URBAN HORIZON

A Technical Report on the Development of a Web Application for Sky View Factor Calculation

Vasileios Bouzas	4744772
Geert Jan (Rob) de Groot	4076125
Melika Sajadian	4708032
Nikolaos Tzounakos	4744268
Teng Wu	4721020

Synthesis Project
2017-2018

Abstract

In this report, we briefly summarize the methodology behind the development of a web application for the municipality of the Hague. This application was developed by a group of students during the Synthesis Project (academic period 2017-2018) of the MSc Geomatics programme of TU Delft University, the Netherlands. The main purpose of this application is the estimation of Sky View Factor (SVF), a necessary element for modern urban planning. To calculate SVF, the methodology used is based on 3D point clouds in order to incorporate the urban environment in its entirety (including vegetation). Development of the webpage, along with use for different location across the Hague, have shown that this approach provides a fast and at the same time, quite accurate calculation for SVF.

Contents

1. Introduction

The main purpose of this report is the summary of the methodology used for the development of an open-source web application upon request of the municipality of the Hague. This application was developed by a group of five students, consisting of Vasileios Bouzas, Geert Jan (Rob) de Groot, Melika Sajadian, Nikolaos Tzounakos and Teng Wu. Development of the application was completed during the Synthesis Project (Academic period 2017-2018) of the MSc Geomatics programme of TU Delft University (the Netherlands). Its main objective is the calculation of Sky View Factor (SVF) for any location in the municipality of the Hague.

Nowadays, the Sky View Factor is widely used in many fields, such as urban climate and heat island studies, human biometeorology and urban planning. SVF refers to the ratio of radiation received by a planar surface to the radiation emitted by the entire hemispheric environment (Johnson & Watson, 1984). It provides a relationship between the visible area of the sky and neighboring surroundings, such as buildings, cars, other artificial objects or vegetation. It is typically represented by a dimensionless value between zero and one, where zero indicates the sky is completely obstructed by obstacles and one indicates there are no obstructions (Brown & Ratti, 2001).

Usually, SVF research focuses only on buildings which mainly obstruct the sky in urban environments. However, trees also play an important role in urban energy exchange and the urban heat balance (Kim, 2014). After discussion with our clients, the two study objects - buildings and trees - are determined, which means we calculate the SVF according to the obstructions made by the buildings and trees.

The structure of this report is the following:
In Chapter 2, we briefly analyze the problem to be addressed and the various parameters needed to be considered. Chapter 3 is a short description of the methodology - preliminary data, preprocessing, main processing and end results. We conclude in Chapter 4 with a discussion on issues still to be addressed by the current methodology, along with some possible future developments.

2. Problem Statement

The main requirements for this web application, set by both developers and clients, were the following:

- Calculate the SVF for any location in the vicinity of the Hague.
- Import multiple locations -at the same time- as a set of coordinates and export the result in a both machine and human-readable file format (i.e. CSV).
- Report how changes on the urban environment (addition or removal of buildings) affects SVF.
- Develop a complete open-source application.

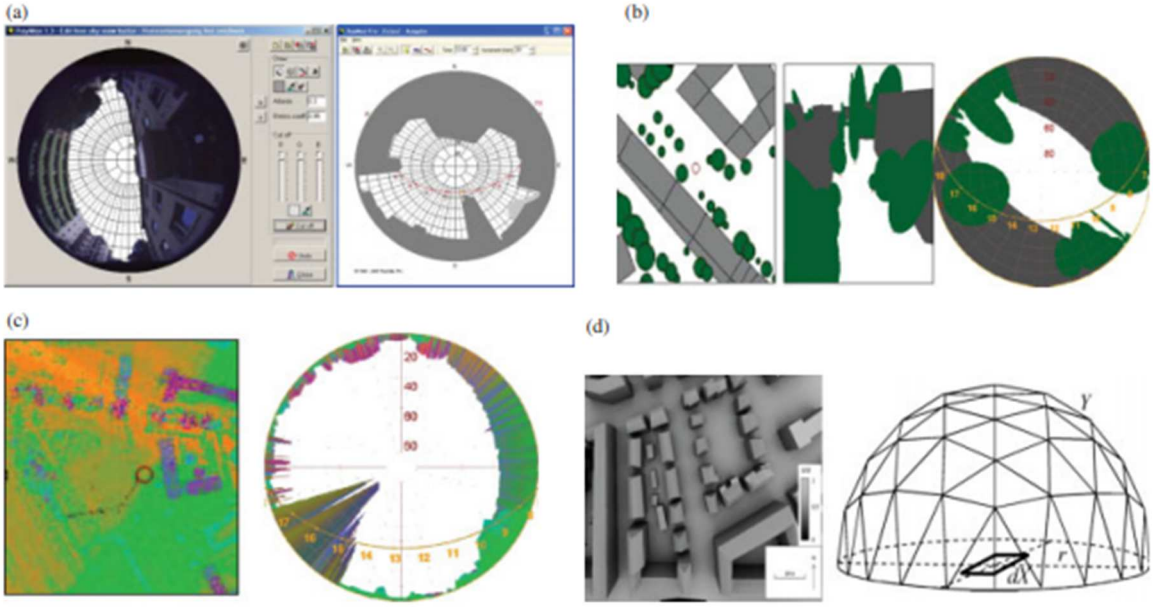
Except for the first one, all requirements are actually functionalities that need to be incorporated in the web page structure. In that respect, the main problem that still needs to be addressed is the definition of a methodology that satisfies two main prerequisites:

- Include the entirety of the urban environment in SVF calculation, mostly buildings and vegetation.
- Calculate SVF as fast - within a reasonable timeframe - and as accurate as possible.

There is a wide variety of available methodologies for estimating SVF in scientific literature, but all of them can be still classified in the following categories, based on their common characteristics:

- Fisheye Lenses: The most common approach for calculating SVF is the acquisition of photographs from various locations distributed in the urban environment with fisheye (ultra wide-angle) lenses. Images are taken by pointing the camera out to the zenith direction. Subsequently, SVF for a given location is analogous to the percentage of sky pixels in the corresponding photograph.
- 2.5D Models: This approach is based on 2D footprints of city features, integrated with elevation information (for example, from DEM or other available sources).
- Raster: This kind of methodology is specifically based on the conversion of 3D point clouds into raster images (hemispherical projections), an approach that resembles fisheye techniques. In a similar way, SVF is estimated as a proportion of pixels containing no points.
- 3D Models: In recent years, the continuously growing production of city models allows SVF study in the 3D environment. The most usual approach is based on the construction of a hemispherical dome around the point of interest. Subsequently casting rays and determining if they are intersecting with surrounding features of the urban environment (buildings, vegetation etc.).

Figure 2.1 - Hemispherical projections of input source types for SVF computation studies
(a) Fish-eye photography-based SVF, (b) vector-based SVF, (c) raster-based SVF, (d) 3D-shape-based SVF



Source: Matzarakis et al., 2007; Matuschek and Matzarakis, 2010; Kastendeuch, 2013

For our project, it is necessary to identify the methodology that is most appropriate for our purposes, based mainly on two criteria: (1) available data (either open data or data from other sources) and (2) computational efficiency (both fast and accurate estimation of SVF).

Since the application needs to be open-source (available to any user for free), it also has to depend on open data and software. In recent years, Dutch authorities have extended the availability and accessibility of spatial data, especially through online services such as the PDOK website and map view service. However, fisheye photographs are not provided through these services since their acquisition is time-consuming and economically inefficient. Hence, fisheye techniques could not be in any case an option to implement, considering that both time and resources are extremely limited for our project.

On the other hand, relevant spatial data such as building polygons are abundant and sufficient for implementation of vector-based approaches. The main disadvantage of these methods is that integration of geometric information with elevation data for any feature of the urban environment (including vegetation and structures other than buildings) is rather difficult or even unavailable - in many cases, either the geometric or elevation information is missing. To avoid that, most approaches of this category focus only on human structures, ignoring vegetation in general (which in reality may block a significant part of the sky), leading to less accurate results. Incorporation of the urban environment in its entirety is necessary to achieve a meaningful level of accuracy in SVF estimation.

From that aspect, 3D city models appear to be the most promising solution for calculating SVF. The main problem with that approach is that availability of 3D models is limited and their production is even more difficult. For example, solid building reconstruction can be accomplished by degrading the level of detail (LoD) - the final result is prismas resulting from the extrusion of building footprints with any available elevation information. Still, even if any possible degradation in accuracy due to lack of details on the roofs of buildings is ignored, 3D surface reconstruction of city objects such as trees is too challenging to be completed within the timeline of our project.

It is concluded that the best method to be used for the purposes of our project is raster-based techniques. Point clouds are easily accessible since PDOK offers point clouds covering the whole extent of the Netherlands for low-density distributions (AHN1, AHN2) and a large part of the country for a higher density distribution (AHN3). Furthermore, processing of point clouds and hemispherical projections are easy to be encoded and implemented with use of high-level programming languages such as C++ or Python. Of course, there are still challenges to be addressed: for example, density of the given point clouds must be high enough in order to achieve the necessary accuracy. This translates into processing large amounts of data, a fact which considerably increases execution time. Taking that into consideration, data preprocessing, along with other techniques, needs to be devised to improve efficiency.

The problem statement in a nutshell, is defined as the following research question:

> *How can an open source web application that generates a SVF for any user-requested point(s) within the municipality of The Hague be developed, using massive point cloud data, while maintaining a reasonable processing speed and providing the highest accuracy as possible?*

The research question will be addressed in the following chapters.

3. Methodology

In this chapter the used methods that yielded the desired results are discussed. The methods used consist of several processes, which are described in full extent including the needed input, inner procedures of the process and the final output that is provided.

3.1 Data Preparation

The initial information and data that are necessary for the web application are:

- Lidar point cloud containing information on the elevation of features of interest on the earth's surface, including the earth's surface itself.
- Points' position, provided by the user through the web application.

Lidar Point Cloud

The algorithm that calculates the SVF requires a point cloud of .las format. The source of the point cloud data is the online portal of 'Publieke Diensten op de Kaart' (PDOK, available at https://www.pdok.nl/nl/ahn3-downloads). The dataset that is selected to use is the AHN3 point cloud, which contains X, Y, Z coordinates as well as other types of information, such as the classification of points. The data format that PDOK provides is in .laz format, which is a compressed form of the .las format. To revert them into a usable format again, an open source executable application is used, called 'LASzip'. By converting the initial datasets, it is possible to process the data. It is also necessary to store all of that data efficiently, so that it can later be retrieved by the main algorithm. The redundant points are removed from the dataset by clipping them out of the study areas' extent.

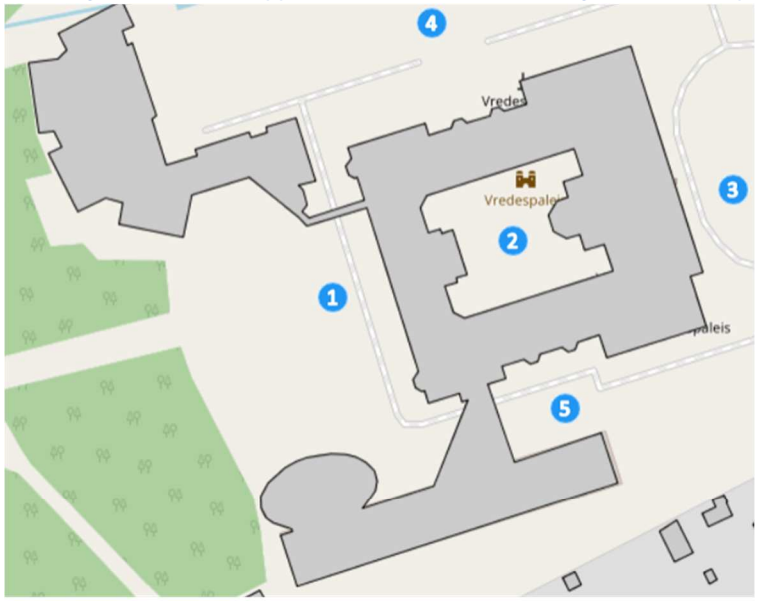
In order to speed up the main algorithm (SVF calculation), the whole point cloud of The Hague was cut into 9,841 tiles with a size of 100 by 100 meters. This specific extent is in accordance with the selected method for SVF calculation, as will be described in section 3.3. A tailor-made python script is developed for this purpose. For any selected viewpoint, the tile containing this viewpoint is determined. Using this tile, the eight neighbouring tiles are determined, so the nine tiles are used as the input point clouds for the main algorithm.

Viewpoint Coordinates

In order to be able to compute the SVF for a certain point, it is necessary to have that point's coordinates on a known reference system. This is handled through the web application. On the application's interface, the users are able to select the points that they want, which are represented as point features on the map. Immediately after creating the points, the application is capable of recognizing their coordinates in the "Amersfoort RD/new" reference system. The cartesian coordinates are then presented on a table on the application interface.

When sending the request for SVF calculation, the viewpoint coordinates are embedded in the request. This triggers a background python script to process that information, and produce the desired results. Figure 3.1 depicts the created points on the map interface, their corresponding coordinates and SVF values in a table positioned below that.

Figure 3.1 - Web Application Interface of Creating Points on Map



ID	X	Y	SVF
1	80109.941	455910.166	72%
2	80157.813	455924.411	57%
3	80216.251	455937.031	74%
4	80137.208	455982.235	79%
5	80170.971	455879.904	48%

Source: Own work

3.2 Web Development

In this section the functional requirements and the architecture of the web application, as well as the facilities that it offers are discussed.

The functional requirements are:

- AHN3 tiles stored on local files.
- Web server for handling HTTP requests and responses.
- PHP interpreter, the medium between the webserver and the python script.
- Python script which does the main processing, i.e. SVF calculation.
- The scripts in JavaScript language, both on client and server side to handle connections between client, servers (Application server and external servers).

- OpenLayers API to facilitate interactions of the user.
- HTML and CSS to markup and style the body of the web page.

The web application is constructed based on one of the well-known client-server architectures called three-tier architecture, which includes: presentation, logic and data tiers. The data tier is mainly about storing and managing data for processing. The only data that we are storing in the local file is point cloud data. The rest of needed data is directly requested from PDOK web services.

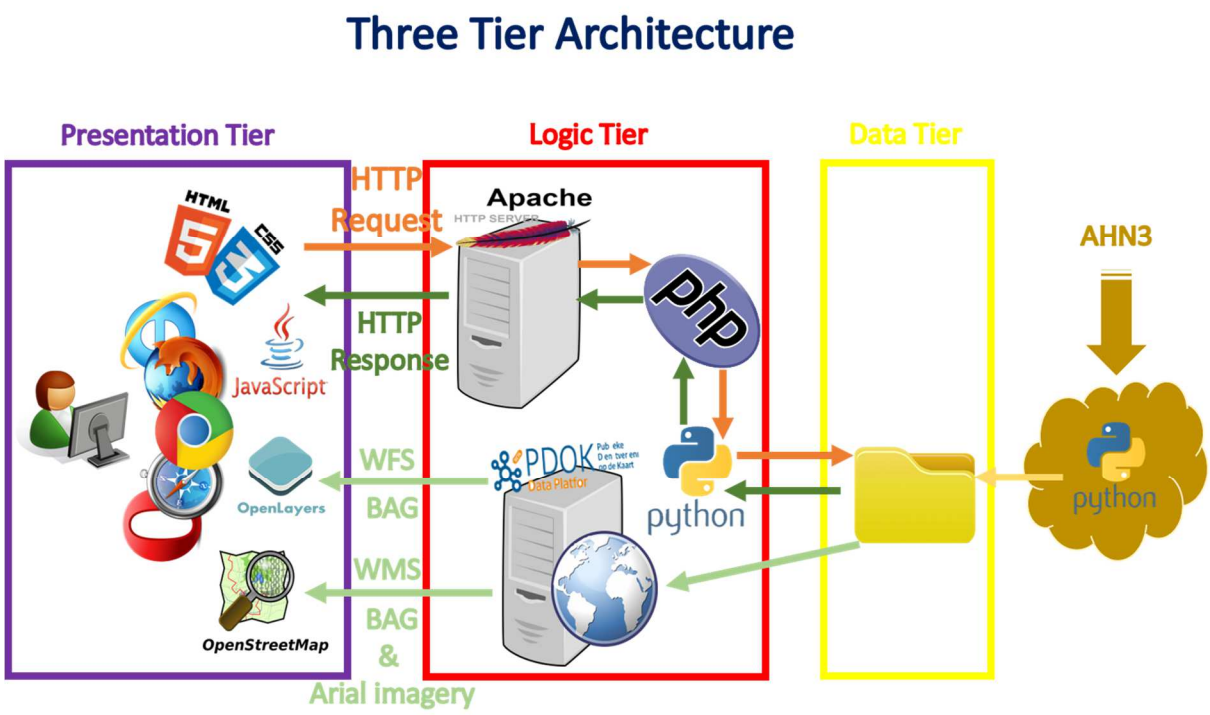
The logic tier coordinates the application, processes, commands, logical decisions, evaluations and performs calculations. This layer is also the medium through which the data is transferred between the two other layers. In this layer, one web server is used to handle the task: Apache coupled with PHP in one open source software called 'XAMPP'. The SVF calculation is done through a Python script which takes the coordinates of a certain point, the geometry of polygons to be added or removed from the user and reads the point cloud data for computation. Whenever a user makes a request to the application, the Apache server runs a PHP program which triggers the Python script and receives the output of the SVF calculation and embeds it in the HTML file.

The presentation tier is the top level, which is the user interface. The main function of the interface is to translate tasks and results to a form that users can understand. In this layer the user is able to see the building polygons overlaid on 'OpenStreetMap' (OSM). The building polygons are directly requested from PDOK's WMS service of its BAG dataset. Using the OpenLayers API and JavaScript codes the user is able to:

- zoom and pan on the map;
- create points (for SVF calculation) and show them on the table;
- select points both from the table and map canvas;
- delete points;
- clear the canvas;
- eliminating existing buildings from calculation (whose geometry is retrieved from Web Feature Service (WFS) of PDOK);
- adding a building with a certain height;
- submit a request for SVF calculation;
- recalculate upon changes.

As web maps are in the WGS84 Web Mercator (EPSG: 3857) coordinate system, the API and JavaScript codes also enable the transformation between the web coordinate system and national Dutch coordinate system (RD new). HTML is an XML like format by which the website is formulated. Finally, CSS is used for styling the web portal.

Figure 3.2 - Three Tier Architecture



Source: Own work

3.3 Sky View Factor Computation

This section dives into the inner processes of the constructed SVF-calculation algorithm and will explain these processes in detail.

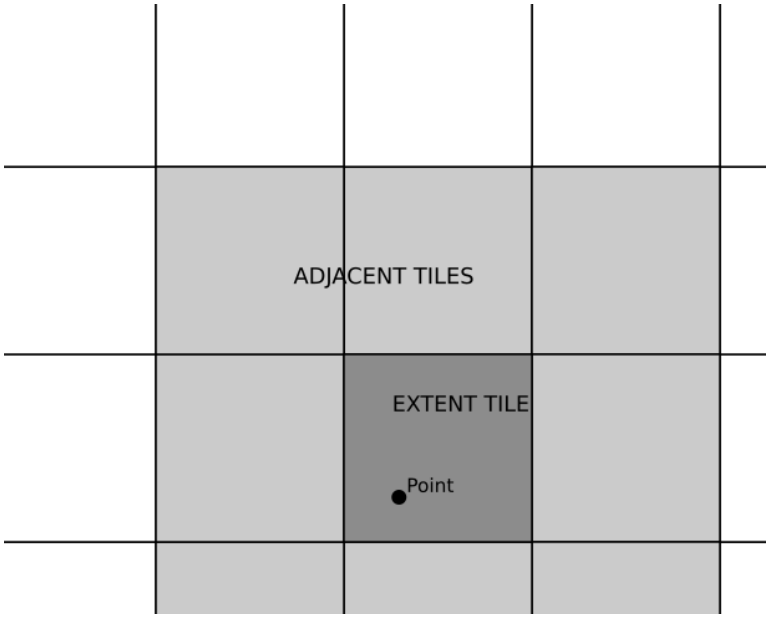
Initialization

The initialization of the algorithm starts with getting information about a point's coordinates, as well as information on the place where the Lidar point cloud data is stored.

Getting Point Cloud Data

The size of the data stored in order to facilitate a functional program for the extent of an area is quite big. Each of the tiles provided by PDOK contains hundreds of millions of points, which translates into many GigaBytes of data. Since the algorithm is supposed to be efficient and produce results within a reasonable timeframe, it is deemed necessary to style the stored data in a way such that we can retrieve the minimum required number of points. For this the tiling of the point cloud is used, as is described in section 3.1. The 100 by 100 metres tiles are indexed using their relative position in a row-column structure. As such, by knowing the coordinates of a point, it is recognized in which tile's spatial extent it lies. After that, the tiles that are horizontally, vertically and diagonally adjacent to the initial tile are retrieved. The resulting nine tiles are kept as a tile grid (as seen in Figure 3.3).

Figure 3.3 – Selecting Adjacent Tiles



Source: Own work

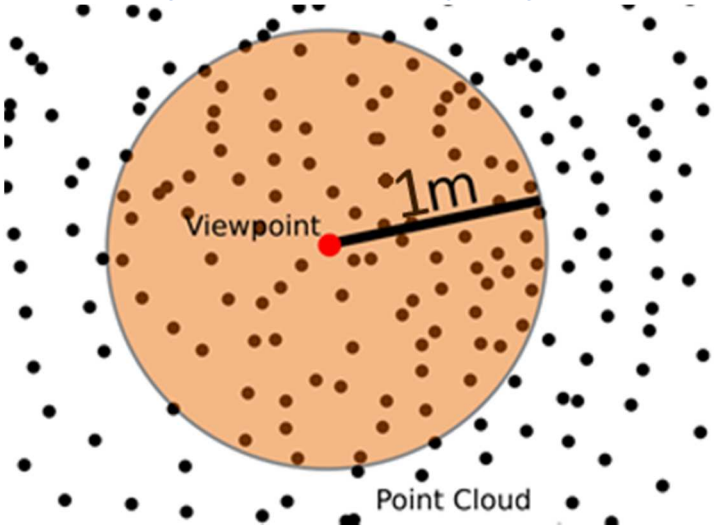
Calculate Viewpoint Height

On the front-end side of the application, only 2D horizontal coordinates are made available to the user. However, in order to calculate accurate SVF values, the elevation of the viewpoint needs to be taken into account. As such, another method needs to be implemented to assign elevation to a viewpoint. An approach to deal with this is to utilize the point cloud points that are classified as ground.

In order to provide the user with a sense of continuity, a distinction between ground viewpoints and on-building viewpoints should be made. If the number of ground points is larger than that of building points surrounding the viewpoint, the viewpoint is considered to be on the ground and vice versa.

Furthermore, from the selected tiles those points that are lying within a set radius are kept (Figure 3.4). To ensure the distinction between building- and ground- viewpoints is established accurately, a one-meter radius can be considered appropriate. The elevation of the surrounding points can be calculated, using only the points of the most prevalent class. This calculated value can then be assigned to the viewpoint.

Figure 3.4 - Selection of Neighboring Points

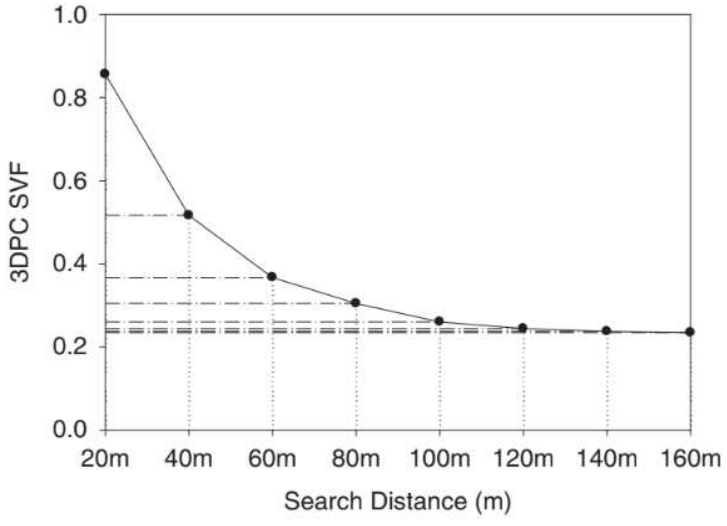


Source: Own work

Retrieving Points

In the same manner as depicted in figure 3.4, using a different radius, the points that are deemed to be of use are selected. Those points represent all features that are obstacles to the viewer, preventing a clear view to the sky. The literature of An et al. (2014) suggests that a radius of 100 metres allows for an effective calculation of the SVF (Figure 3.5). In this case, the most accurate result is for a radius of 160 metres, however a radius of 100 metres gives a result that is less than 2% different. Therefore this radius is accepted in order to increase the computation time of the algorithm.

Figure 3.5 - SVF related to Distance of Selected Points



Source: An et al. (2014)

Out of all the points that fall within the defined radius, only the points that are classified as vegetation or buildings are used for the calculation. Because it is assumed that those are the main elements that play a part in the obstruction of the sky from a certain viewpoint.

Sky Dome Creation

A model is necessary for supporting the calculation of the SVF on a certain position. A dome is selected to be the model that supports the calculation. The dome is a representation of the sky, going from the horizon all the way to the zenith (directly on top) of the viewpoint. The dome can be split into sectors based on horizontal and vertical directions, in essence creating a dome-like shaped grid. The units used to split the sectors are 2 degrees horizontally, and 1 degree vertically, which are considered as appropriate values for calculation (Gal et al. 2009).

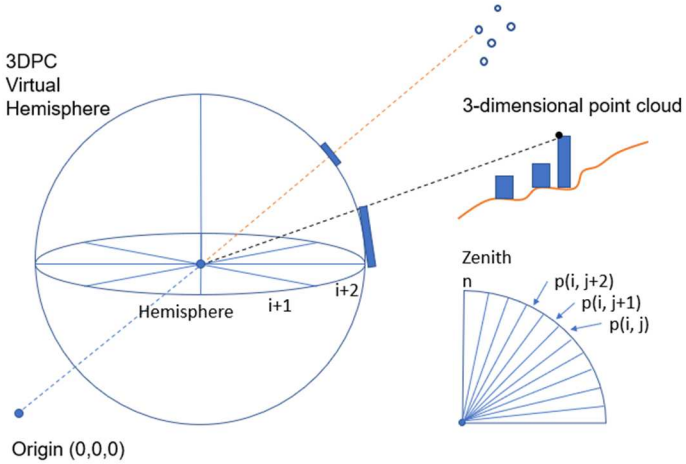
Figure 3.6 - 3D Dome Grid Model

Source: Bourke P. (2001)

Sky View Factor Calculation

At this point, all that is necessary in order to calculate the SVF is present. Initially, the existing point cloud points need to be projected on the dome, in order to find which sectors are obstructed from view. Whenever a point falls within the extent of a certain sector, that particular sector is deemed to be obstructed. It is also taken into account that if multiple points are falling within the same sector, the point that lies closest to the viewpoint is determined to be the one that is obstructing the view. Furthermore, in case that the obstructing element is a building, then also all other sectors below it in that direction will become obstructed by that same element (unless another element is indeed closer in a certain sector below). A visual application of this methodology can be seen on Figure 3.7.

Figure 3.7 - Visual representation of 3D Point Cloud Technique



Source: An et al. (2014)

After all relevant sectors are deemed as obstructed or not, the next step would be calculating the areas of the sectors. For this part, a simplified method is used, which is calculating the sector areas on a 2D projected dome. As such, each elementary area becomes a circular sector, the area of which is easier to calculate.
The proportion of the unobstructed area to the total area of the projected dome, denotes the number that is the Sky View Factor of the viewpoint. This calculation is presented in Equation 3.1:

Equation 3.1 - SVF calculation

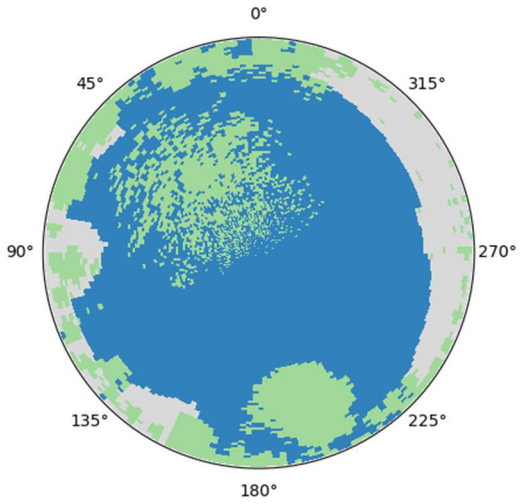
$$SVF = \frac{Total\ Area\ -\ Obstructed\ Area}{Total\ Area}$$

Creation of Sky View Plot

One final remark that is considered important is the visualization of the results, in order to help the user understand the output of our algorithm to a greater degree. For this reason, the choice has been made on creating a circular plot that represents the projected dome, containing all of the dome's sectors, and coloring them differently depending on the element that is obstructing the sky in that sector. This visualization conforms to the result derived from fisheye view cameras.

The dome model with information on what feature is obstructing which sector is already prevalent and needs to be visualized. With a plotting library, it is possible to create a visualization of the result from within the python script, in the way that is explained previously. The plot image created contains the sectors that are colored according to whether the sector is unobstructed sky, or obstructed by a building or vegetation. The image is oriented so that it is pointing to the north and contains information on directions. Figure 3.8 is an example of the outcome of this process.

Figure 3.8 - Projected Dome Plot Image



Source: Own Work

Removing & Adding Buildings

The main task of our application is the calculation of SVF, assuming that the urban environment does not change over the course of time. Of course, this is not true and the need for a more dynamic approach that can adapt to changes in the urban surroundings becomes apparent. In that respect, development of functionalities to both add and remove buildings is necessary in order to also report how changes in the urban environment reflects to SVF.

Removing buildings is one of these tasks. Since 3D point clouds are used to calculate SVF, the points that lie within the corresponding footprint need to be ignored from the computation process. For that purpose, the Point-in-Polygon algorithm (PNPoly) is implemented to identify which part of the point cloud lies within the building polygon. PNPoly is a faster alternative to the ray-casting algorithm, which is most commonly used to address Point-in-Polygon problems. PNPoly conducts the least number of conditions necessary and decreases the computation time as much as possible.

The addition of new buildings to the calculation process is a bit more complex. One way to go about this is to identify, again with PNPoly, what existing points lie inside the corresponding polygon of the added building, and assign them with the newly determined elevation (building height) and class (building class) values. The main issue that this approach raises is that the density of that area of the pointcloud may not be sufficient for the SVF calculation.

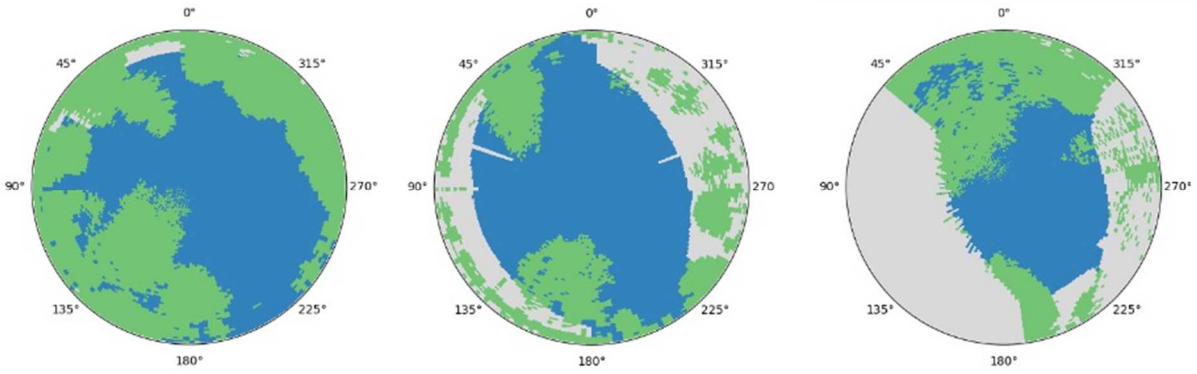
The other option is to clear all existing points that lie within the corresponding polygon of the added building, and fill this area with a newly generated grid of points with a predefined density, guaranteeing a sufficient density for the SVF calculation. It

is reasonable to assume that the second option provides a higher certainty about the accuracy of the results, which is why that is the prefered method.

3.4 Outputs

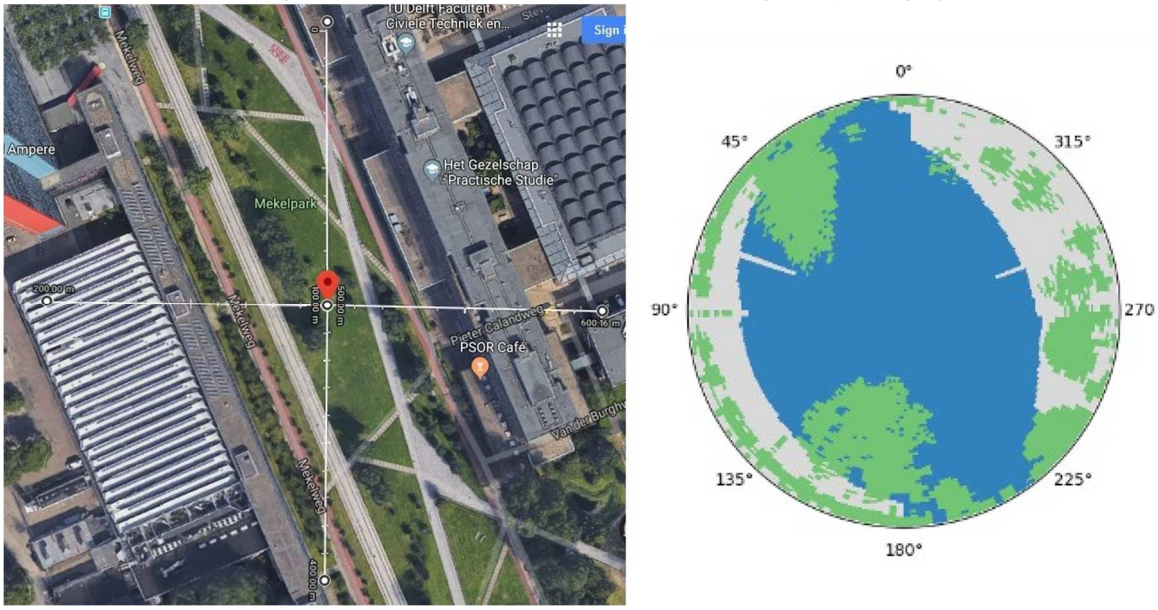
Finally, results from the algorithm have to be returned as output on the application. The SVF value and the plot image related to a point are returned and loaded on the web application interface. This allows the users to see the final results and derive their own conclusions, or even export the information for later study. Example fisheye plots can be found in figure 3.9, while 3.10 provides a side-by-side comparison with Google Maps imagery, to provide a more insightful way to validate the output.

Figure 3.9 - Example output plots for Skyview Factor



Source: Own work

Figure 3.10 -Side by side comparison with Google Maps imagery



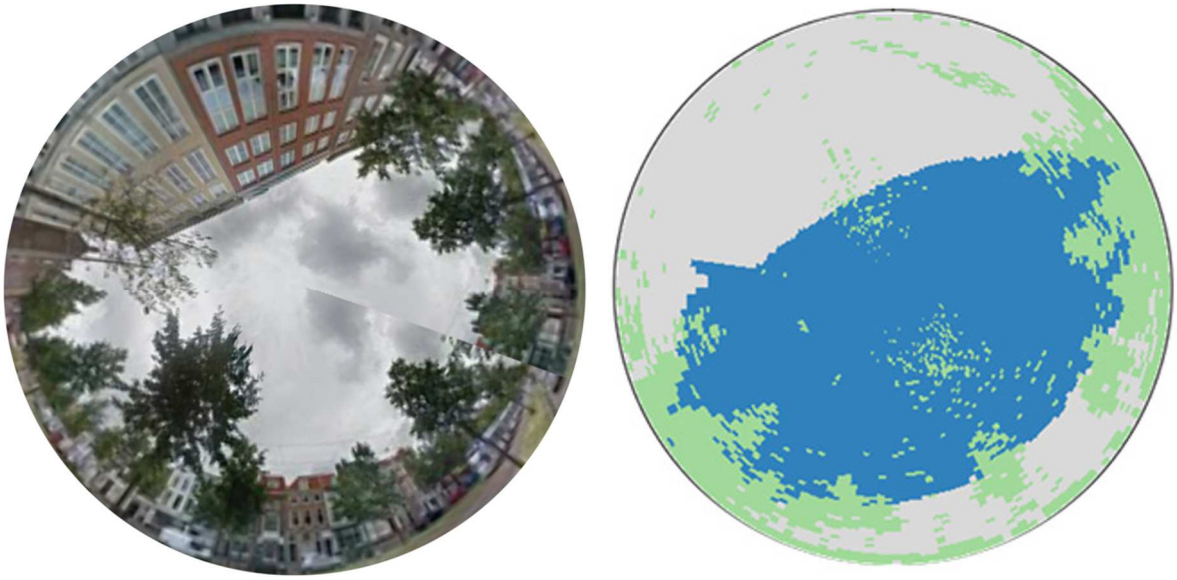
Source: Left - Google Maps, Right - Own work

4. Discussion

The final product of this Synthesis Project is an open-source web application that allows SVF calculation in a fast and quite accurate way.

The visual result can be considered of good quality. The image plot represents the view of the sky that a person would have on a certain location. However, since the aim was for the user to have an intuitive perspective on the result, the image was oriented according to the map, and not as a true fisheye camera product. When a street view image is oriented according to the map, the result is as shown in Figure 4.1. In this figure, the two images are compared with each other and the geometrical similarities are evident.

Figure 4.1 - Side by side comparison with fisheye plot



Source: Left - notlion.github.io, Right - Own work

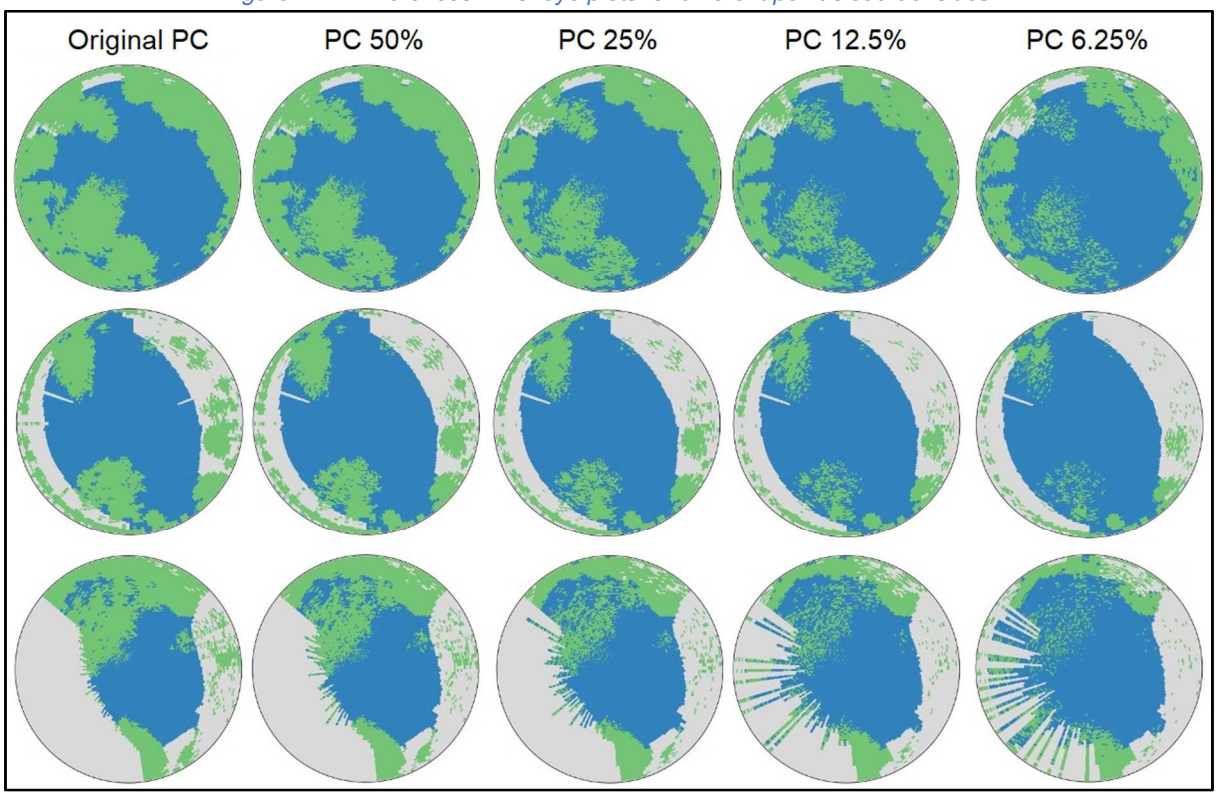
While the results are fast and accurate, there are still minor drawbacks that can be addressed in order to have an optimized and fully-scaled application with an interface as user-friendly as possible.

One of the main issues for this application is the storage and processing of massive point cloud data. For example, for the municipality of the Hague, we need to store and process over 60GB of data to achieve a satisfying level of accuracy. Processing has already been improved significantly during development by constantly optimizing the programming process. However, the storage problem still remains unsolved. For a fully-operational server that serves one municipality, memory size will most likely not be an issue. For a fully-operational server that would be scaled to e.g. serve an entire country, this may be considered a significant hindrance. One option to solve this issue would be to access the necessary part of the point cloud from PDOK through a web service, instead of storing and tiling up entire datasets. Unfortunately,

web point cloud services have not been fully developed yet and this option may not be implemented in the near future.

Another option to solve the storage issue would be to lower the density of the point cloud that needs to be processed. This option produces suboptimal results however, there is a linear relationship between yielded result accuracy and the density of the used point cloud. The deterioration of the quality of the plots, based on different point cloud densities can be seen in figure 4.2.

Figure 4.2 - Differences in fisheye plots for different point cloud densities



Source: Own work

Furthermore, the terrain of the study area is considered to be relatively flat. In other words, there are no ground features such as hills or dikes that could possibly obstruct sky view. Due to this property of the Dutch terrain, the ground points are not taken into consideration for the computation of SVF. The main argument behind the exclusion of ground points from the main process was deterioration in computational speed due to the larger number of points to be processed. Still, this assumption holds true for any typical terrain in the Netherlands, but not for countries with more complex topography. This problem will need to be addressed in future versions in our application, especially if it is going to be used in other countries. A solution to this would be to include ground points not only for assigning elevation to the viewpoint, but also to the general processing of the point cloud for SVF calculation.

5. Conclusions & Recommendations

Conclusion

The goal of this technical report was to provide insight into the inner workings of the creation of a web application for Sky View Factor Calculation. The research question was as follows:

> *How can an open source web application that generates a SVF for any user-requested point(s) within the municipality of The Hague be developed, using massive point cloud data, while maintaining a reasonable processing speed and providing the highest accuracy as possible?*

Such an application can be created making use of the Python programming language in order to get the computational process with a high accuracy. In order to acquire a reasonable processing speed, a number of open source libraries have been used; one of them being Numpy, which greatly improved the speed of the computational process. To allow the interaction between the web application and the offline application a remote server and an open source software called XAMPP are used. The connecting element between the server and the python script is PHP. This would be the answer to the research question in short, however it is very general. A more detailed answer to each challenge that arose during the answering of this question is entailed in chapters one through four.

Future Developments

The application that is created is a very general web application that makes use of locally stored point cloud data. Due to this generality, the application can be used as a basis for the development of other web applications for study of the urban environment. For example, one possible alternative could be an application that calculates DOP values to evaluate the quality of GPS measurements. DOP values are highly dependent on the geometry of satellites visible from a GPS receiver at a given point in time. Even if one satellite is no longer visible due to obstructions - for example, in cases of urban canyons, the satellite geometry changes dramatically and so does the corresponding DOP. Since we have already developed a process to detect obstructions in the urban environment, integration of satellite data to this application can lead to an operational program. Satellite data could be data such as the ephemeris or any available information on satellite network orbits.

Another future development of the application can be to provide an online web application for editing and saving point clouds. This application already allows the editing of (parts of) point clouds, but only stores the resulting outcomes, since only this is relevant to the SVF application. It would be an interesting development to provide a web application that shows the resulting point cloud when a building is removed from the existing point cloud or a user-generated building is added to the existing point cloud. It would also be a possible development to show the user this

point cloud data in 3D on the application itself, this can be a huge strain on the server, but the data is already available.

The last possible development would be the scaling of the application. We strongly believe it is scalable enough to incorporate the whole extent of the country since the necessary data is already available. The only thing missing is the required infrastructure, mainly a larger functional server. Also, we would be very pleased if this application could be also used or even further developed by organizations outside the Netherlands because it is our sincere belief that our application addresses a very important issue usually posed in modern urban planning in an effective and efficient way.

References

- An, S. M., Kim, B. S., Lee, H. Y., Kim, C. H., Yi, C. Y., Eum, J. H., & Woo, J. H. (2014). Three□dimensional point cloud based sky view factor analysis in complex urban settings. International Journal of Climatology, 34(8), 2685-2701.
- Bourke, P. (2001). Computer Generated Angular Fisheye Projections. http://paulbourke.net/dome/fisheye/ (viewed at 23-5-2018)
- Brown, M. J., Grimmond, S., & Ratti, C. (2001). Comparison of methodologies for computing sky view factor in urban environments (No. LA-UR-01-4107). Los Alamos National Lab., NM (US).
- Gal T., Lindberg F, Unger J. (2009). Computing continuous sky-view factors using 3D urban raster and vector databases: comparison and application to urban climate. Theor. Appl. Climatol.
- Johnson, G. T., & Watson, I. D. (1984). The determination of view-factors in urban canyons. Journal of Climate and Applied Meteorology, 23(2), 329-335.
- Publieke Dienstverlening Op de Kaart (2018), https://www.pdok.nl/nl/ahn3-downloads, (visited at 23-5-2018)

Appendix A – Python script: Main code

```
import numpy as np
import laspy as lp
import json
import math
import os
import matplotlib.pyplot as plt
import time
import sys
from shapely.geometry import Polygon
from operator import itemgetter

# Determine tile containing point,
# based on whether point lies within tile's extent
def find_tile():
    for tile in tilelist:
        # tile name contains coordinates of bounding box
        new_tile = tile.strip(".las").split(",")

        tile_min_x = float(new_tile[2])
        tile_min_y = float(new_tile[3])
        tile_max_x = float(new_tile[4])
        tile_max_y = float(new_tile[5])
        if tile_min_x <= x <= tile_max_x and tile_min_y <= y <= tile_max_y:
            return int(new_tile[0]), int(new_tile[1])

# Search for the tiles that are adjacent to the initially selected tile
def find_tile_grid(row, col):
    iteration_list = [-1, 0, 1]
    tile_grid = []

    for i in iteration_list:
        for j in iteration_list:
            tilename_start = "{},{}".format((row + i), (col + j))
            for tile in tilelist:
                if tile.startswith(tilename_start):
                    tile_grid.append(tile)
                    break
    # tile_grid: list of 9 tiles
    return tile_grid

# determine height of viewpoint by sampling the ground points
# center: location of viewpoint
def getheight(tile_grid):
    center = np.array([x, y])
    pointheight = 0
    points_number = 0

    for tile in tile_grid:
        # read the .las file
        file_input = lp.file.File("{}{}{}".format(path, "/Tiles/", tile),
mode='r')

        # keep groundpoints satisfying ground_rules:
        # classification 2 for ground, inside las file
        # keep points within radius of 5 metres
        ground_rules = np.logical_and(
            file_input.raw_classification == 2,
```

```
            np.sqrt(np.sum((np.vstack((file_input.x,
file_input.y)).transpose() - center) ** 2, axis=1)) <= 1)
        build_rules = np.logical_and(
            file_input.raw_classification == 6,
            np.sqrt(np.sum((np.vstack((file_input.x,
file_input.y)).transpose() - center) ** 2, axis=1)) <= 1)

        ground_points = file_input.points[ground_rules]
        build_points = file_input.points[build_rules]

        # make array with heights of each point
        if ground_points.size > build_points.size:
            ground_point_heights =
np.array((ground_points['point']['Z'])).transpose()
        else:
            ground_point_heights =
np.array((build_points['point']['Z'])).transpose()

        if ground_point_heights.size > 0:
            pointheight += float(np.sum(ground_point_heights))
            points_number += ground_point_heights.size

    # get mean value of points' heights
    if points_number > 0:
        height = pointheight / points_number
        return height
    else:
        return 0


# function to get all points lying within range of the defined radius from
the viewpoint
def getPoints(tile_grid, radius, view_height):
    # Viewpoint
    center = np.array([x, y])

    # Gather points
    arraysX, arraysY, arraysZ = [], [], []  # list of arrays of X,Y,Z
coords
    arrayDistances = []  # Horizontal distances
    arrayClasses = []  # Classifications
    toBeAdded = []
    for tile in tile_grid:
        inFile = lp.file.File("{}{}{}".format(path, "/Tiles/", tile),
mode='r')
        coords = np.vstack((inFile.x, inFile.y)).transpose()
        elevation = inFile.z
        distances = np.sqrt(np.sum((coords - center)**2, axis=1))

        keep_points = np.logical_and(np.logical_and(np.logical_or(
            inFile.raw_classification == 1,
            inFile.raw_classification == 6),
            distances < radius),
            elevation >= view_height/1000)

        # Get coordinates
        arraysX.append(inFile.x[keep_points])
        arraysY.append(inFile.y[keep_points])
        arraysZ.append(inFile.z[keep_points])
        # Get distances
        arrayDistances.append(distances[keep_points])
```

```
        # Get classifications
        arrayClasses.append(inFile.raw_classification[keep_points])

    # Concatenate all information
    X, Y, Z = arraysX[0], arraysY[0], arraysZ[0]
    distances = arrayDistances[0]
    classes = arrayClasses[0]
    for arrayX, arrayY in zip(arraysX[1:], arraysY[1:]):
        X = np.hstack([X, arrayX])
        Y = np.hstack([Y, arrayY])
    for arrayZ in arraysZ[1:]:
        Z = np.hstack([Z, arrayZ])
    for arDist in arrayDistances[1:]:
        distances = np.hstack([distances, arDist])
    for arClass in arrayClasses[1:]:
        classes = np.hstack([classes, arClass])

    if removeLst:
        toBeRemoved = clearpoints(removeLst, boundRemoveLst, X, Y, classes,
6)

        X=np.delete(X,toBeRemoved)
        Y=np.delete(Y,toBeRemoved)
        Z=np.delete(Z,toBeRemoved)
        distances=np.delete(distances,toBeRemoved)
        classes=np.delete(classes,toBeRemoved)

    if addLst:
        toBeCleared = clearpoints(addLst, boundAddLst, X, Y, classes, 0)

        X=np.delete(X,toBeCleared)
        Y=np.delete(Y,toBeCleared)
        Z=np.delete(Z,toBeCleared)
        distances=np.delete(distances,toBeCleared)
        classes=np.delete(classes,toBeCleared)

        for i in range(len(boundAddLst)):
            Xarray, Yarray, Zarray, Carray = create_pc(boundAddLst[i][0],
boundAddLst[i][1], boundAddLst[i][2], boundAddLst[i][3], height[i])
            toBeAdded=clearpoints(addLst, boundAddLst, Xarray, Yarray,
Carray, 6)

            Xarray = Xarray[toBeAdded]
            Yarray = Yarray[toBeAdded]
            Zarray = Zarray[toBeAdded]
            Carray = Carray[toBeAdded]
            coordsNew = np.vstack((Xarray, Yarray)).transpose()
            distancesNew = np.sqrt(np.sum((coordsNew - center) ** 2,
axis=1))

            keepNewPoints = np.logical_and(
                distancesNew < radius,
                Zarray >= view_height / 1000)

            Xarray = Xarray[keepNewPoints]
            Yarray = Yarray[keepNewPoints]
            Zarray = Zarray[keepNewPoints]
            Carray = Carray[keepNewPoints]
            distancesNew = distancesNew[keepNewPoints]

            X = np.hstack([X, Xarray])
```

```
            Y = np.hstack([Y, Yarray])
            Z = np.hstack([Z, Zarray])
            classes = np.hstack([classes, Carray])
            distances = np.hstack([distances, distancesNew])

    return X, Y, Z, distances, classes

def clearpoints(geomLst, boundLst, X, Y, classes, theClass):
    toBeRemoved = []
    for i in range(len(boundLst)):
        x_min = boundLst[i][0]
        x_max = boundLst[i][1]
        y_min = boundLst[i][2]
        y_max = boundLst[i][3]

        if theClass != 0:
            incl = np.where(np.logical_and(np.logical_and(np.logical_and(X
> x_min, X < x_max),
                                                          np.logical_and(Y
> y_min, Y < y_max)),
                                           classes == theClass))
        else:
            incl = np.where(np.logical_and(np.logical_and(X > x_min, X <
x_max),
                                           np.logical_and(Y > y_min, Y <
y_max)))

        pts = np.stack((X[incl], Y[incl]), axis=-1)

        geom = geomLst[i]
        # Edges
        minY = np.fmin(geom[:, 1][:-1], geom[:, 1][1:])
        maxY = np.fmax(geom[:, 1][:-1], geom[:, 1][1:])
        maxX = np.fmax(geom[:, 0][:-1], geom[:, 0][1:])
        nom = geom[:, 0][1:] - geom[:, 0][:-1]
        denom = geom[:, 1][1:] - geom[:, 1][:-1]
        fraction = np.divide(nom, denom)
        curtime = time.clock()

        for i in range(len(pts)):
            pointInside = inside_polygon(pts[i], minY, maxY, maxX, geom,
fraction)
            if pointInside is not None:
                toBeRemoved.append(incl[0][i])
    return toBeRemoved

# Create dome
def createDome(X, Y, Z, dists, classes, view_height):
    # Initialize dome
    # Indices = (Azimuth, Elevation)
    dome = np.zeros((180, 90), dtype=int)
    domeDists = np.zeros((180, 90), dtype=int)

    if X.size > 0:
        # Azimuths
        dX, dY = X - x, Y - y
        azimuths = np.arctan2(dY, dX) * 180 / math.pi - 90
        azimuths[azimuths < 0] += 360

        # Elevations
        dZ = Z - view_height / 1000
```

```
        elevations = np.arctan2(dZ, dists) * 180 / math.pi

        # Shade sectors
        # Array with dome indices, distances & classifications
        data = np.stack((azimuths // 2, elevations // 1, dists, classes),
axis=-1)
        # Sort according to indices & classifications
        sortData = data[np.lexsort([data[:, 2], data[:, 1], data[:, 0]])]

        # Spot where azimuth & elevation values change
        azimuth_change = sortData[:, 0][:-1] != sortData[:, 0][1:]
        elevation_change = sortData[:, 1][:-1] != sortData[:, 1][1:]
        keep = np.where(np.logical_or(azimuth_change, elevation_change))
        # Take position of next element, plus add first row
        shortestDistance = sortData[
            np.insert(keep[0] + 1, 0, 0)]  # (inserts second element of
change, first position, index of first point)
        # Define indices & classifications
        hor = shortestDistance[:, 0].astype(int)
        ver = shortestDistance[:, 1].astype(int)
        classif = shortestDistance[:, 3].astype(int)
        dists = shortestDistance[:, 2]

        # Update dome
        dome[hor, ver] = classif

        domeDists[hor, ver] = dists

        # Buildings as solids

        # Find building positions in dome
        # print dome[dome == 6].size
        if dome[dome == 6].size > 0:
            bhor, bver = np.where(dome == 6)
            # Create an array out of them
            builds = np.stack((bhor, bver), axis=-1)
            shape = (builds.shape[0] + 1, builds.shape[1])
            builds = np.append(builds, (bhor[0], bver[0])).reshape(shape)

            # Spot azimuth changes
            azimuth_change = builds[:, 0][:-1] != builds[:, 0][1:]
            keep = np.where(azimuth_change)
            # keep = np.insert(np.where(azimuth_change==True), 0, 0)
            # Change to building up to roof for each row
            roof_rows, roof_cols = builds[keep][:, 0], builds[keep][:, 1]
            for roof_row, roof_col in zip(roof_rows, roof_cols):
                condition = np.where(np.logical_or(domeDists[roof_row,
:roof_col] > domeDists[roof_row, roof_col],
                                                   dome[roof_row,
:roof_col] == 0))
                dome[roof_row, :roof_col][condition] = 6
    plot(dome)
    return dome

# Plot dome
def plot(dome):
    # Create circular grid
    theta, radius = np.mgrid[0:(2*np.pi+2*np.pi/180):2*np.pi/180, 0:90:1]
    Z = dome.copy().astype(float)
    Z = Z[0:, ::-1]  # Reverse array rows
    # assign colors depending on class
```

```
    Z[Z == 0] = 0
    Z[Z == 1] = 0.5
    Z[Z == 6] = 1

    if Z[Z == 6].size == 0:
        Z[0,0] = 1
    axes = plt.subplot(111, projection='polar')

    cmap = plt.get_cmap('tab20c')


    axes.pcolormesh(theta, radius, Z, cmap=cmap)
    axes.set_ylim([0, 90])
    axes.tick_params(labelleft=False)
    axes.set_theta_zero_location("N")
    plt.savefig("Plots/"+'{}_point{}.png'.format(filename[8:-5],fid))
    plt.close()

# calculate SVF, and percentage of building/vegetation obstructions
def calculate_SVF(radius, dome):
    obstructedArea = 0
    treeObstruction = 0
    buildObstruction = 0
    for i in range(0, 180):
        for j in range(0, 90):
            if dome[i, j] != 0:
                v = 90 - (j + 1)
                R = math.cos(v * math.pi / 180) * radius
                r = math.cos((v + 1) * math.pi / 180) * radius
                # calculate area of each obstructed sector (circular sector
area calculation)
                cell_area = (math.pi / 180.0) * (R ** 2 - r ** 2)
                obstructedArea += cell_area

                if dome[i, j] == 1:
                    treeObstruction += cell_area
                elif dome[i, j] == 6:
                    buildObstruction += cell_area

    circleArea = math.pi * (radius ** 2)

    # SVF: proportion of open area to total area
    SVF = (circleArea - obstructedArea) / circleArea
    treeObstructionPercentage = treeObstruction / circleArea
    buildObstructionPercentage = buildObstruction / circleArea
    return SVF, treeObstructionPercentage, buildObstructionPercentage

def integer(geom):
    geometry = []
    append = geometry.append
    for point in geom:
        x, y = point[0], point[1]
        append([x, y])
    geometry = np.array(geometry) #Closed polygon
    return geometry


def inside_polygon(pt, minY, maxY, maxX, geom, fraction):
    condition1 = np.logical_and(pt[1] > minY, pt[1] <= maxY)
    condition2 = pt[0] <= maxX
    condition = np.logical_and(condition1, condition2)
```

```
    intersX = geom[:, 0][:-1][condition] + (pt[1] - geom[:, 1][:-
1][condition]) * fraction[condition]
    truth = np.logical_or(geom[:, 0][:-1][condition] == geom[:,
0][1:][condition],
                          pt[0] <= intersX)

    intersections = truth[truth == True].size

    if intersections % 2 == 1:
        return pt

def create_pc(Xmin, Xmax, Ymin, Ymax, height, density=0.5):
    dx = Xmax - Xmin
    dy = Ymax - Ymin
    # print dx, dy
    Xadd=[]
    Yadd=[]
    Zadd=[]
    Cadd=[]
    # print int(math.ceil(dx/density))
    for i in range(int(math.ceil(dx/density))):
        for j in range(int(math.ceil(dy/density))):
            Xadd.append(Xmin+i*density)
            Yadd.append(Ymin+j*density)
            Zadd.append(float(height))
            Cadd.append(6)
    return np.asarray(Xadd), np.asarray(Yadd), np.asarray(Zadd),
np.asarray(Cadd)

def run():
    start = time.clock()
    row, col = find_tile()
    tile_grid = find_tile_grid(row, col)
    view_height = getheight(tile_grid)
    X, Y, Z, distances, classes = getPoints(tile_grid, radius, view_height)
    dome = createDome(X, Y, Z, distances, classes, view_height)

    SVF, tree_percentage, build_percentage = calculate_SVF(radius, dome)
    SVF, tree_percentage, build_percentage = round(SVF*100),
round(tree_percentage*100), round(build_percentage*100)
    print ('{}%'.format(int(SVF)) + "\n" +
'{}%'.format(int(tree_percentage)) + "\n" +
'{}%'.format(int(build_percentage)))
    end = time.clock()
    duration = end-start

if __name__ == '__main__':
    """GLOBAL VARIABLES"""
    # path for tile directory and list of tilenames
    filename=str(sys.argv[1])
    # filename = 'filelog/'+'testtest'+'.json'
    #path for tile directory and list of tilenames
    path = os.getcwd()
    tilelist = os.listdir(path+"/Tiles")
    # define radius
    radius = 100
    bufferRadius=1.5
    """END GLOBAL VARIABLES"""

    datadict = json.loads(open(filename).read())
```

```
nr_remove_polygons= len(datadict['polygonRemove'])
nr_add_polygons=len(datadict['polygonAdd'])

m = 1
removeLst = []
boundRemoveLst=[]
addLst = []
boundAddLst = []
height = []
while True:
    try:
        coorNum = len(datadict['polygonRemove'][m]['value'][0])
        coorLst = []
        for i in range(coorNum):
            x= float(datadict['polygonRemove'][m]['value'][0][i][0])
            y= float(datadict['polygonRemove'][m]['value'][0][i][1])
            coorLst.append((x,y))
        tempPolygon = Polygon(coorLst)

        bufferPolygon = tempPolygon.buffer(bufferRadius, resolution=2)
        geom=integer(list(zip(*bufferPolygon.exterior.coords.xy)))
        x_min, x_max = np.amin(geom[:,0]), np.amax(geom[:,0])
        y_min, y_max = np.amin(geom[:,1]), np.amax(geom[:,1])
        boundRemoveLst.append([x_min, x_max,y_min, y_max])
        removeLst.append(geom)
        # define next point
        m += 1
    except:
        break

n = 1
while True:
    try:
        coorNum = len(datadict['polygonAdd'][n]['value'][0])
        height.append(datadict['polygonAdd'][n]['value'][1])
        # define next point
        coorLst = []
        for i in range(coorNum):
            x = float(datadict['polygonAdd'][n]['value'][0][i][0])
            y = float(datadict['polygonAdd'][n]['value'][0][i][1])
            coorLst.append((x, y))

        X_min = min(coorLst, key=lambda item: item[0])[0]
        X_max = max(coorLst, key=lambda item: item[0])[0]
        Y_min = min(coorLst, key=lambda item: item[1])[1]
        Y_max = max(coorLst, key=lambda item: item[1])[1]

        boundAddLst.append([X_min, X_max, Y_min, Y_max])
        addLst.append(np.asarray(coorLst))

        n += 1
    except:
        break


o=1
while True:
    try:
        x = float(datadict['coordinates'][o]['value'][0])
        y = float(datadict['coordinates'][o]['value'][1])
        fid = int(datadict['coordinates'][o]['key'])
```

```
        # define viewpoint
        run()

        # define next point
        o += 1
    except:
        break
```

Appendix B – HTML

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<title>Application – Urban Horizon</title>
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet"
href="https://openlayers.org/en/v4.6.5/css/ol.css" type="text/css">
    <link rel="stylesheet" href="https://www.w3schools.com/w3css/4/w3.css">
    <link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Lato">
    <link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Montserrat">
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-
awesome.min.css">
    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.0.13/css/all.css"
integrity="sha384-
DNOHZ68U8hZfKXOrtjWvjxusGo9WQnrNx2sqG0tfsghAvtVlRW3tvkXWZh58N9jp"
crossorigin="anonymous">
    <link rel="icon" type="image/ico" href="Images/favicon.ico">
    <style type="text/css">

        body {
            margin: 1px;
        }

        #map {
            position: absolute;
            top: 92px;
            width: calc(100% – 740px);
            height: calc(100% – 93px);
            border: 1px solid black;
            left: 290px;
            min-width: 400px;
        }

        #olattribution {
            position: absolute;
            margin: 0px 15px;
            bottom: 0px;
            z-index: 1;
        }

        #banner {
            position: absolute;
            top: 100%;
            height: 40px;
            background-color: #9fd0f9;
            width: 100%;
        }

        #toolbar_1 {
            position: absolute;
            top: 10%;
            left: 290px;
        }

        #toolbar_2 {
```

```
    position: absolute;
    top: 10%;
    left: 0.1%;
}

#toolbar_map {
    position: absolute;
    margin: 5px 5px;
    right: 0px;
}

#CC {
    position: absolute;
    margin: 5px 5px;
    right: 0px;
    bottom: 0px;
    z-index: 1;
}

#SVFimage{
    position: absolute;
    top: 86px;
    border: none;
    right: 0px;
    z-index: -1;
}

#loader {
    position: absolute;
    bottom: 0px;
    left: 0px;
}

#legend {
    position: absolute;
    top: 525px;
    left: calc(100% - 450px);
    z-index: -1;
    display: none;
}

button {
    position:relative;
    height: 30px;
    width:25px;
    background-color: #2196F3;
    border: none;
    border-radius: 5px;
    color: white;
    padding: 5px 5px;
    font-size: 16px;
    cursor: pointer;
    z-index: 1;
}

a.fa {
    text-decoration: none;
    position:relative;
    top: 1px;
    height: 30px;
    width:25px;
```

```
    background-color: #2196F3;
    border: none;
    border-radius: 5px;
    color: white;
    padding: 7px 5px;
    font-size: 16px;
    cursor: pointer;
    z-index: 1;
}

#CC-CC, #CC-BY, #CC-NC {
    text-decoration: none;
    color: #085391;
}

#togglelayer, #togglepoly {
    right: -30px;
    padding: 0px 0px;
}

#togglezoom {
    top: 36px;
    padding: 0px 0px;
}

input {
    position:absolute;
    height: 0px;
    width:0px;
    z-index: -1;
    opacity: 0;
}

button:hover, a.fa:hover {
    background-color: #085391;
}

#contain {
    width: calc(100% - 740px);
    height: calc(100% - 93px);
}

#start:focus, #select:focus {
    background-color: black;
}

#myTableSpace {
    position: absolute;
    display: block;
    top: 92px;
    width: 288px;
    height: calc(100% - 93px);
    overflow-y: auto;
}

#myTable {
    width: 100%;
    border: 1px;
    top:92px;
    text-align: center;
    border-collapse: collapse;
```

```
        }

        #percentage {
            position: absolute;
            top: 614px;
            left: calc(100% - 449px);
            z-index: -1;
        }

        #percentageTB{
            width: 449px;
            float: left;
            border: none;
            text-align: center;
            border-collapse: collapse;
            display: none;
        }

        th {
            border: 1px;
            border-style: solid;
            border-color: #085391;
            background-color: #2196F3;
            color: #ffffff;
        }

        td {
            border: 1px;
            border-style: solid;
            border-color: #085391;
        }

        tr:hover {
            background-color: #9fd0f9;
        }

        tr:active {
            background-color: #cfe8fc;
        }

        html {
            display:block;
            overflow:auto;
        }

        #transparent {
            opacity: 0.0;
            position:absolute;
            width: 99%;
            height: 99%;
            background: rgba(100, 100, 0, 0.8);
            z-index: 1;
        }

    </style>

    <script src="https://openlayers.org/en/v4.6.5/build/ol.js"></script>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></sc
ript>
```

```
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/proj4js/2.3.15/proj4.js"></scri
pt>
    <script src="https://epsg.io/28992.js"></script>
    <script src="papaparse.min.js"></script>
</head>
<body>
    <!-- Create Container so map and mask (that disables double clicks)
overlay -->
    <div id="contain">
        <div id="map" class="map">
            <div id="olattribution"></div>
            <div id="toolbar_map">
                <button id='togglelayer' class="fas fa-map" title="Toggle
Background" onclick ="togglelayer()"></button>
                <button id='togglepoly' class="fas fa-home" title="Toggle
Building Footprints" onclick ="togglepolygons()"></button>
                <button id="togglezoom" class="fas fa-crosshairs"
title="Toggle Zoom on Selection" onclick="togglezoom()"></button>
            </div>
            <div id="CC">
                <a id='CC-CC' class="fab fa-creative-commons w3-xlarge"
href="http://creativecommons.org/licenses/by-nc/4.0/"></a>
                <a id='CC-BY' class="fab fa-creative-commons-by w3-xlarge"
href="http://creativecommons.org/licenses/by-nc/4.0/"></a>
                <a id='CC-NC' class="fab fa-creative-commons-nc w3-xlarge"
href="http://creativecommons.org/licenses/by-nc/4.0/"></a>
            </div>
        </div>
        <img id="transparent" src="transparent.png" style='display: none;'
align=middle />
    </div>
    <div id="SVFimage"></div>
    <img id="legend" src="legend.png" style='display: none;'/>
    <div id="percentage">
        <!-- Table result with percentages of sky -->
        <table id= "percentageTB">
            <tr>
               <td></td>
               <th>Unobstructed Vision</th>
               <th colspan="2">Obstructed Vision</th>
            </tr>
            <tr>
               <th>ID</th>
               <th>Sky View Factor</th>
               <th>Vegetation</th>
               <th>Building</th>
            </tr>

            <!-- Need this, introduces new row to table for number results
-->
            <tr id="contentRow">
               <td></td>
               <td></td>
               <td></td>
               <td></td>
            </tr>
        </table>
    </div>
    <!-- Table containing basic information on points (coordinates, SVF) --
>
```

```
    <div id="myTableSpace">
        <table id="myTable" >
            <tr>
                <th>ID</th>
                <th>X</th>
                <th>Y</th>
                <th>SVF</th>
            </tr>
        </table>
    </div>
    <!-- Gif appearing during loading time of calculation process -->
    <img id="loader" src="reload.gif" width='288px' height='288px'
style='display: none;'>
    <!-- Navbar -->
    <div class="w3-top">

        <div class="w3-bar w3-blue w3-card w3-left-align w3-large">
            <a class="w3-bar-item w3-button w3-hide-medium w3-hide-large
w3-right w3-padding-large w3-hover-white w3-large w3-blue"
href="javascript:void(0);" onclick="myFunction()" title="Toggle Navigation
Menu"><i class="fa fa-bars"></i></a>
            <a href="Home.html" class="w3-bar-item w3-button w3-padding-
large w3-hover-white">Home</a>
            <a href="SVF.html" class="w3-bar-item w3-button w3-hide-small
w3-padding-large w3-white">Application</a>
            <a href="Tutorial.html" class="w3-bar-item w3-button w3-hide-
small w3-padding-large w3-hover-white">Tutorial</a>
            <a href="Documentation.html" class="w3-bar-item w3-button w3-
hide-small w3-padding-large w3-hover-white">Documentation</a>
        </div>
        <!-- Navbar on small screens/ Icon Appears on top-right with drop
down list -->
        <div id="navDemo" class="w3-bar-block w3-light-blue w3-hide w3-
hide-large w3-hide-medium w3-large">
            <a href="SVF.html" class="w3-bar-item w3-button w3-padding-
large">Application</a>
            <a href="Tutorial.html" class="w3-bar-item w3-button w3-
padding-large">Tutorial</a>
            <a href="Documentation.html" class="w3-bar-item w3-button w3-
padding-large">Documentation</a>
        </div>

        <!-- Bar containing function buttons -->
        <div id="banner" class="w3-top">
            <div id="toolbar_1" class="btn-group">
                <button id="start" class="fas fa-pencil-alt" title="Draw
Points"></button>
                <button id="stop" class="fas fa-ban" title="Stop
Drawing"></button>
                <button id="select" class="fas fa-mouse-pointer"
title="Select Point"></button>
                <button id="delete" class="fas fa-times" title="Delete
Selected Point"></button>
                <button id="removebuild" class="far fa-eye-slash"
title="Ignore Building"></button>
                <button id="addbuild" class="fas fa-plus" title="Add New
Building"></button>
                <button id="undo" class="fas fa-undo" title="Undo
Calculation"></button>
                <button id="clear" class="fas fa-eraser " title="Clear
Canvas"></button>
```

```
                <button id="SVF" class="fas fa-calculator "
title="Calculate SVF" ></button>
            </div>
            <div id=toolbar_2>
                <input type="file" id="files"  class="form-control"
accept=".csv" required />
                <button type="submit" id="submit-file" class="fa fa-upload"
title="Upload Points"></button>
                <a href="#" id="xx" class="fa fa-download" title="Export
Table to Excel"></a>
            </div>
        </div>
    </div>
</div>
</body>
<script>
    // Retrieve data from OpenLayer Map layer
    var source = new ol.source.Vector({wrapX: false});
    var source1 = new ol.source.Vector({wrapX: false});
    var source2 = new ol.source.Vector({wrapX: false});
    var vectorLayer = new ol.layer.Vector({source: source});
    var vectorLayer1 = new ol.layer.Vector({source: source1});
    var vectorLayer2 = new ol.layer.Vector({source: source2});
    // Initialize variables
    var draw; // global so we can remove it later
    var featureID = 0;
    var singleClick;
    var selectedFeatureID = 0;
    var ccord;
    var flag = 0;
    var removeNum = 0;
    var buildingPercent = {};
    var treePercent = {};
    var currentDate = new Date();
    var userID = currentDate.getTime();
    var deleted = 0;
    var selectedFeature;
    var stopremovevar = true;
    var stopselectvar = true;
    var zoomselectvar = true;
    // Set initial view position
    var view =  new ol.View({
                center: centerWebMercator,
                zoom: 17,
                });
    var viewProjection = view.getProjection();
    var viewResolution = view.getResolution();
    var container = document.getElementById('information')


    var databuild = [{"key":"0"}];
    var bcnt = 1;
    var bcnt_p = 1;
    var addedbuild = [{"key":"0"}];



    // Inserted Points Style Function
    function styleFunction(description) {
        return [
            new ol.style.Style({
```

```
                image: new ol.style.Circle({
                    fill: new ol.style.Fill({
                    color: '#2196F3'
                    }),
                    stroke: new ol.style.Stroke({
                        color: '#ffffff',
                        width: 1
                    }),
                    radius: 11
                }),

                text: new ol.style.Text({
                    font: '16px Calibri,sans-serif',
                    fill: new ol.style.Fill({ color: '#ffffff' }),
                    stroke: new ol.style.Stroke({
                        color: '#ffffff',
                        width: 1
                    }),
                // get the text from the feature - `this` is ol.Feature
                text: description
                })
            })
        ];
    }
    // Selected Point Style function
    function styleFunctionSelect(description) {
        return [
            new ol.style.Style({
                image: new ol.style.Circle({
                    fill: new ol.style.Fill({
                        color: '#085391'
                    }),
                    stroke: new ol.style.Stroke({
                        color: '#ffffff',
                        width: 2
                    }),
                    radius: 15
                }),

                text: new ol.style.Text({
                    font: '22px Calibri,sans-serif',
                    fill: new ol.style.Fill({ color: '#ffffff' }),
                    stroke: new ol.style.Stroke({
                        color: '#ffffff', width: 1
                    }),
                    // get the text from the feature - `this` is ol.Feature
                    text: description
                })
            })
        ];
    }

    // Topographic map layer
    var baselayer = new ol.layer.Tile({
        source: new ol.source.OSM()
    });
    baselayer.setVisible(true);

    // Layer of Airplane Captured photos
    var satlayer = new ol.layer.Tile({
        source: new ol.source.TileWMS({
```

```
            url:
'https://geodata.nationaalgeoregister.nl/luchtfoto/rgb/wms',
            params: {'LAYERS': 'Actueel_ortho25', 'FORMAT':'image/png'},
            minZoom: 100,
            maxZoom: 200
        })
    });
    satlayer.setVisible(false);
    // Building footprint images retrieved from PDOK
    var polylayer = new ol.layer.Tile({
        source: new ol.source.TileWMS({
            url: 'https://geodata.nationaalgeoregister.nl/bag/wms',
            params: {'LAYERS': 'pand', 'FORMAT':'image/png'},
            opacity: 0.0
        })
    });

    // Response from request about the Hague's municipality borders
    var boundarylayer = new ol.layer.Tile({
        source: new ol.source.TileWMS({
            url:
'https://geodata.nationaalgeoregister.nl/bestuurlijkegrenzen/wfs?&TYPENAME=
bestuurlijkegrenzen:gemeenten&SRSNAME=EPSG:3857&cql_filter=(bestuurlijkegre
nzen:code=%270518%27)',
            params: {'LAYERS': 'gemeenten', 'FORMAT': 'image/png'}
        })
    });

    var layers = [baselayer, satlayer, polylayer,boundarylayer,
vectorLayer1, vectorLayer2, vectorLayer]


    // Initialization of map focus
    var centerLonLat = [4.30046, 52.07455]; //This is for The Hague, use
zoomlevel 12
    // var centerLonLat = [4.373003, 52.0006821];
    var centerWebMercator = ol.proj.fromLonLat(centerLonLat);

    var map = new ol.Map({
        layers: layers,
        target: 'map',
        controls: ol.control.defaults({
            attributionOptions: {
                target: document.getElementById('olattribution')
            }
        }),
        view: new ol.View({
            center: centerWebMercator,
            zoom: 12
        })
    });

    // "Start drawing Points" button function
    $("button#start").on("click", function (event) {
        // Disable "stopremovevar" and "stopselectvar" states
        stopremovevar = true;
        stopselectvar = true;
        map.removeInteraction(draw);
        map.removeInteraction(singleClick);
        addInteraction();
    });
```

```
    // "Stop creating Points" function
    $("button#stop").on("click", function (event) {
        stopremovevar = true;
        stopselectvar = true;
        map.removeInteraction(singleClick);
        map.removeInteraction(draw);
    });

    // "Select Point" function
    $("button#select").on("click", function (event) {
        stopselectvar = false;
        stopremovevar = true;
        map.removeInteraction(draw);
        select();
    });

    // "Point Deletion" function
    $("button#delete").on("click", function (event) {
        map.removeInteraction(draw);
        stopremovevar = true;
        stopselectvar = true;
        remove();
    });

    // "Remove Calculation Results" Button function
    $("button#undo").on("click", function (event) {
        map.removeInteraction(draw);
        stopremovevar = true;
        stopselectvar = true;
        undo();
    });

    // Button Function to initialize "Building Removal" function
    $("button#removebuild").on("click", function (event) {
        map.removeInteraction(draw);
        stopremovevar = false;
        stopselectvar = true;
        removebuild();
    });

    // Button Function to initialize "Building Addition" function
    $("button#addbuild").on("click", function (event) {
        map.removeInteraction(draw);
        stopremovevar = true;
        stopselectvar = true;
        addPoly();
    });

    // Button function to initialize "Clear Canvas" function
    $("button#clear").on("click", function (event) {
        map.removeInteraction(singleClick);
        map.removeInteraction(draw);
        stopremovevar = true;
        stopselectvar = true;
        clearCanvas();
    });

    // Button function to initiate the SVF calculation
    $("button#SVF").on("click", function (event) {
        stopremovevar = true;
```

```
        stopselectvar = true;
        sendData()
    });

    // Function to make Image Plot and Percentages Table appear on point
selection from table
    $('#myTable').on('click', 'tr', function(){
        var table = document.getElementById("myTable");
        var row = table.rows[selectedFeatureID];
        if (selectedFeatureID > 0){

selectedFeature.setStyle(styleFunction(selectedFeatureID.toString()));
            row.style.backgroundColor = '#ffffff'
        };
        currentIndex = $(this).index();
        if (currentIndex != 0){
            selectedFeatureID = currentIndex;
        }
        if (selectedFeatureID <= flag){
            var features = source.getFeatures();
            if (features != null && features.length > 0) {
                for (x in features) {
                    var properties = features[x].getProperties();
                    var id = properties.id;
                    if (id == selectedFeatureID) {
                        $("#legend").show();
                        $("#percentageTB").show();
                        // Create percentage table
                        var cellPercentege =
document.getElementById("percentageTB").rows.namedItem("contentRow").cells;
                        cellPercentege[0].innerHTML = selectedFeatureID;
                        cellPercentege[1].innerHTML =
table.rows[selectedFeatureID].cells[3].innerHTML;
                        cellPercentege[2].innerHTML =
treePercent[selectedFeatureID];
                        cellPercentege[3].innerHTML =
buildingPercent[selectedFeatureID];
                        // Retrieve SVF plot corresponding to point
                        var imgstr = '../SynthesisProject/Plots/' + userID
+ '_' + 'point' + id + '.png';
                        document.getElementById("SVFimage").innerHTML =
'<img id="plotimg" src="'+ imgstr +'"/>';
                        break;
                    }
                }
            }
        }
        var features = source.getFeatures();
        if (features != null && features.length > 0) {
            for (x in features) {
                var properties = features[x].getProperties();
                var id = properties.id;
                if (id == selectedFeatureID) {
                    selectedFeature = features[x];
                    if (zoomselectvar == true) {
                        var ext =
selectedFeature.getGeometry().getCoordinates();
                        map.getView().animate({center:ext,zoom:18});
                    }
                    break;
                }
```

```
            }
        }


selectedFeature.setStyle(styleFunctionSelect(selectedFeatureID.toString()))

        row = table.rows[selectedFeatureID];
        row.style.backgroundColor = '#cfe8fc'
    });

    // Function about creating points on map canvas
    var addInteraction = function () {
        draw = new ol.interaction.Draw({
            source: source,
            type: "Point"
        });

        map.addInteraction(draw); //allows drawing point on map

        // Nested function initialized when clicking on the map layer
        draw.on('drawend', function (event) {
            // Bring transparent image on the front, making user unable of
clicking on the map for set time
            $("#transparent").show();
            setTimeout(function removeMask() {
                $("#transparent").hide()
                }, 200); //wait 0.2 seconds to re-enable clicking
            featureID = featureID + 1;
            // Inserts Point information on table
            var table = document.getElementById("myTable");
            var row = table.insertRow(featureID);
            row.id = featureID;
            var cell1 = row.insertCell(0);
            var cell2 = row.insertCell(1);
            var cell3 = row.insertCell(2);
            cell1.innerHTML = featureID;
            var coords =
ol.proj.transform(event.feature.getGeometry().getCoordinates(),
'EPSG:3857', 'EPSG:28992');
            cell2.innerHTML = coords[0].toFixed(3);
            cell3.innerHTML = coords[1].toFixed(3);
            event.feature.setProperties({
               'id': featureID
            })
            event.feature.setStyle(styleFunction(featureID.toString()))
        })
    };

    // Function to make Image Plot and Percentages Table appear on point
selection from map canvas
    var select = function () {
        singleClick = new ol.interaction.Select({
            layers: [vectorLayer]
        });
        map.addInteraction(singleClick);
        singleClick.getFeatures().on('add', function(event){
            if (stopselectvar == true) {
                map.removeInteraction(singleClick);
                return;
            };
            var properties = event.element.getProperties();
```

```
            var table = document.getElementById("myTable");
            var row = table.rows[selectedFeatureID];
            if (selectedFeatureID !== 0){

selectedFeature.setStyle(styleFunction(selectedFeatureID.toString()));
                row.style.backgroundColor = '#ffffff'
            };
            selectedFeatureID = properties.id;
            selectedFeature = event.element;

selectedFeature.setStyle(styleFunctionSelect(selectedFeatureID.toString()))
            row = table.rows[selectedFeatureID];
            row.style.backgroundColor = '#cfe8fc';
            if (selectedFeatureID <= flag){
                var features = source.getFeatures();
                if (features != null && features.length > 0) {
                    for (x in features) {
                        var properties = features[x].getProperties();
                        var id = properties.id;
                        if (id == selectedFeatureID) {
                            $("#legend").show();
                            $("#percentageTB").show();
                            var cellPercentege =
document.getElementById("percentageTB").rows.namedItem("contentRow").cells;
                            cellPercentege[0].innerHTML =
selectedFeatureID;
                            cellPercentege[1].innerHTML =
row.cells[3].innerHTML;
                            cellPercentege[2].innerHTML =
treePercent[selectedFeatureID];
                            cellPercentege[3].innerHTML =
buildingPercent[selectedFeatureID];
                            var imgstr = '../SynthesisProject/Plots/' +
userID + '_' + 'point' + id + '.png';
                            document.getElementById("SVFimage").innerHTML =
'<img id="plotimg" src="'+ imgstr +'"/>';
                            break;
                        }
                    }
                }
            }
        });
    };

    // Function on Removing a Building from Map Canvas
    var removebuild = function () {
        map.on('click', function(evt) {
            if (stopremovevar == true) {
                return;
            };
            // Get building feature by clicking on map canvas
            coord = ol.proj.transform(evt.coordinate, 'EPSG:3857',
'EPSG:28992')
            var url = polylayer.getSource().getGetFeatureInfoUrl(
                coord, viewResolution, 'EPSG:28992',
            {'INFO_FORMAT': 'application/json'});
            // Construct polygon to be sent as JSON request
            if (url) {
                var parser = new ol.format.GeoJSON();
                $.ajax({
                    url: url,
```

```
                    dataType: 'json',
                    jsonpCallback: 'parseResponse'
                }).then(function(response) {
                    var build_id =
(response['features']['0']['properties']['identificatie'])
                    var polygon =
(response['features']['0']['geometry']['coordinates']['0']['0'])
                    var geometry =
(response['features']['0']['geometry']['coordinates']['0'])
                    databuild.push({key: bcnt, value: [polygon,
build_id]});

                    polyCor = [];
                    polygon.forEach(function(geomCor){
                        polyCor.push(ol.proj.transform(geomCor,
'EPSG:28992', 'EPSG:3857'))
                    })
                    // Style deleted building with red color
                    var style = new ol.style.Style({
                        stroke: new ol.style.Stroke({color: 'grey', width:
1}),
                        fill: new ol.style.Fill({color: '#ff6666'})
                    })
                    // Define new polygon feature
                    var feature = new ol.Feature({
                        geometry: new ol.geom.Polygon([polyCor]),
                        id: bcnt,
                    })
                    bcnt = bcnt + 1;
                    source1.addFeature(feature);
                    feature.setStyle(style)
                });
            };
        });
    };

    // Function to add a new polygon
    var addPoly = function () {
        // Draw polygon on canvas
        draw = new ol.interaction.Draw({
            source: source2,
            type: "Polygon"
        });

        map.addInteraction(draw);
        draw.on('drawend', function (event) {
            var txt;
            // Set height of building
            var height = window.prompt("Please enter the height for the
building:","");
            while (height == null || height == "" || Number(height) <= 0 ||
isNaN(height)) {
                var height = prompt("The height is not valid! Please enter
the height for the building:", "");
            }
            var buildingHeight = Number(height);
            addedPolyCor =
event.feature.getGeometry().getCoordinates()['0'];
            var coordLst = [];
            addedPolyCor.forEach(function(addedPoly){
```

```
                coordLst.push(ol.proj.transform(addedPoly, 'EPSG:3857',
'EPSG:28992'))
            });
            addedbuild.push({key: bcnt_p, value:
[coordLst,buildingHeight]})
            bcnt_p = bcnt_p+1;
            console.log(addedbuild);

            var styleadd = new ol.style.Style({
                stroke: new ol.style.Stroke({color: 'grey', width: 1}),
                fill: new ol.style.Fill({color: '#99e699'})
            })
            event.feature.setStyle(styleadd)
        });
    };

    // Function to undo all calculations performed and remove all results
    var undo = function () {
        flag = 0;
        buildingPercent = {};
        treePercent = {};
        currentDate = new Date();
        userID = currentDate.getTime();
        deleted = 0;
        currentDate = new Date();
        userID = currentDate.getTime();
        document.getElementById("SVFimage").innerHTML = "";
        $("#legend").hide();
        $("#percentageTB").hide();
        var table = document.getElementById("myTable");
        var rows = table.getElementsByTagName("tr");
        for(i = 1; i < rows.length; i++){
            var currentRow = table.rows[i];
            currentRow.deleteCell(3)
        }
    }
    // Function that resets everything
    var clearCanvas = function(){
        flag = 0;
        featureID = 0;
        selectedFeatureID = 0;
        removeNum = 0;
        buildingPercent = {};
        treePercent = {};
        currentDate = new Date();
        userID = currentDate.getTime();
        deleted = 0;
        currentDate = new Date();
        userID = currentDate.getTime();
        stopremovevar = true;
        stopselectvar = true;
        databuild = [{"key":"0"}];
        addedbuild = [{"key":"0"}];
        document.getElementById("SVFimage").innerHTML = "";
        $("#legend").hide();
        $("#percentageTB").hide();
        var table = document.getElementById("myTable");
        var rows = table.getElementsByTagName("tr");
        while (rows.length > 1){
            document.getElementById("myTable").deleteRow(1);
        rows = table.getElementsByTagName("tr");}
```

```
        var features = source.getFeatures();
        features.forEach(function(feature){
            source.removeFeature(feature);
        })
        var features = source1.getFeatures();
        features.forEach(function(feature){
            source1.removeFeature(feature);
        })
        var features = source2.getFeatures();
        features.forEach(function(feature){
            source2.removeFeature(feature);
        })
    }

    // Function that presents the SVF plot to the user
    var showPlot = function (){
        var features = source.getFeatures();
        if (features != null && features.length > 0) {
            for (x in features) {
                var properties = features[x].getProperties();
                var id = properties.id;
                if (id == selectedFeatureID) {
                    var imgstr = '../SynthesisProject/Plots/' + userID +
'_' + 'point' + id + '.png';
                    document.getElementById("SVFimage").innerHTML = '<img
id="plotimg" src="'+ imgstr +'"/>';
                    break;
                }
            }
        }
    }

    // Function to delete selected existing point(s)
    var remove = function () {
        var features = source.getFeatures();
        if (features != null && features.length > 0) {
            for (x in features) {
                var properties = features[x].getProperties();
                var id = properties.id;
                if (id == selectedFeatureID) {
                    var cell =
document.getElementById("myTable").rows[id].cells;
                    cell[0].innerHTML = '';
                    cell[1].innerHTML = '';
                    cell[2].innerHTML = '';
                    if (cell[3] !== undefined){
                        cell[3].innerHTML = '';}
                    if (flag > 0){removeNum = removeNum + 1;}
                    source.removeFeature(features[x]);
                    map.removeInteraction(singleClick);
                    break;
                }
            }
        }

        stopselectvar = false;
        stopremovevar = true;
        map.removeInteraction(draw);
        select();
    }
```

```
    // Function that toggles the canvas from topographic map to aerial
imagery
    var togglelayer = function() {
        if (baselayer.getVisible() == true) {
            baselayer.setVisible(false);
            satlayer.setVisible(true);
            polylayer.setOpacity(0.5);
        } else {
            baselayer.setVisible(true);
            satlayer.setVisible(false);
            polylayer.setOpacity(1);
        }
    };

    // Function to toggle polygon footprints on/off
    var togglepolygons = function() {
        if (polylayer.getVisible() == true) {
            polylayer.setVisible(false);
        } else {
            polylayer.setVisible(true);
        }
    };

    var togglezoom = function() {
        var target = document.getElementById("togglezoom");
        if (zoomselectvar == true) {
            zoomselectvar = false;
            target.style.opacity = "0.6";
        } else {
            zoomselectvar = true;
            target.style.opacity = "1.0"
        }
    };

    proj4.defs("EPSG:28992","+proj=sterea +lat_0=52.15616055555555
+lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel
+towgs84=565.417,50.3319,465.552,-0.398957,0.343988,-1.8774,4.0725 +units=m
+no_defs");

    // Function that constructs request
    var sendData = function () {
        var trackList = [];
        var features = source.getFeatures();
        var data = [{userID:userID}];
        features.forEach(function(feature){
            var properties = feature.getProperties();
            var id = properties.id;
            if (id > flag){
                trackList.push(id)
                var coord = feature.getGeometry().getCoordinates();
                coord = ol.proj.transform(coord, 'EPSG:3857',
'EPSG:28992');
                data.push({key: id, value: coord});
            }
        });
        var requestData = [data, databuild, addedbuild];
        // console.log(requestData)
        $.ajax({
            type: "POST",
            url: "responsePHP.php",
            data: {kvcArray : requestData},
```

```
            beforeSend: function(){
                // Show image container
                $("#loader").show();
                map.removeInteraction(draw);
            },
            success: function(data, textStatus) {
                var table = document.getElementById("myTable");
                var cnt = 0;
                var outputCnt = 0;
                var SVFs = data.split("\n");
                SVFs.forEach(function(svf){

                    outputCnt = outputCnt + 1;
                    if (outputCnt%3 == 1){
                        var row = $('table#myTable tr#'+trackList[cnt]);
                        row.append($("<td>"+svf.substring(0,5)+"</td>"));
                    }

                    else if (outputCnt%3 == 2) {
                        treePercent[trackList[cnt]] = svf;
                        //console.log(treePercent)
                    }

                    else if (outputCnt%3 == 0) {
                        buildingPercent[trackList[cnt]] = svf;
                        //console.log(buildingPercent)
                        cnt = cnt + 1;
                    }

                });
            },
            complete:function(data){
                // Hide image container
                $("#loader").hide();
                flag = featureID;
                map.removeInteraction(singleClick);
                addInteraction();
            }
        });

    };

    // Function to enable user to download the point table in CSV format
    function exportTableToCSV($table, filename) {
        var $rows = $table.find('tr:has(td),tr:has(th)');
        // Temporary delimiter characters unlikely to be typed by keyboard
        // This is to avoid accidentally splitting the actual contents
        tmpColDelim = String.fromCharCode(11), // vertical tab character
        tmpRowDelim = String.fromCharCode(0), // null character
        // actual delimiter characters for CSV format
        colDelim = '","',
        rowDelim = '"\r\n"',

        // Grab text from table into CSV formatted string
        csv = '"' + $rows.map(function (i, row) {
            var $row = $(row), $cols = $row.find('td,th');

            return $cols.map(function (j, col) {
                var $col = $(col), text = $col.text();
                return text.replace(/"/g, '""'); // escape double quotes
            }).get().join(tmpColDelim);
```

```
        }).get().join(tmpRowDelim)
        .split(tmpRowDelim).join(rowDelim)
        .split(tmpColDelim).join(colDelim) + '"',
        // Data URI
        csvData = 'data:application/csv;charset=utf-8,' +
encodeURIComponent(csv);
        //console.log(csvData);
        if (window.navigator.msSaveBlob) { // IE 10+
            //alert('IE' + csv);
            window.navigator.msSaveOrOpenBlob(new Blob([csv], {type:
"text/plain;charset=utf-8;"}), "csvname.csv")
        }
        else {
            $(this).attr({ 'download': filename, 'href': csvData, 'target':
'_blank' });
        }
    }

    // Button function to initiate CSV table download
    $("#xx").on('click', function (event) {
        exportTableToCSV.apply(this, [$('#myTable'), 'export.csv']);
        // IF CSV, don't do event.preventDefault() or return false
        // We actually need this to be a typical hyperlink
    });

    // Function to import point coordinates
    $('#submit-file').on("click",function(e){
        $( "#files" ).trigger( "click" );
        e.preventDefault();
        $( "#files" ).change(function(){ $('#files').parse({
            config: {
                delimiter: "auto",
                complete: displayHTMLTable,
            },
            before: function(file, inputElem){
            //console.log("Parsing file...", file);
            },
            error: function(err, file){
            //console.log("ERROR:", err, file);
            },
            complete: function(){
            //console.log("Done with all files");
            }
        });});
    });

    // Adds the points inputted from submited file into the coordinate
table
    function displayHTMLTable(results){
        var table = document.getElementById("myTable");
        var data = results.data;
        for(i = 0;i < data.length;i++){
            var row = data[i];
            var cells = row.join(",").split(/,/);
            for(j = 0;j < cells.length;j += 2){
                var cellContent1 = parseFloat(cells[j]);
                var cellContent2 = parseFloat(cells[j+1]);
                if (!isNaN(cellContent1) && !isNaN(cellContent2)){
                    featureID = featureID + 1;
                    // Create point feature on map canvas
```

```
                    var point_feature = new ol.Feature({ });
                    var mapcoord = ol.proj.transform([cellContent1,
cellContent2], 'EPSG:28992', 'EPSG:3857');
                    var point_geom = new ol.geom.Point(
                        [mapcoord[0],mapcoord[1]]
                    );
                    point_feature.setGeometry(point_geom);
                    point_feature.setProperties({
                       'id': featureID
                    })
                    source.addFeature(point_feature);

point_feature.setStyle(styleFunction(featureID.toString()))
                    var trow = table.insertRow(featureID);
                    trow.id = featureID;
                    var cell1 = trow.insertCell(0);
                    var cell2 = trow.insertCell(1);
                    var cell3 = trow.insertCell(2);
                    cell1.innerHTML = featureID;
                    cell2.innerHTML = cellContent1.toFixed(3);;
                    cell3.innerHTML = cellContent2.toFixed(3);;
                }
            }
        }
    }

</script>
<script>
    //Used to toggle the menu on small screens when clicking on the menu
button
    function myFunction() {
        var x = document.getElementById("navDemo");
        if (x.className.indexOf("w3-show") == -1) {
            x.className += " w3-show";
        } else {
            x.className = x.className.replace(" w3-show", "");
        }
    }
</script>
</html>
```

Appendix C – Python Code: Tiling

```
from Tkinter import Tk
from tkFileDialog import askopenfilename, asksaveasfile
import Tkinter
import tkMessageBox
import shapefile
import laspy
import numpy as np
import os
import time

def integer(geom):
    geometry = []
    append = geometry.append
    for point in geom:
        x, y = point[0], point[1]
        x, y = int(x*1000), int(y*1000)
        append([x, y])
    geometry = np.array(geometry) #Closed polygon
    return geometry

def inside_polygon(pt, index, minY, maxY, maxX, geom, fraction):
    condition1 = np.logical_and(pt[1] > minY, pt[1] < maxY)
    condition2 = pt[0] < maxX
    condition12 = np.logical_and(condition1, condition2)

    intersX = geom[:,0][:-1] + (pt[1] - geom[:,1][:-1])*fraction
    condition3 = pt[0] <= intersX

    truth = np.logical_and(condition12, condition3)
    intersections = truth[truth == True].size

    if intersections%2 == 1:
        tiling(pt, index)

def tiling(pt, index):
    found = np.where(np.logical_and(np.logical_and(pt[0] >= tiles[:,2],
pt[0] < tiles[:,4]),
                                    np.logical_and(pt[1] >= tiles[:,3],
pt[1] < tiles[:,5])))
    col, row = tiles[found][0][0], tiles[found][0][1]
    tile_dict[col, row].append(index)

def write():
    for i in xrange (columns):
        for j in xrange (rows):
            if tile_dict[i,j] != []:
                write_tile = tiles[i*columns + j]
                name = [j, i, write_tile[2]/1000.0,write_tile[3]/1000.0
                        , write_tile[4]/1000.0, write_tile[5]/1000.0]
                fileName = "{},{},{},{},{},{}.las".format(*name)
                outFile = laspy.file.File(path + fileName, mode = "w",
                                          header = head)
                outFile.points = inFile.points[tile_dict[i, j]]
                outFile.header.scale = [1,1,1]
                outFile.close()

if __name__ == '__main__':
    #SHP Message Box
    Tk().withdraw()
```

```
tkMessageBox.showinfo("Information", "Choose .shp file")

#SHP Selection Window
Tk().withdraw()
SHPfile = askopenfilename()
shpFound = False

while shpFound == False:
    try:
        sf = shapefile.Reader(SHPfile)
        shpFound = True
    except:
        tkMessageBox.showwarning('WARNING!', 'Choose .shp file')
        Tk().withdraw()
        SHPfile = askopenfilename()

#Read .shp file
records = sf.iterShapeRecords()
record = next(records)

#Multipolygon Geometry
geom = record.shape.points
geom = integer(geom)

#Edges
minY = np.fmin(geom[:,1][:-1], geom[:,1][1:])
maxY = np.fmax(geom[:,1][:-1], geom[:,1][1:])
maxX = np.fmax(geom[:,0][:-1], geom[:,0][1:])
nom = geom[:,0][1:] - geom[:,0][:-1]
denom = geom[:,1][1:] - geom[:,1][:-1]
fraction = np.divide(nom, denom)

#Bounding Rectangle
x_min, x_max = np.amin(geom[:,0]), np.amax(geom[:,0])
y_min, y_max = np.amin(geom[:,1]), np.amax(geom[:,1])

#LAS Message Box
Tk().withdraw()
tkMessageBox.showinfo("Information", "Choose .las file(s)")

#LAS Selection Window
Tk().withdraw()
LASfile = askopenfilename()
lasFound = False

while lasFound == False:
    try:
        #Read .LAS file
        begin = time.clock()
        inFile = laspy.file.File(LASfile, mode = "r")
        lasFound = True
    except:
        tkMessageBox.showwarning('WARNING!', 'Choose .las file')
        Tk().withdraw()
        LASfile = askopenfilename()

#Create 100m tiles
x_ext = x_max - x_min
y_ext = y_max - y_min

columns = (x_ext / 100000) +1
```

```
rows = (y_ext / 100000) +1

head = inFile.header
path = os.getcwd() + '/Tiles/'
tiles = []
tile_dict = {}
append = tiles.append
for i in xrange(columns):
    for j in xrange(rows):
        xmin, xmax = x_min + i*10**5, x_min + (i+1)*10**5
        ymin, ymax = y_min + j*10**5, y_min + (j+1)*10**5
        tile = [i, j, xmin, ymin, xmax, ymax]
        append(tile)
        tile_dict[i, j] = []
tiles = np.array(tiles)

for i in xrange(500):
    print i
    #Sampling interval
    start = i*10**6
    end = (i+1)*10**6

    # FILTER CLASSES - X, Y, classes arrays
    # X, Y = inFile.get_x()[start:end], inFile.get_y()[start:end]
    # classes = inFile.get_raw_classification()[start:end]
    # indices = np.arange(10**6) + start

    #Keep only vegetation, ground, buildings
    keep = np.logical_or(np.logical_or(classes == 1, classes == 2),
                         classes == 6)
    X, Y = X[keep], Y[keep]
    indices = indices[keep]

    #Check inside bbox
    incl = np.where(np.logical_and(np.logical_and(X>x_min, X<x_max),
                                   np.logical_and(Y>y_min, Y<y_max)))
    pts = np.stack((X[incl], Y[incl]), axis=-1)
    indices = indices[incl]

    for pt, index in zip(pts, indices):
        inside_polygon(pt, index, minY, maxY, maxX, geom, fraction)

write()

end = time.clock()
duration = end - begin
print('Duration: {:.3f} s'.format(duration))
```