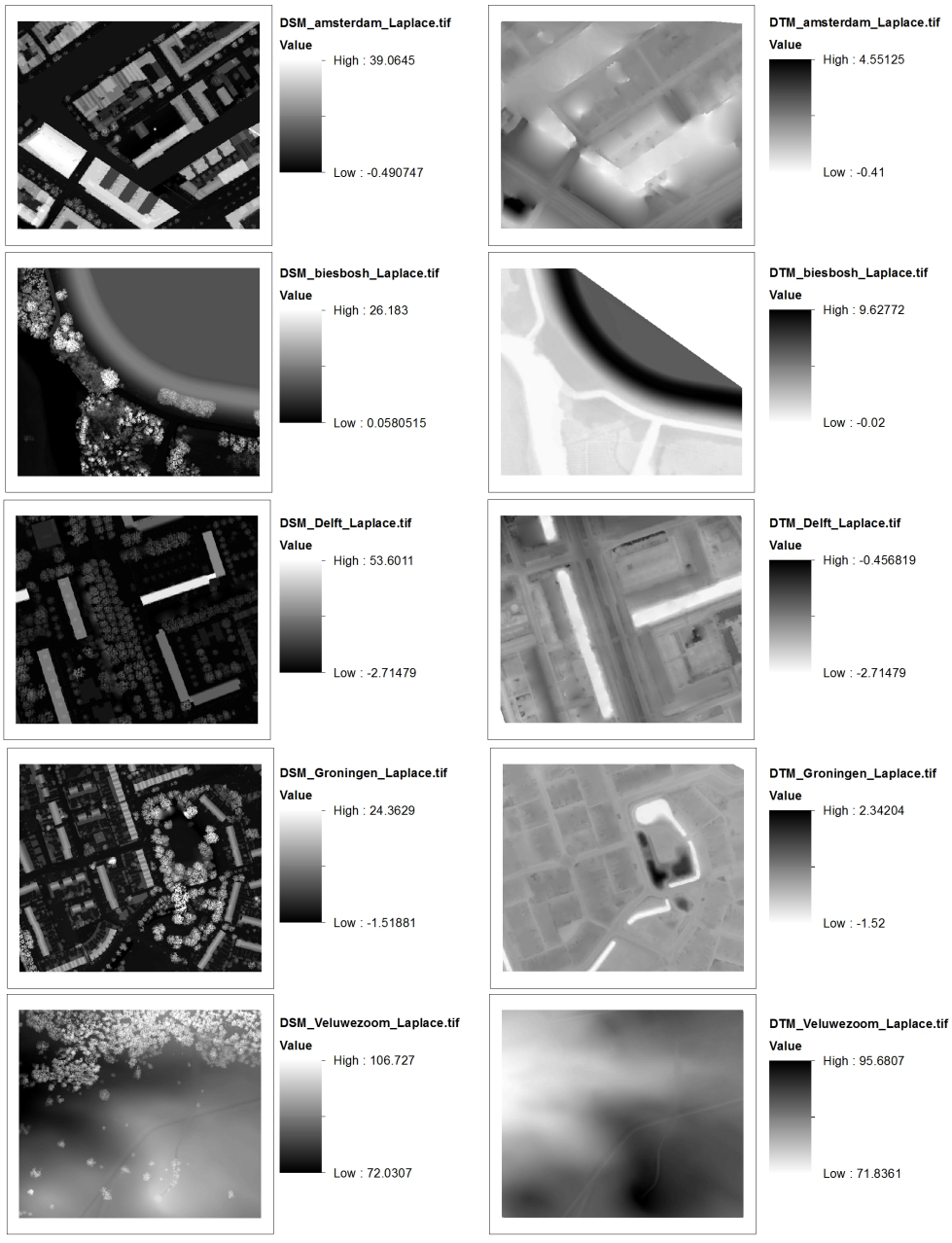


Figure B.4: Startin - Laplace

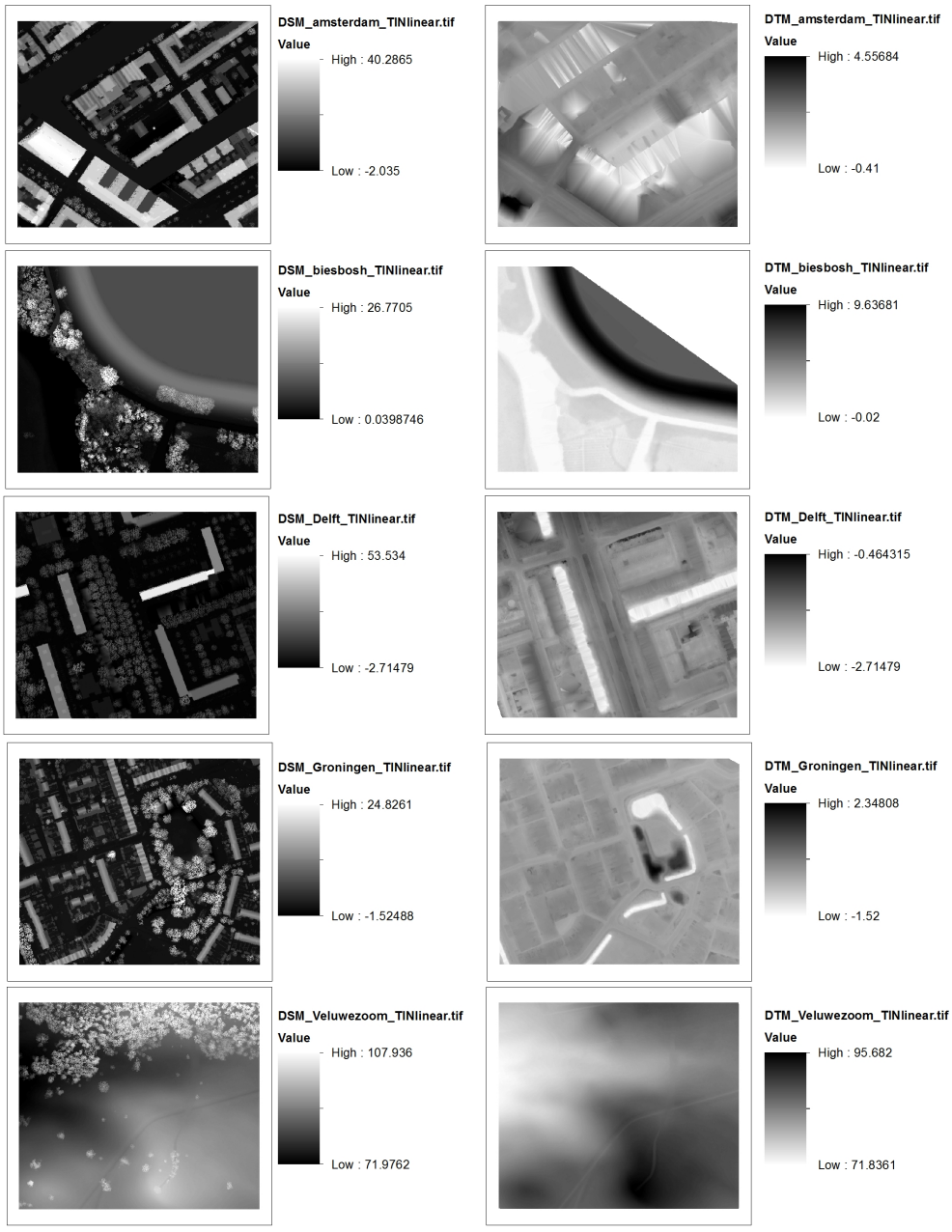


Figure B.5: Startin - TINlinear

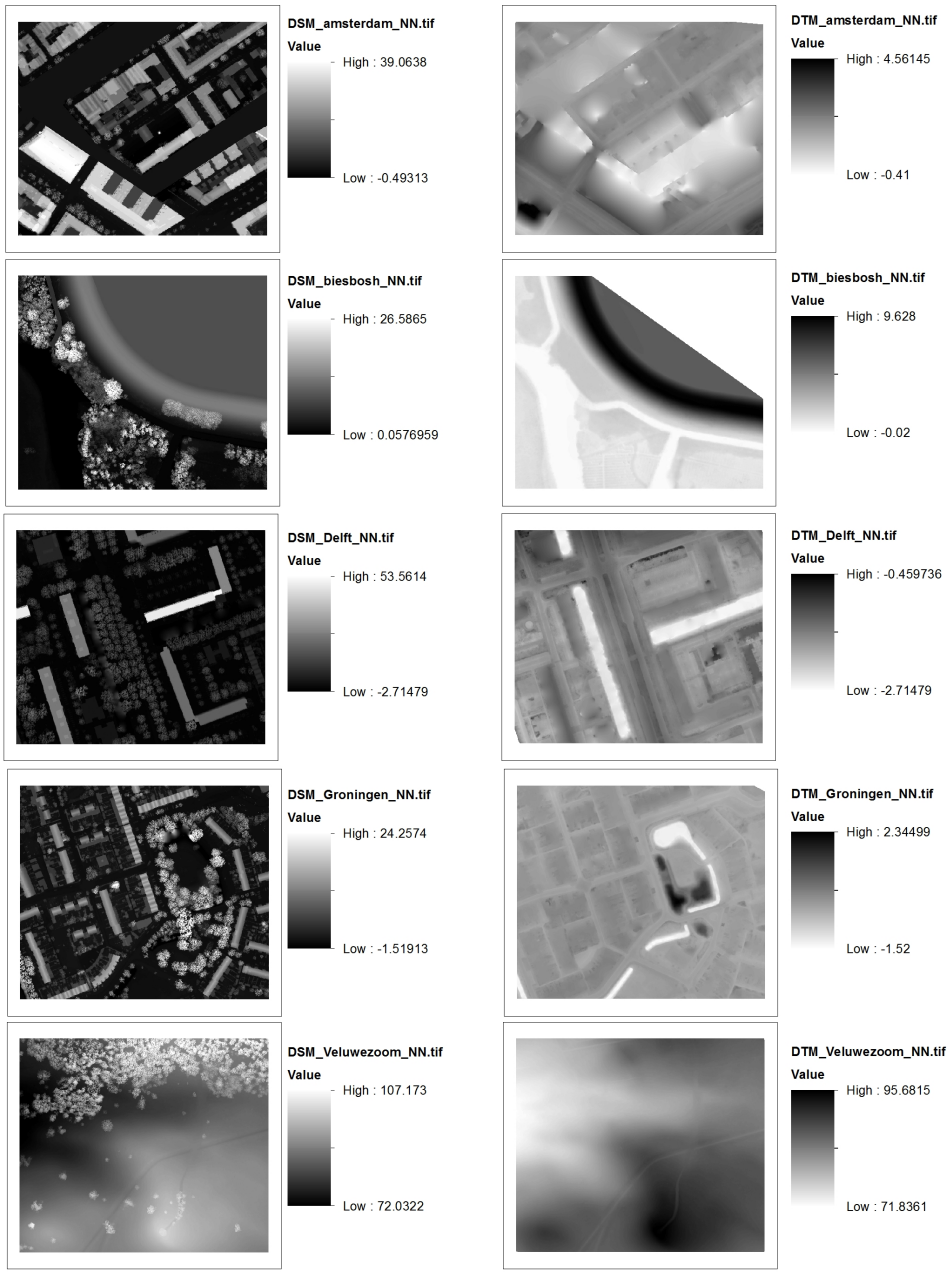


Figure B.6: CGAL - NN

C Vertical differences

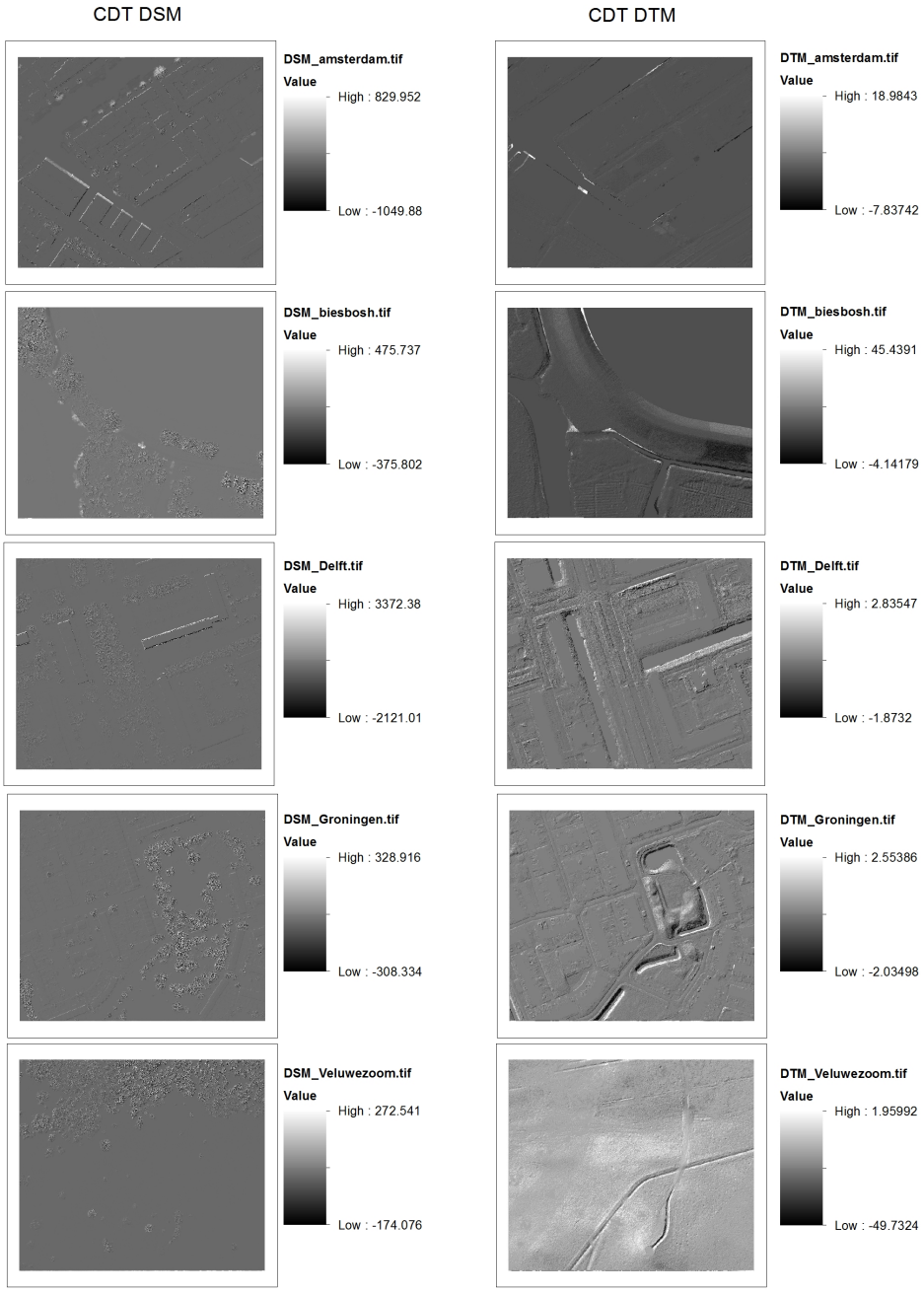


Figure C.1: CDT vertical differences

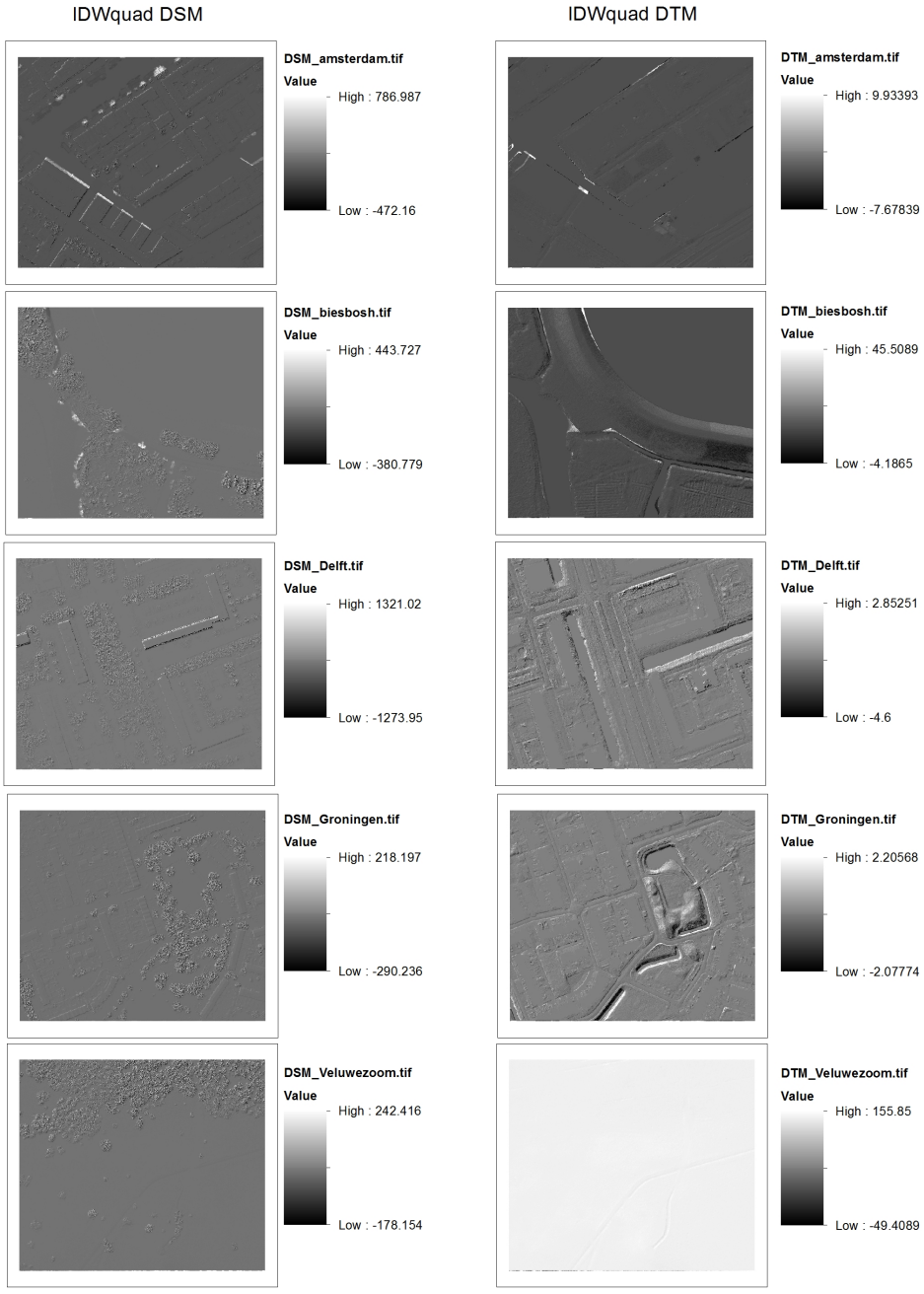


Figure C.2: IDWquad-radial - vertical differences

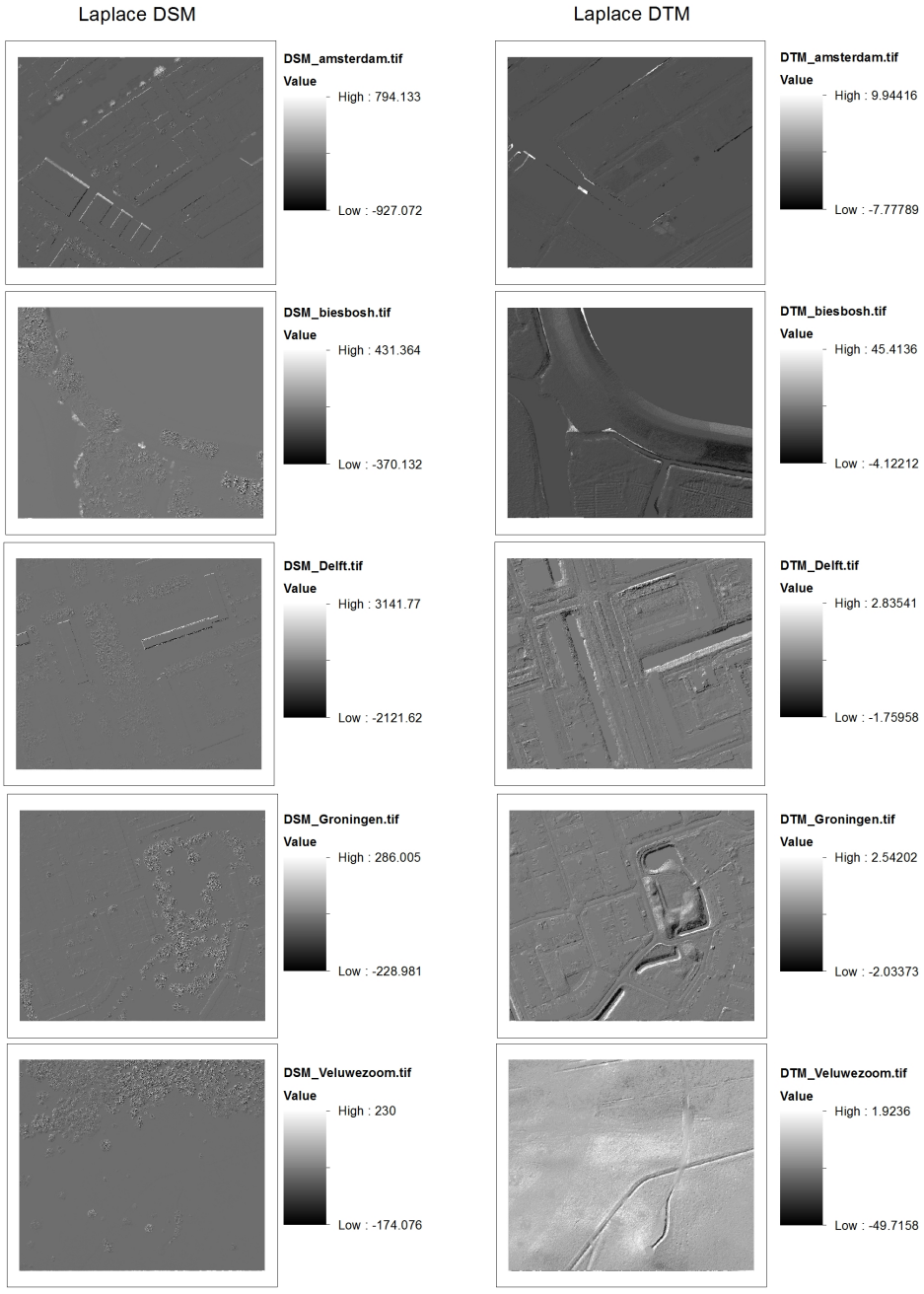


Figure C.3: Laplace - vertical differences

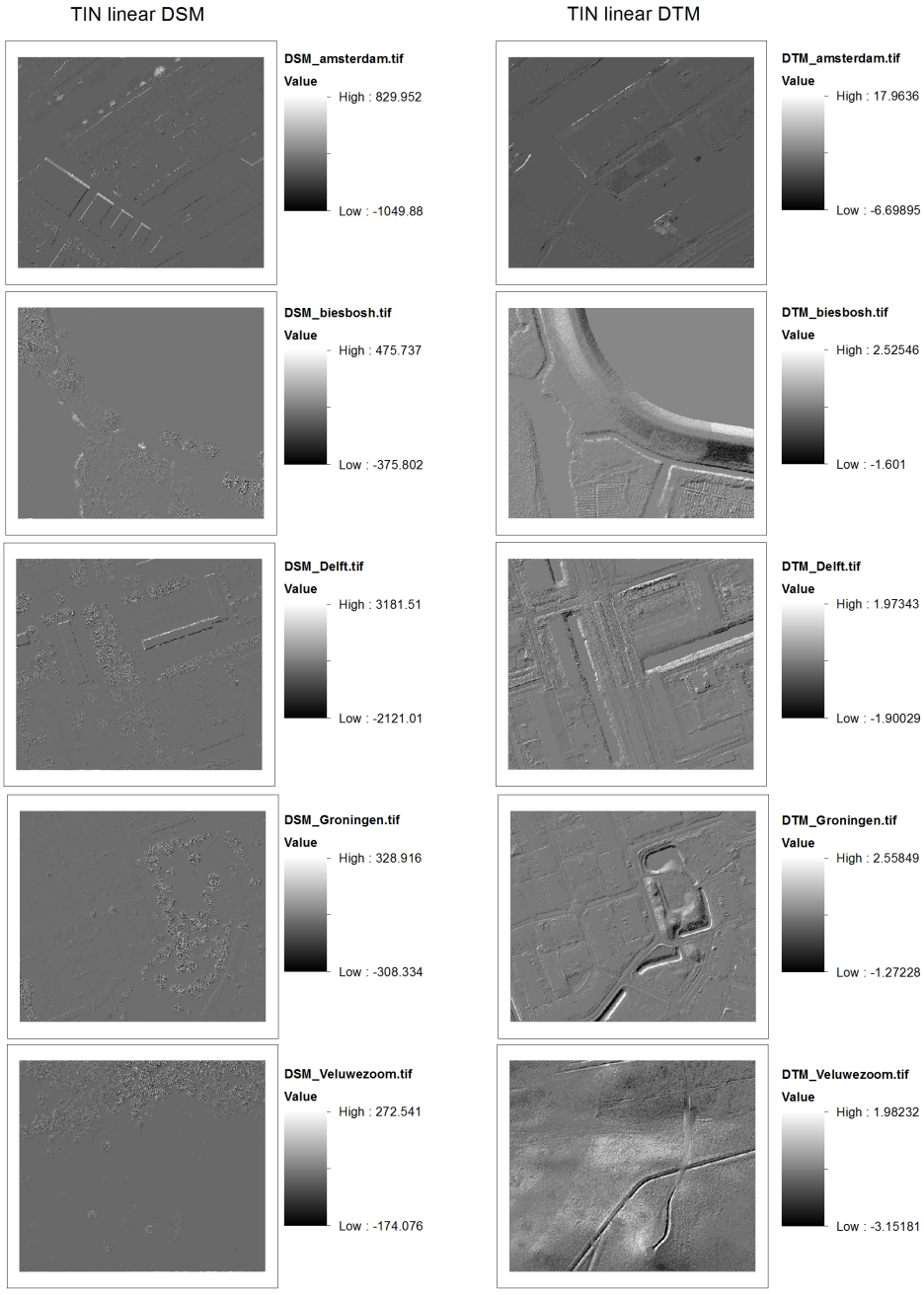


Figure C.4: TINlinear - vertical differences

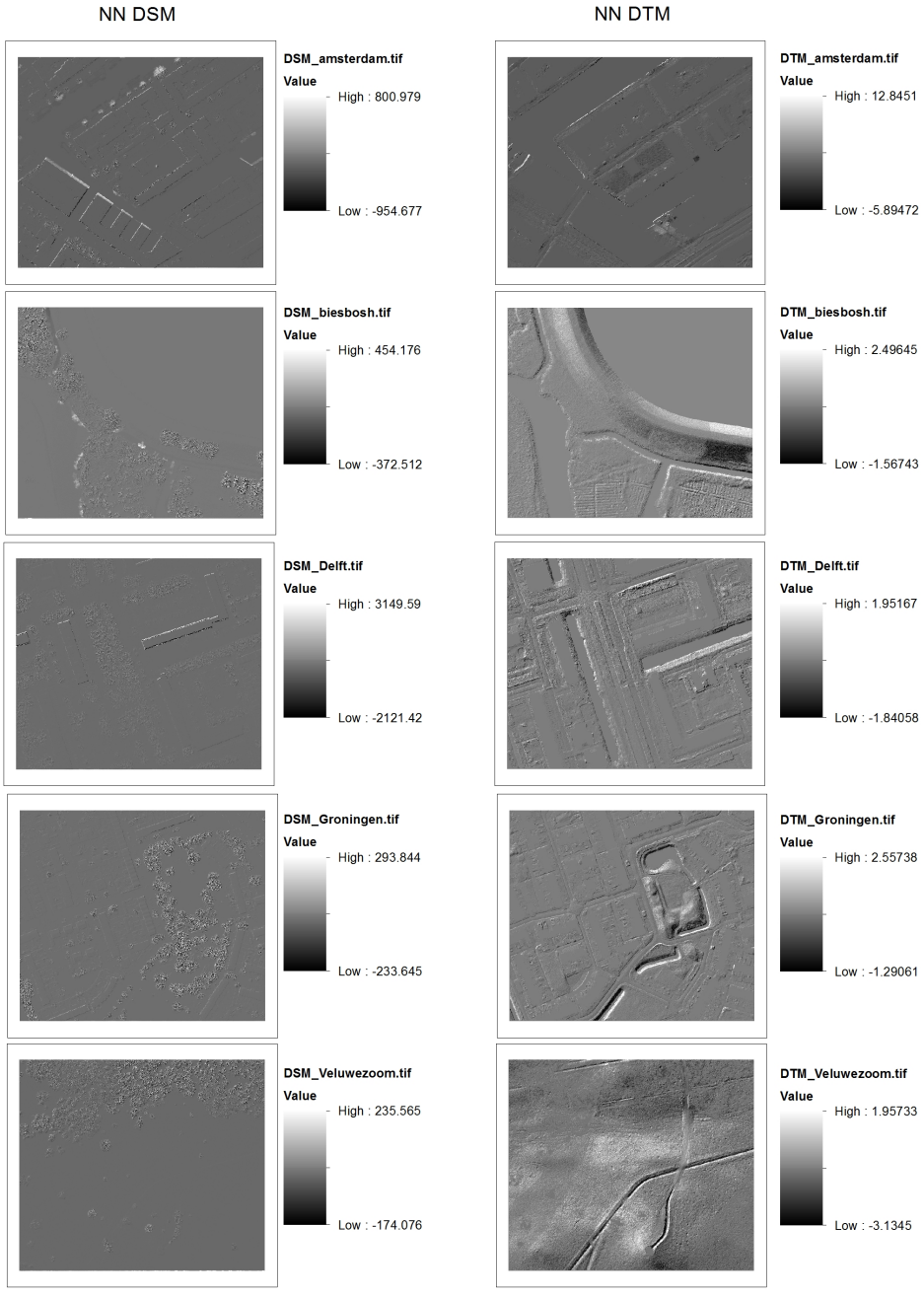


Figure C.5: CGAL-NN - vertical differences

Bibliography

(2020). Ahn. Actueel Hoogtebestand Nederland (AHN): The making of. url: https://www.ahn.nl/ahn-making. (accessed June 17, 2020).

Bell, A., Chambers, B Butler, H., and Gerlek, M. (2020). Pdal point data abstraction library.

Isenburg, M. (2016). Lastools - efficient lidar processing software.

Janet, W. (2016). Mae and rmse — which metric is better? *Human in a Machine World*, 41.

Ledoux, H., Arroyo Ohori, K., and Peters, R. (2020). *Computational modelling of terrains*.

Pingel, T. J., Clarke, K. C., and McBride, W. A. (2013). An improved simple morphological filter for the terrain classification of airborne lidar data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 77:21–30.

Zhang, K., Chen, S.-C., Whitman, D., Shyu, M.-L., Yan, J., and Zhang, C. (2003). A progressive morphological filter for removing nonground measurements from airborne lidar data. *IEEE transactions on geoscience and remote sensing*, 41(4):872–882.

Zhang, W., Qi, J., Wan, P., Wang, H., Xie, D., Wang, X., and Yan, G. (2016a). Cloudcompare - easy-to-use airborne lidar data filtering method based on cloth simulation-remote sensing. 2016; 8(6):501.

Zhang, W., Qi, J., Wan, P., Wang, H., Xie, D., Wang, X., and Yan, G. (2016b). An easy-to-use airborne lidar data filtering method based on cloth simulation. *Remote Sensing*, 8(6):501.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.