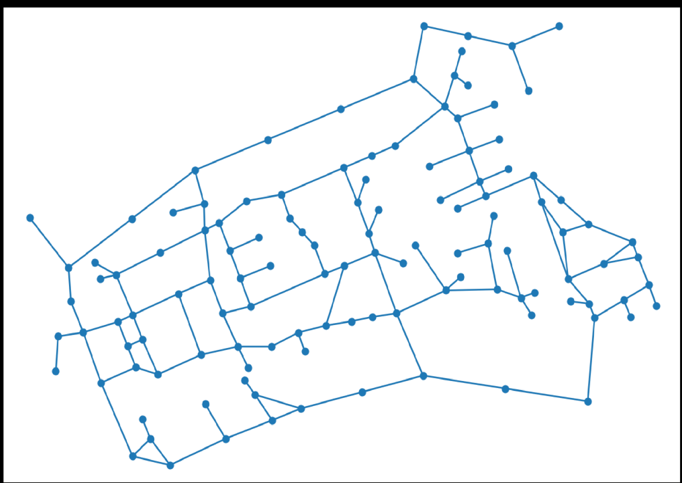
APDUDS

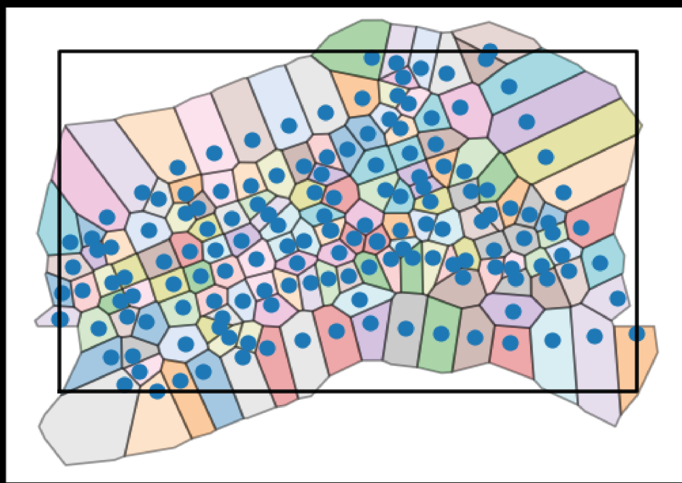
Automated Preliminary Design of Urban Drainage Systems

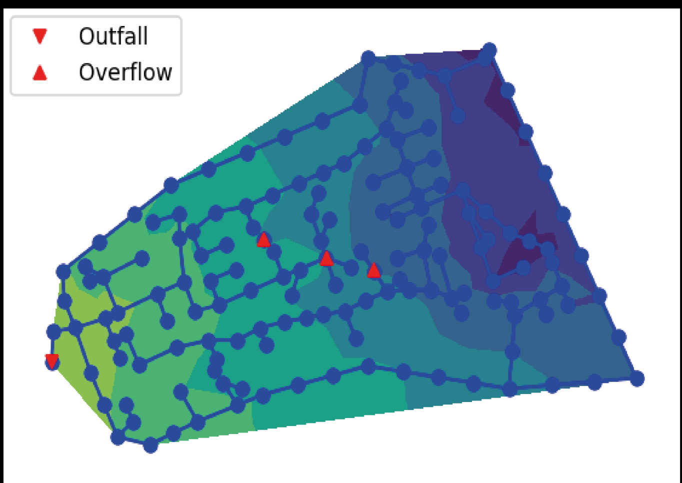
CTB3000-16: Bachelor Eindwerk

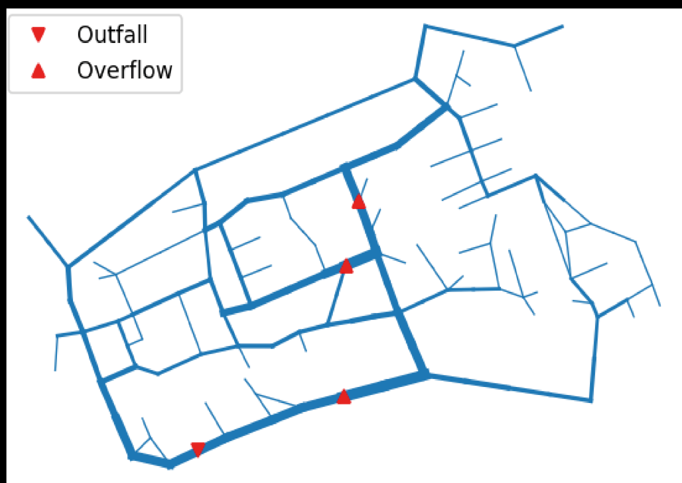
Max Lange

Delft University of Technology









APDUDS

Automated Preliminary Design of Urban Drainage Systems

by

Max Lange

First Supervisor:	J.A. Garzón Díaz
Second Supervisor:	Dr. R. Taormina
Track Coordinator:	Dr.ir. R.J. van der Ent
Course Coordinator:	Y. De Las Heras
Project Duration:	April, 2022 - June, 2022
Faculty:	Civil Engineering and Geo-Sciences, Delft
Student Number:	5169402

Cover: Collection of current results of the APDUDS software
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

I am writing this thesis as a bachelor's student in the Civil Engineering and Geo-Sciences bachelor at the TU Delft. During my bachelor's, I have also completed a minor in Computer Science, again at the TU Delft. This thesis is written as part of my Bachelor's End Project, which will allow me to receive my degree upon completion. My personal experience and interest in coding and computer science combined with my theoretical knowledge from my civil engineering bachelor's have led me to this subject. I see the goal of optimizing the civil engineering world using code, software, and computers as one of the biggest goals to strive for in the engineering world today. Thus, seeing that the preliminary design aspect of the urban drainage design world could benefit from such a tool has led me to take on this project as my bachelor thesis.

This thesis is directed towards people and engineers who are active in research or work in the urban drainage (re)design sector. It may also prove interesting for students who have already had a course on urban drainage design theory.

I would like to thank both of my supervisors J.A. Garzón Díaz and Dr. R. Taormina for assisting me in creating this thesis, both on the theoretical and execution side. I would also like to thank the track coordinator, Dr. Ir. R.J. van der Ent, and the course coordinator, Y. De Las Heras, for managing the course and making it possible for me to create this thesis.

Max Lange
Delft, June 2022

Summary

The project's objective is to create software that can quickly create a realistic preliminary design of an urban drainage system with only a few needed input parameters. With such software, designers are not only able to create a network layout with only a few manual inputs, but can also quickly ascertain the values of multiple network attributes with only a small number of needed parameters. Doing this process with software will save a significant amount of time for designers.

The software is coded entirely in Python, with the GitHub codebase being publicly available to serve as a place to download it. The program downloads the road network data for the desired area from OpenStreetMap and converts it into a network of nodes and conduits. With a small list of general system parameters given by the user, Automatic Preliminary Design of Urban Drainage Structures (APDUDS) can calculate initial values for attributes of elements in the system. The software calculates the subcatchment area for each node, the needed installation depth of the nodes, the flow direction through the conduits, and the required conduit diameters for a given design storm. As the last step, the network is transferred into a Storm Water Management Model file, making it possible to simulate the effects of a design storm to obtain a more in-depth evaluation of the system.

The software can create preliminary designs of systems for almost any kind of road network. The resulting network and its attributes are displayed to the user using multiple graphs, and the user can interact with the program through terminal prompts. The generated networks can almost completely handle the given design storm, but some issues do occur. Some flooding exist for nodes on the fringes of the system, although this is a low amount (below five percent). It also does not consider the actual geography of the area, which may result in the software creating a system that is unusable in the real world. Overall the objective of quickly creating preliminary networks has been completed, with network creation possible for any area with available OpenStreetMap road data.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
APDUDS	Automated Preliminary Design of Urban Drainage Systems
UDS	Urban Drainage System
SWMM	Storm Water Management Model

Symbols

Symbol	Definition	Unit
A	Area	$[m^2]$
Q	Flow rate	$[m^3/s]$
d	Diameter	$[m]$
u	Speed	$[m/s]$

1

Introduction

1.1. Problem Statement

When designing a new Urban Drainage System (UDS) for an area, many different variations in layout or network attribute combinations are possible. Making initial designs for multiple variations is desirable, as the values for the system parameters that come out of these designs can determine if it is a viable option. This iterative design process allows for better fine-tuning of the network, creating a more (cost) efficient final design. Even though the advantages are clear, creating these designs (and doing subsequent attribute calculations) can be time-consuming. There exist many algorithms and software which can provide quick cost assessment (Maurer et al., 2013) or faster calculations (Palumbo et al., 2014, Strecker and Huber, 2012, Wang and Wang, 2018), but a way to quickly create a preliminary design of a UDS from scratch is not among them. Having such an algorithm or software would allow designers to go through their ideas quicker, and see the immediate results of making adjustments while saving them manual changing and recalculating (in the software they use for this). Using this freed-up time to create a more detailed and optimized design would benefit the entire design process.

1.2. Objective

To develop a software tool that can create a realistic preliminary design of a UDS based on only a small number of parameters. It achieves this by using publicly available road data (to create the network layout) and several user-set parameters (to calculate various attributes such as installation depth, flow direction and rate, and conduit diameters). As an additional requirement, the program will be able to convert this constructed network into a Storm Water Management Model (SWMM) (Rossman, 2010) file format, as this software is one of the most used programs worldwide for testing UDS or similar systems.

To summarize, the requirements of the program are:

1. Construct a basic network layout, based on publicly available road network data using OpenStreetMap
2. Calculate preliminary values for major attributes of the system, such as flow direction, installation depth, and conduit dimensions
3. Convert the constructed network into the SWMM format so that it can be used in the SWMM software.

1.3. Development Approach

Python is chosen as the primary coding language for two reasons. Firstly, Python is designed to be efficient at data science operations and algorithms, which this project's software uses extensively. Secondly, it is the language with which I, as the sole developer, have the most experience. A GitHub repository has been created to contain the codebase and to facilitate a continuous integration development cycle (Duvall et al., 2007). This repository can be accessed using the following URL: https://github.com/Max-Lange/APDUDS . As the codebase is publicly available, this will also function as a link to download the software for personal use.

1.4. Code Structure

This software employs a Mediator Design Pattern, as it follows a step-by-step creation process. This way, new "steps" (a.k.a features) can be developed as standalone functions, which will then be combined into a functioning program by a "main" script (the mediator). Using this design, none of the scripts (outside of the main) interact with each other or the user. Instead, all serve as defining scripts that the main uses to create the program. It also has the added benefit of making the logic of the program easy to follow, as everything goes through the main.

The figure below shows the module structure and external package dependencies. Chapter 3 describes the inner workings of each module and elaborates on the packages used.

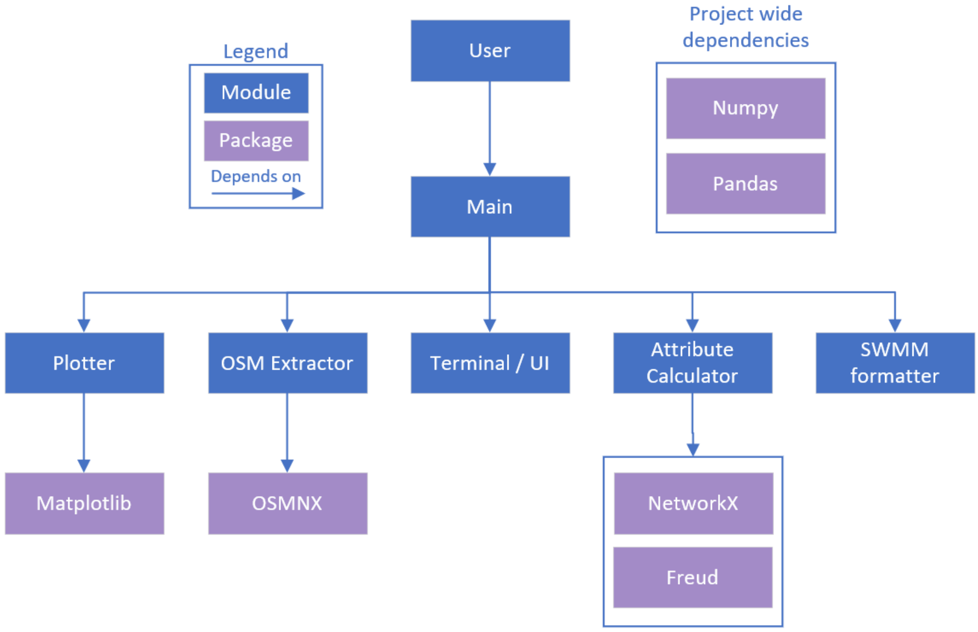


Figure 1.1: Module and Package Dependencies

2 Methodology

This chapter focuses on the methodology behind creating a preliminary UDS. It details the thought process and mathematics behind the layout of the network and the calculating of the various attributes. Translating this into code and how the program itself displays these results to the user can be found in chapter 3 and 4 respectively.

2.1. Assumptions

As the goal is to create a UDS with as few parameters as possible, some assumptions are required to achieve a simple but functioning design. These assumptions are listed below. Most of these can be accounted for using more complex algorithms but at the cost of dramatically increasing the complexity. For this reason, the project will focus on the less complex, static design methodology described in this chapter. All of these assumptions should be kept in mind when analyzing the results.

1. **Combined system type**: To keep the preliminary design simple, a combined, gravity-based system type with no pumps or more complicated elements is used. Only nodes, conduits, outfalls and overflows are implemented.
2. **Flat geography:** The selected area is assumed to be perfectly flat. This means that the actual geography and elevation of the area is not taken into account. To make this assumption work, a second smaller assumption is made, that the water which falls on a the area closest to a particular node also flows to that node.
3. **Initial one meter per second flow speed and full pipe flow:** For calculating the diameters, a flow speed of 1 $[m/s]$ is assumed. This is a value which is regularly used in basic sewer system design, but is does not reflect the real world condition. The actual flow speed is determined by multiple factors, such as flow rate and the physical characteristics of the conduits, and so may differ from the design created here.
4. **All conduits are circular:** The calculations for conduit diameters will only be done for circular cross sections. This is done as circular is the most used conduit shape, and is also a good approximation for most of the other used shapes.
5. **Homogeneous impervious ground**: To preserve simplicity, the entire area is assumed to have the same percentage of impervious ground. With the main use of the software being to design preliminary systems for neighborhoods, it is assumed that an almost homogeneous distribution of streets, houses and greenery is present.

2.2. Creating a Network Layout

Calculating attributes of a UDS will first require the construction of a network. Such a graph consists of four (4) major components:

- **Nodes:** Defines a point where water can flow into the network. These represent manholes in the actual UDS.
- **Conduits:** A pipeline which runs from one node to another, through which water can flow.
- **Outfalls:** A special kind of node. These are places where the water can drain freely. The system should be designed in such a way that water flows into these points.
- **Overflows:** Another special kind of node. These represent emergency outfalls for water to flow out of when the system is filled to over its capacity. The system should be designed with these kinds of nodes at its highest point(s).

With these components, it is possible to create a simple network. The first step is to obtain the road network for the desired area, and then place a Node at each corner or junction, connecting them up using conduits, which follow the roads. This is in line with the current design philosophy of conduits being primarily installed under paved roads, as it is easier to access them this way with roads being public property. We now have the road network, but some conduits may be too long for manholes at only the ends. Most governments require manholes to be placed within a certain distance from each other. To adhere to this rule, conduits that are too long are split into equal-sized pieces such that their new lengths are below this maximum distance. New nodes are added between the breaks to reconnect these new, smaller conduits. The result is the desired distribution of nodes and conduits.

With the nodes of the system determined, we can also calculate which section of the total area will drain into that nodes. As we assume that our area is perfectly flat, it also means that the region that is closest to a node will drain to that node. Calculating this ”closest area” can be done using the Voronoi algorithm (Voronoi, 1908). This area is also referred to as the ”subcatchment area” of that node.

2.3. Determine the Installation Depth and Direction of Flow

Now that we have the layout and sub-catchments of our network, we can start calculating some attributes of the UDS. Determining the direction of flow is the first step, which also allows us to calculate the needed depth of each node and the needed diameters of the conduits.

First is specifying the boundary conditions for the network by defining a single outfall point and multiple overflow points (described in section 2.2). A single overflow is also allowed. These will serve as the boundary points for the system, with them having to be the lowest and highest points respectively. It also needs to be clarified which design situations are desirable and which should be avoided. In general, having the total depth of all the nodes be as small as possible is preferred, as this means that less excavation is needed. Another aspect to consider is that a gravity-based sewer system requires that all conduits be installed with a minimum slope to prevent sediment buildup. With a minimum slope, having a longer conduit will hurt the design, as the nodes at the end have to be located deeper, which requires more excavation. This means that the best route for the water to take is that of the shortest distance. This assure that the final depth of the nodes is minimal.

To get the shortest path between two points on a network, we can use Dijkstra's algorithm (Dijkstra, 1959). Doing this for every node, with outfall as the endpoint, gives us the path for each. This process can be made more efficient by first looking at the leaf nodes, nodes with only one conduit connecting them. These nodes have a bigger chance of being on the outskirts of the system and thus will need to take a relatively long route to get to the outfall, going through many nodes along the way. As the network is static, meaning that the paths do not change, we can set the paths for all intermediate nodes to be the same, thus using Dijkstra's algorithm only once to cover all these nodes.

With the path of a node know, determine the needed depth for the nodes along this path becomes possible. This requires the minimum slope of the conduits. The minimum slope determines how steep the conduits have to be, such that sediment is not deposited inside by slow-moving water. Going conduit to conduit along the path, we can determine the needed depth of the end node for each by multiplying its length by the minimum slope and adding that to the depth of the beginning node. This is done for the paths of all nodes in the network. If a path goes through a node with an already established depth, the depth is updated if the newly calculated one is greater. This guarantees that the minimum depth is satisfied for all possible paths.

Last is a check for if the network satisfies the maximum slope. It is the counterpart to the minimum and makes sure that the sediment in the water does not move too fast, as that might cause wear to the conduits. To do this check, we go through all the paths of the nodes in reverse order. If a conduits slope exceeds the maximum slope, the highest node (shallowest depth) is lowered to satisfy it. The network now adheres to both the minimum and maximum slope. The flow direction is from the highest node to the lowest.

2.4. Calculating Flow Rate and Conduit Diameter

Calculating the total flow rate through each conduit needs four things. First is the peak rainfall, the maximum amount of water that will fall on the area during a single moment and is used to determine the peak inflow into the system. Second is the subcatchment area of each node, in this case already calculated in section 2.2. The third is the percentage of ground that is impervious to water. This percentage of a subcatchments area contributes to the inflow of its associated node. Lastly, we need to set the flow speed through the conduits, which in this method will always be one meter per second (see section 2.1). Summing up the inflow for all nodes for which the paths pass through a particular conduit will give us its total flow. By multiplying the peak rainfall value by the area of its subcatchment and the percentage of impervious ground, inflows of the nodes are obtained.

The conduit diameter does not require advanced calculation. $u = \frac{Q}{A}$ calculates the cross-sectional area of a conduit, as the flow rate and amount of each one are known. This equation simplifies to $A = Q$ using the assumed speed is one meter per second. $d = \sqrt{\frac{4A}{\pi}}$ then calculates the diameter of the conduits, as they will all be circular (d being the diameter). As the last step, the calculated diameter will be rounded upwards to the nearest available conduit size so that only actually used diameters are selected.

During a design storm, the overflows function as outfall points for the system. It is desirable that the conduits which flow to the overflows also have relatively large diameters so that this emergency flow is not impaired. We achieve this by redoing the flow direction, needed depth, inflow, and conduit diameter calculations for each overflow, with them taking turns serving as the designated outfall. If the conduit diameters resulting from this are larger than those of the original network, they are updated to this larger size. The entire system now has large enough conduits to flow to any outfall or overflow point.

3

Code Implementation

This chapter will outline the workings of the program itself, explaining how the design process of chapter 2 translates into code. The focus is on the actual code, with chapter 4 discussing the results and realism of the created systems. First is a list of the software limitations, while all subsequent sections explain the role of each module, along with its notable elements. Appendix A shows the source code for the entire software. In the code snippets seen throughout this report, conduits are called ”edges”. This is a remnant of earlier stages of development, and is left in to avoid major reworks.

3.1. Limitations

As one of the focus points of this project is the speed at which new designs are created, having the runtime of the software be as low as possible becomes a large part of the design philosophy. While algorithms can always be optimized, the speed of specific steps is determined by external factors (for example user input) which are out of the control of the software itself. To maintain relatively low runtimes, the limitations of the software should be known and taken into account during use. While most of them can be ignored without causing the software the crash, all will dramatically increase the runtime. A list of these limitations is given below.

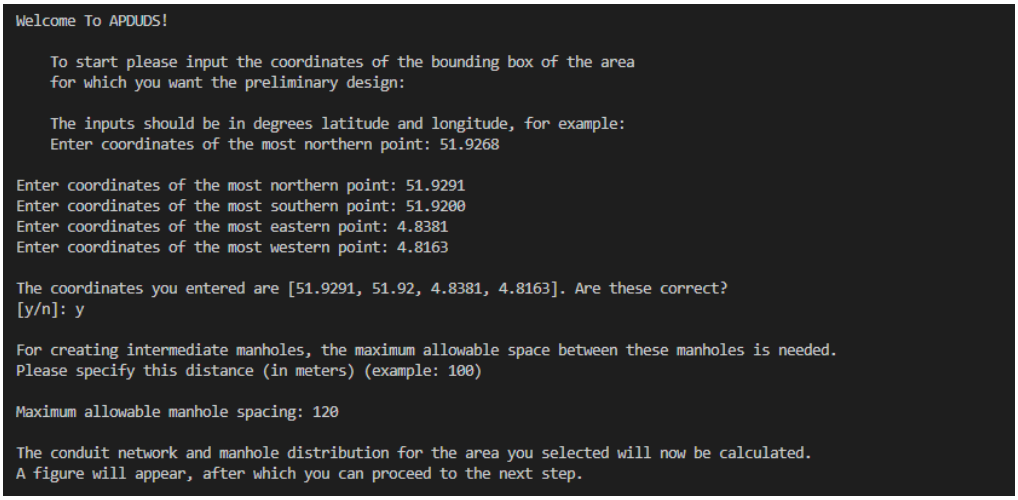
1. **Internet connection required:** As the software downloads the road network data from OpenStreetMap in real time, a constant (and stable) internet connection is required during for this step.
2. **Area size limit:** The larger the design are is, the more nodes and edges are present, which will exponentially increase computation times for almost all parts of the software. The area should not be larger than a single neighbourhood (around five square kilometers) if quick repeated use is desired.
3. **Too small gully spacing:** Making the maximum allowable gully spacing to small (below 50 [m]), can create an excessive number of extra nodes and edges. This works exponentially.
4. **Few usable diameters:** As the program always selects a diameter from the given diameter list, if this list contains either not enough possible values, or only one small diameter, the network that is produced might not comply with the given peak rainfall. It will always select the largest possible diameter if the needed diameter exceeds this largest value, but this does not mean that this diameter will be large enough.
5. **Only branched networks:** When calculating the amount of flow through each conduit, some conduits will have zero flow, and because of this they are removed from the network. This makes all the networks created by the software branched network, containing no loops.

3.2. Main

The main script is the heart of the program. It strings all the other functions together to create APDUDS. It is also the script that runs if the user wants to start the software in its entirety. They can use the program in two ways. The first is the intended way, by being given terminal prompts to insert their information (see section 3.3) and then using the inputs as parameters for the other functions. The second is a terminal-skipping or testing way. Here the skipping function contains a dictionary of all the parameters, which can be manually adjusted. When run, no terminal prompts are given, and the calculations start immediately. The main file also splits the program into three ”steps”, which reflect the objective requirements in section 1.2.

3.3. Terminal

Facilitates all the interactions the user has with the software, gives the user information about the parameters they should enter, and issues warnings if crucial information is entered incorrectly. The terminal is the only part of the software that the user sees. For example, the image below shows how the user goes about obtaining the first step of downloading and creating the network.


```
Welcome To APDUDS!

    To start please input the coordinates of the bounding box of the area
    for which you want the preliminary design:

    The inputs should be in degrees latitude and longitude, for example:
    Enter coordinates of the most northern point: 51.9268

Enter coordinates of the most northern point: 51.9291
Enter coordinates of the most southern point: 51.9200
Enter coordinates of the most eastern point: 4.8381
Enter coordinates of the most western point: 4.8163

The coordinates you entered are [51.9291, 51.92, 4.8381, 4.8163]. Are these correct?
[y/n]: y

For creating intermediate manholes, the maximum allowable space between these manholes is needed.
Please specify this distance (in meters) (example: 100)

Maximum allowable manhole spacing: 120

The conduit network and manhole distribution for the area you selected will now be calculated.
A figure will appear, after which you can proceed to the next step.
```

Figure 3.1: Terminal input example of the first window

3.4. OSM Extractor

Step 1 of the software is to download the road network and transform it into a UDS system. The OSM (OpenStreetMap) Extractor is responsible for this. It extracts the road data from the internet, removes unwanted data, cleans it up, and splits the conduits according to the maximum allowable spacing. Below is a snippet of the splitting algorithm and an image of a network before and after the split.

```
            # Add new nodes and edges for the needed nodes in the middle
            if amount > 1:
                for i in range(2, amount+1):
                    index_i = len(nodes)
                    nodes.loc[index_i] = [from_node.x + x_step_size * i, \
                        from_node.y + y_step_size * i]
                    new_edges.loc[len(new_edges)] = [index_i - 1, index_i, new_length]
```

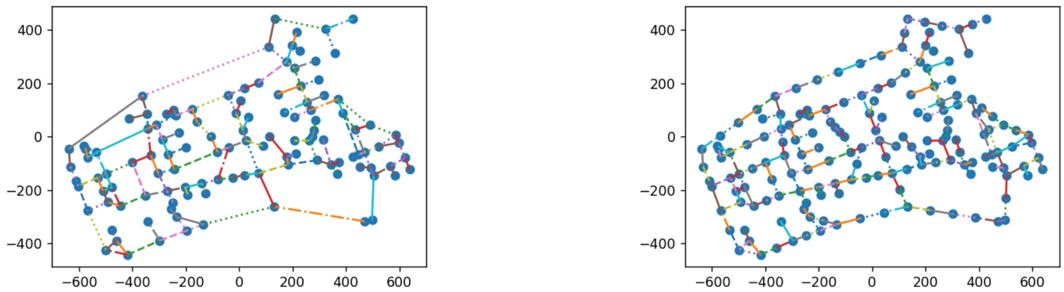


Figure 3.2: Example: Network as downloaded from OSM (left), and after being split according to the manhole spacing limit (right).

3.5. Attribute Calculator

The attribute calculator contains all the functions for point 2 of the requirements (section 1.2). It calculates the direction of flow and node depth using the Dijkstra function from NetworkX, which is a package that focuses on network operations and analysis. Even though NetworkX also has a method for the Voronoi algorithm, it does not give the area of the Voronoi cells that it creates. For this, we use the locality module of Freud, a network and graph analysis package.

3.6. SWMM Formater

For the network to open in SWMM, we must create a file that uses the same structure as a regular SWMM file. Luckily the SWMM files are stored in an easy-to-read text-based format, making them easy to replicate. The SWMM Formater is essentially one writing function, which creates all the sections of a SWMM file, and fills them with the specification of the created network. Below is an example of how APDUDS constructs one of these file sections.

```
def create_junctions_coordinates(nodes: pd.DataFrame):
    """Returns a list of strings for the junctions coordinates section"""

    coordinates = ["[COORDINATES]",
                   ";;Node           X-Coord            Y-Coord",
                   ";;-------------- ------------------ ------------------"]

    for index, node in nodes.iterrows():
        coords = "j_" + str(index) + (17 - 2 - len(str(index))) * " "
        coords += str(node.x) + (19 - len(str(node.x))) * " "
        coords += str(node.y) + (18 - len(str(node.y))) * " "
        coordinates.append(coords)
    coordinates.append("\n")
    return coordinates
```

3.7. Plotter

This module keeps the use of the matplotlib package contained to only one module. It contains all the functions that create all the different plots (which can be seen throughout this report). All the plots are graphed to a subplot so that new arrangements can be made without altering the plotting functions themselves.

4

Example Use and Results Evaluation

This chapter will first show an example use of the software in its current state, following the same steps that a user has to take. Section 4.2 tests and analyzes the software to see if it meets the set-out objective.

4.1. Example Use Case

For this example use case, we will be looking at the relatively small Dutch village of Groot-Ammers. First is the network layout creation. An explanation of the needed inputs and prompts to insert is shown in the terminal (see figure 3.1). The user inputs the north, south, west, and east coordinates of the area, alongside the maximum allowable manhole spacing. A figure appears showing the network layout for the selected region (see figure 4.1). At the same time, the program already gives the prompts to enable the next step of calculating the network attributes (figure 4.2).

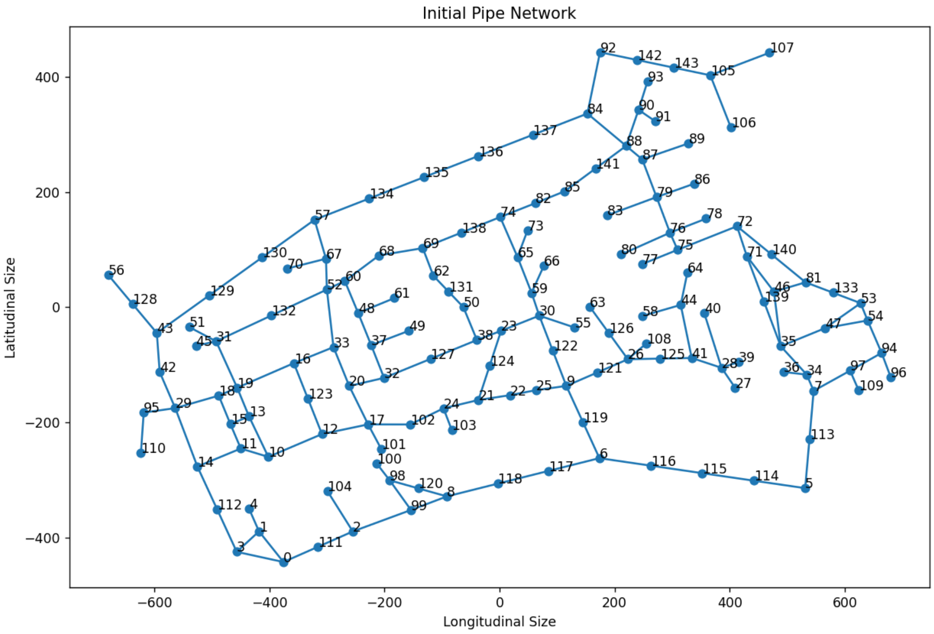


Figure 4.1: Result of the network layout creation step

```
Now that the network has been generated, some attributes can be calculated.
Please enter the described information to enable the next set of calculation steps:

The index of the point you want to designate as an outfall/pumping point:
(Should be a positive integer, for example: 78)

Outfall point index: 78


The indices of the points which you want to designate as overflow points:
(Positive integers separate by space, for example: 23 65 118)

Overflows points indices: 23 65 118


The minimum depth below the ground at which conduits can be installed:
(Should be a positive integer or decimal number, for example: 1.1)

Minimum installation depth [m]: 1.1
```

Figure 4.2: Example section of the second input window

With the figure remaining visible, the user can input a new list of variables that are parameters needed to complete the attribute calculations. The list below describes these inputs. For why the software asks for these variables please refer to chapter 2.

1. **Outfall point index**: The index of the node which will be the designated outfall.
2. **Overflow points indices**: The indices of the nodes which will be designated as overflows.
3. **The minimum depth**: The minimum required depth of the nodes and conduits in meters
4. **The minimum slope**: The minimum required slope for the conduits
5. **The maximum slope**: The maximum allowed slope for the conduits
6. **Design storm rainfall value**: The amount of rainfall for the design storm in mm/h
7. **Percentage of impervious area**: The fraction of the area for which the rainwater will flow into the system, as a percentage
8. **List of available diameters**: A list of all the diameters which can be used for the network, all in millimeters.

With the attributes calculations done, a new set of figures will appear, showing the subcatchment distribution, a contour map of the depth of the nodes, and the relative diameters of the conduits (figure 4.3). Like the first step, when displaying the graphs to the user, prompts for the SWMM file creation show up alongside them. These prompts ask for the desired name of the text file and the duration of the design storm in hours. Lastly, there is the option to enable the subcatchment polygons.

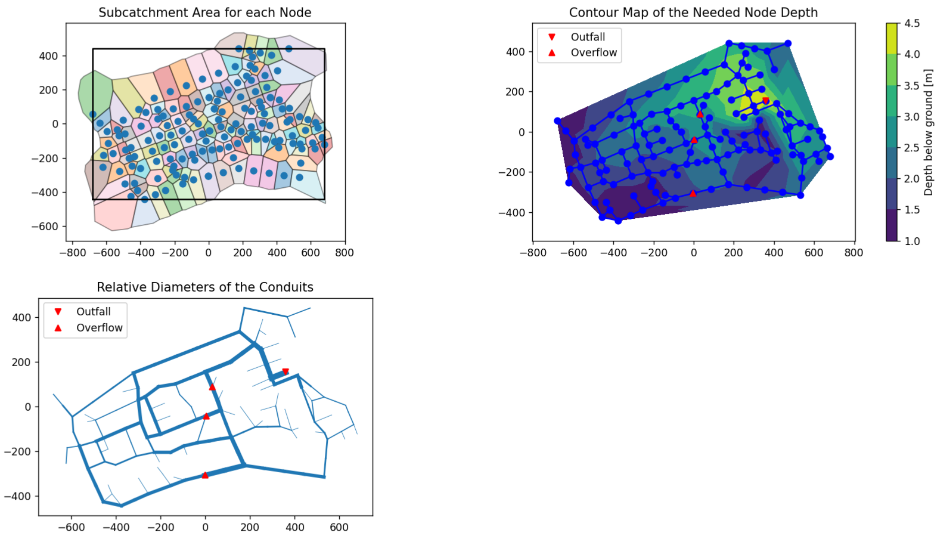


Figure 4.3: Results of the attribute calculations step

With these last inputs given, the software creates the SWMM file and then closes itself (along with the graphs of figures 4.1 and 4.3), completing a use case for APDUDS. The user can now open the created text file in SWMM, and run a simulation using a time series created based on the previously given design storm intensity and duration. They are also free to alter the system in SWMM, as the file is now separate from APDUDS. How the created network for Groot-Ammers looks in SWMM can be seen in figure 4.4, with and without the subcatchment polygons.

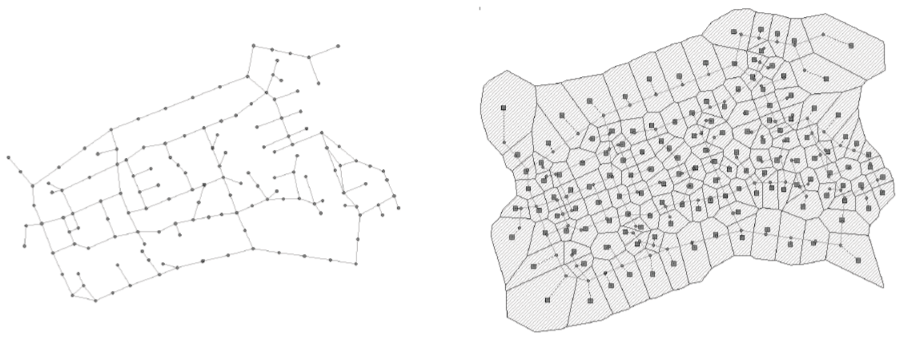


Figure 4.4: Network when opened in SWMM, with and without subcatchment polygons visible

4.2. Evaluating the results

With the program completed, evaluating it is possible for multiple criteria. This section looks at four of them: The response of the software to different situations, the realism of the obtained networks, the success of needed minimal input, and the overall speed of the software.

4.2.1. Testing on Different Situations

With APDUDS designed to work for all locations on earth that have OpenStreetMap road data available, it needs to be able to deal with all different kinds of road network shapes. The software has been run multiple times on three different types of networks. Figure 4.5. shows one example for each.

1. **A large village or small city**: Any area which is around five square kilometers in size, as this is the upper limit what the software was designed around.
2. **Design sized villages and neighborhoods**: Areas which are representative of the kinds of sizes that APDUDS was designed around
3. **Oddly shaped areas**: Oddly shaped areas might create problems for the attributes calculation step.

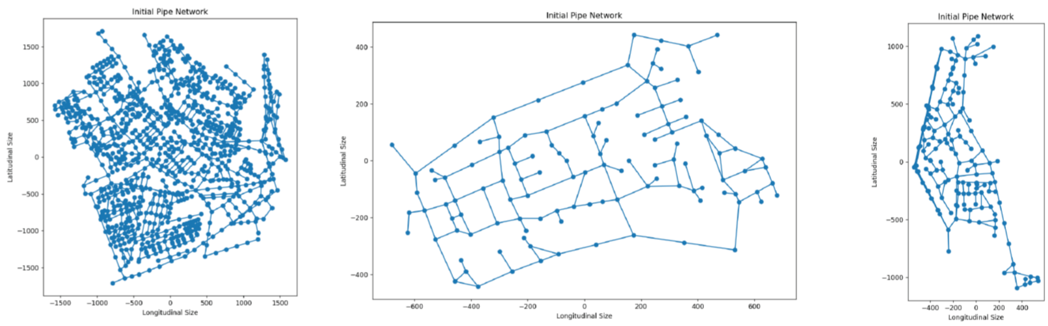


Figure 4.5: Examples of all three types of system on which APDUDS was tested: The city of IJsselstein, the village of Groot-Ammers and the villages of Zoeterwoudedorp en Zuidbuurt (left to right)

In all tests, APDUDS successfully created a network for the selected area. While the download time for all the networks remained roughly the same, the large networks took a couple of seconds longer when calculating the attributes. Some networks put conduits under road types that usually never have them, such as large highways or bridges. While the software does not see any issue with this, removing them in the future will create more realistic designs.

4.2.2. The Realism of the Created Systems

SWMM can simulate a design storm in a step-wise fashion, making it possible to test networks for things such as flooding. When running these simulations, one design flaw always seems to appear. Leaf nodes, nodes with only one conduit connected to them, almost always experience some flooding. Two possible causes are suspected. It could be that the water head in a leaf conduit is too low compared to the connecting main conduit, blocking the water of the former. The second possible reason is that the leaf conduits get smaller sizes assigned to them than they are supposed to have. With them smaller than they should be, the inflow is too large, causing flooding. Figure 4.6 shows an example. The leaf conduit on the right cannot discharge its water into the larger one and subsequently floods. While the flooding is always relatively low, below five percent of the total inflow, fixing it would improve the software. It now requires manual adjusting to resolve, which takes away from the "automatic" part of the objective.

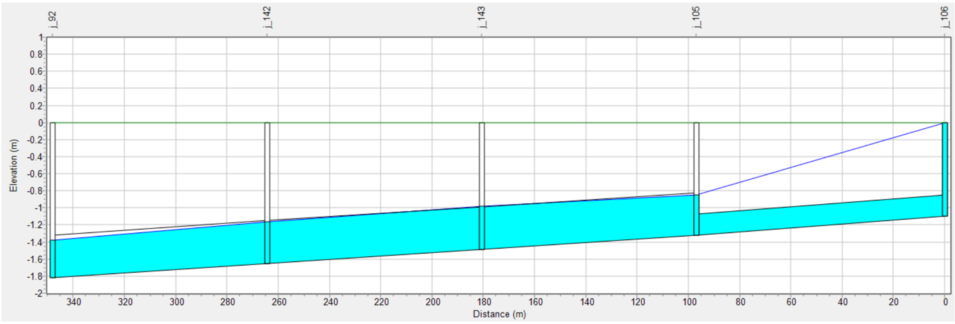


Figure 4.6: An example of conduit situation which causes upstream flooding

4.2.3. Achieving Minimal Needed Input

In the current version of the software, twelve (12) inputs complete the entire generation process, with ten (10) used for creating the network, and the last two (2) being for the SWMM file creation. These specific items are the minimum amount needed to make a preliminary system, while still giving the user options to customize it according to their parameters. If it asked fewer, the UDS would become too generalized, and the software might stray from its goal. If one or more of the attributes (see the list below) is excluded (except for 11 and 12), it would be impossible to calculate the attributes of the network without making assumptions for values that are never "assumed" in the current UDS design philosophy.

The current list of parameters that need to be entered by the user are:

1. The coordinates of the desired area's bounding box
2. The maximum allowable manhole spacing
3. The index of the outfall point
4. The index or indices of overflow point(s)
5. The minimum needed installation depth below the surface
6. The minimum slope of the conduits
7. The maximum slope of the conduits (optional)
8. The design storm hourly rainfall
9. The general percentage of impervious ground surface for the area
10. A List of available diameter sizes for the conduits
11. The duration of the design storm
12. A name for the SWMM file

4.2.4. Software speed and usability

When looking at the speed of the software, both a positive and a negative point stand out. On the positive side, the actual calculations and algorithms that the different steps use are fast, taking below a minute even for large networks like that of the city of IJsselstein. APDUDS can handle the calculations for large networks equally well as those of small systems, except for the downloading. It is the most significant point that impacts the program's speed. In some cases, the data request seems to get stuck in a waiting loop, causing a delay of around two minutes. While it does not happen all the time, improving or solving this issue will increase the user experience.

The usability also has positives and negatives. While the terminal style input windows are clear in what they ask of the user, and the graphs present themselves in windows that can be enlarged and zoomed into, a more user-friendly user interface is preferred. As the target demographic of APDUDS are engineers who most likely already have experience with computers or software like SWMM, the terminal might still hold some people back from regular use of the software. An actual user interface, multiple windows, explanations screens, and options for going back and forth through the designing steps could improve the user experience. It can also speed up the designing process, as the user would not have to re-download the road data from OpenStreetMap if they could go back from the SWMM step to the attribute calculation step, should they want to change some of the parameters. Overall the current terminal prompt method is sufficient in guiding the user through the designing process while being easy to implement, which freed up more time to work on the actual functionality of the algorithms themselves.

5 Discussion

In this chapter, the success of fulfilling the original goal will is discussed, alongside other possible uses for the software. There will also be a reflection on the difficulties faced during development and possible future improvements.

5.1. The Potential Use and Impact

Except for the flooding issue described in section 4.2.2, the software can create a preliminary design of a UDS for an area based on only the given road network and ten other general parameters entered by the user. While the network does require adjustments to satisfy as a working system, it does complete its goal of letting designers quickly create preliminary designs for the desired area. With APDUDS, going through possible designs is made quicker, and if one requires further tweaking, they can manually do this in SWMM.

With how the software creates its designs, another possible use for the software would be generating large numbers of possible UDSs for artificial intelligence training. As the network is relatively low in complexity, it is easy to make hundreds of variations for any given area. Generating training data this way can be much faster than manually obtaining existing UDS designs, and could improve or speed up the training of such artificial intelligence. A tester function already present in the current version of APDUDS could easily be converted into such a generation function.

5.2. Troubles During Development

During the development of APDUDS, a complication that kept cropping up was the personal lack of understanding of how UDSs functioned. For example, the behaviors of outfalls and overflows or the workings of the SWMM software were unknown to me before starting this project. Discovering this in the middle of the development period meant that some algorithms had to be adjusted or remade. Working the methodology out in better detail would have prevented such errors, saving time.

5.3. Future Improvements

With the time window for this project coming to a close, the current version of APDUDS is in its completed form. Improvements are still possible regarding the realism of the created designs. The major ones are listed below:

- **Solving the flooding problem**: The flooding issue described in section 4.2.2 is a hamper on the system's realism. For the design storm, the system should be able to drain all the rainfall with zero flooding. Obtaining this would enable the software to create a realistic upper bound for a possible UDS.
- **Better road type filtering**: The issue described in 4.2.1 poses another conflict with reality. If we want to avoid manually filtering the incorrect road types, the program needs to sort them out in advance. An option for this is present in the OSMNX package used to download the data, but more research into its filtering system needs is required to see if it is possible.
- **Conduit height offset by conduit diameter**: In the current version, the bottom of the cross-section of a conduit determines its depth. If one with a diameter of, for example, 3 meters is situated 1 meter below the ground, it sticks out two meters above the surface. While this is not realistic, determining the node depth is done with a currently arbitrary zero value, so the actual values of the depths do not impact the workings of the system, as long as the depth is relative to the other nodes and conduits is correct. It would be beneficial to fix this issue to avoid confusion in the future.

Next to these "fixes" other possible feature additions have been conceptualized. Implementing these in the future will make the software both easier to use and improve its realism:

- **A user interface**: Having a user interface for the program would improve the user experience. It would also allow for more complex navigation options, such as returning to previous steps or excluding certain attributes from being calculated. This way users would have more control over which kinds of preliminary designs they want to create.
- **Including geographical elevation data**: With the current version assuming that the selected area is perfectly flat, translating the designs into the real world becomes inaccurate once the actual area is not. A more complicated algorithm, which would take the potential and kinetic energy into account, could solve this issue. While this would require an almost complete rework of the software, it would also dramatically improve the realism of the designs, saving more time in the designing process.

6
Conclusion

APDUDS can create a realistic preliminary design of an urban drainage system for any area, given that OpenStreetMap road data is available. With the user only needing to input 12 parameters to create the system, and with the software converting the network into a SWMM file, a new preliminary design is ready to be analyzed in mere minutes. Time is saved for the user by not having to manually construct the network themselves, which can be put into iterating or optimizing these designs, thus speeding up the designing process.

While the created designs are valid, improvements are possible, and some issues still need to be resolved. Leaf nodes almost always seem to experience relatively small amounts flooding, with the specific reason for this not yet known. Ignoring the actual geography of the area may require manual adjusting in SWMM to have the network better reflect the real world situation. Still, the ability to quickly create a SWMM network from a small list of parameters enables time-saving, compensating for these required adjustments. Not needing to manually insert each node or conduit into the network saves a significant amount of time that can instead be spent on optimizing and improving the UDS.

References

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematlk I*, 269–271. http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf

Duvall, P., Matyas, S., & Glover, A. (2007). *Continuous integration: Improving software quality and reducing risk* (Illustrated). Pearson Education. https://martinfowler.com/books/duvall.html

Maurer, M., Scheidegger, A., & Herlyn, A. (2013). Quantifying costs and lengths of urban drainage systems with a simple static sewer infrastructure model. *Urban Water Journal*, *10*, 268–280. https://doi.org/10.1080/1573062X.2012.731072

Palumbo, A., Cimorelli, L., Covelli, C., Cozzolino, L., Mucherino, C., & Pianese, D. (2014). Optimal design of urban drainage networks. *Civil Engineering and Environmental Systems*, *31*, 79–96. https://doi.org/10.1080/10286608.2013.820277

Rossman, L. (2010). Storm water management model user's manual version. *United States Enviromental Protection Agency*. https://www.epa.gov/water-research/storm-water-management-model-swmm

Strecker, E. W., & Huber, W. C. (2012). Global solutions for urban drainage. *Global Solutions for Urban Drainage*. https://ascelibrary.org/doi/abs/10.1061/9780784406441

Voronoi, G. (1908). Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, *1908*, 97–102. https://doi.org/10.1515/crll.1908.133.97

Wang, S., & Wang, H. (2018). Extending the rational method for assessing and developing sustainable urban drainage systems. *Water Research*, *144*, 112–125. https://doi.org/10.1016/j.watres.2018.07.022

A

Source Code

Attribute Calculator

```python
"""Defining file for all the attribute calculation functions of step 2 of APDUDS

This script requires that `networkx`, `pandas`, `freud` and `numpy` be installed within the Python
environment you are running this script in.

This file contains the following major functions:

    * voronoi_area - Calculates the catchment area for each node using voronoi
    * flow_and_height - Determinte the flow direction and set the node depth
    * flow_amount - Determine the amount of water flow through each conduit
    * diameter_calc - Determine the appropriate diameter for eac conduit
    * cleaner_and_trimmer - Remove intermediate information and precision from the data
    * attribute_calculations - Runs the entire attribute calculation process
    * tester - Only used for testing purposes
"""

import networkx as nx
import pandas as pd
from freud.box import Box
from freud.locality import Voronoi
import numpy as np

def voronoi_area(nodes: pd.DataFrame):
    """Calculates the catchment area for the nodes using voronoi

    Args:
        nodes (pd.DataFrame): The node data of a network

    Returns:
        tuple([pd.DataFrame, Freud.locality.Voronoi]): Node data with added subcatchment area
        values, and freud voronoi object
    """

    nodes = nodes.copy()

    box = Box(Lx=nodes.x.max() * 2, Ly=nodes.y.max() * 2, is2D=True)
    points = np.array([[nodes.x[i], nodes.y[i], 0] for i in range(len(nodes))])

    voro = Voronoi()
    voro.compute((box, points))

    nodes["area"] = voro.volumes

    return nodes, voro
```

```
def flow_and_depth(nodes: pd.DataFrame, edges: pd.DataFrame, settings:dict):
    """Determines the direction of flow of the water (using Dijkstra's algorithm) and
    needed installation depth of the nodes based on the given settings.

    Args:
        nodes (DataFrame): The node data of a network
        edges (DataFrame): The conduit data of a network
        settings (dict): Parameters for the network

    Returns:
        tuple[DataFrame, DataFrame]: Node data with added depth and path values,
        and conduit data with "from" and "to" columns corrected
    """

    nodes = nodes.copy()
    edges = edges.copy()

    nodes, edges, graph = intialize(nodes, edges, settings)
    end_points = settings["outfalls"]
    nodes.loc[end_points, "considered"] = True
    # Create a set of all the "to" "from" combos of the conduits for later calculations
    edge_set = [set([edges["from"][i], edges["to"][i]]) for i in range(len(edges))]

    i = 1
    while not nodes["considered"].all():
        # Using the number of connections to sort them will make leaf nodes be considered
            first,
        # which has a larger change to include more nodes in one dijkstra run
        leaf_nodes = nodes.index[nodes.connections == i].tolist()

        for node in leaf_nodes:
            if not nodes.at[node, "considered"]:
                path = determine_path(graph, node, end_points)
                nodes = set_paths(nodes, path)
                nodes = set_depth(nodes, edges, path, settings["min_slope"], edge_set)

                nodes.loc[path, "considered"] = True
        i += 1

    if "max_slope" in settings:
        nodes = uphold_max_slope(nodes, edges, edge_set, settings["max_slope"])

    edges = reset_direction(nodes, edges)
    return nodes, edges


def intialize(nodes: pd.DataFrame, edges: pd.DataFrame, settings: dict):
    """Add the needed columns to the node and edge datasets to facilitate the operations
    of later functions. Also creates a networkx graph for the dijkstra calculations

    Args:
        nodes (DataFrame): The node data of a network
        edges (DataFrame): The conduit data of a network
        settings (dict): Parameters for the network

    Returns:
        tuple[DataFrame, DataFrame, Graph]: The node and edge datasets with the needed
            columns
        added, and a networkx graph of the network
    """

    nodes["considered"] = False
    nodes["depth"] = settings["min_depth"]
    nodes["role"] = "node"
    nodes["path"] = None

    # Some more complex pandas operations are needed to get the connection numbers in a few
        lines
    ruined_edges = edges.copy()
    edges_melted = ruined_edges[["from", "to"]].melt(var_name='columns', value_name='index')
```

```python
    edges_melted["index"] = edges_melted["index"].astype(int)
    nodes["connections"] = edges_melted["index"].value_counts().sort_index()

    graph = nx.Graph()
    graph.add_nodes_from(list(nodes.index.values))

    for _, edge in edges.iterrows():
        graph.add_edge(edge["from"], edge["to"], weight=edge["length"])

    return nodes, edges, graph


def determine_path(graph: nx.Graph, start: int, ends: list[int]):
    """Determines the shortest path from a certain point to another point on a networkx graph
    using Dijkstra's shortes path algorithm

    Args:
        graph (Graph): A NetworkX Graph object of the network
        start (int): The index of the starting node
        end (int): The index of the end node

    Returns:
        list[int]: The indicies of the nodes which the shortes path passes through
    """

    shortest_length = np.inf
    best_path = []

    for end_point in ends:
        length, path = nx.single_source_dijkstra(graph, start, target=end_point)

        if length < shortest_length:
            best_path = path
            shortest_length = length

    # Generator expression is needed to remove the .0 that is added by networkx' dijkstra
    return [int(x) for x in best_path]


def set_paths(nodes: pd.DataFrame, path: list):
    """Determine the path to the outfall for each node, and add this to the node data

    Args:
        nodes (DataFrame): The node data for a network
        path (list[int]): The indicies of the nodes which a path passes through

    Returns:
        DataFrame: Node data with the relevant path values updated
    """

    for i, node in enumerate(path):

        if not nodes.loc[node, "path"]:
            nodes.at[node, "path"] = path[i:]

    return nodes


def set_depth(nodes: pd.DataFrame, edges: pd.DataFrame,
              path: list, min_slope: float, edge_set: list[set[int]]):
    """Set the depth of the nodes along a certain route using the given minimum slope.

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        path (list): All the indicies of the nodes which the path passes through
        min_slope (float): The value for the minimum slope [m/m]

    Returns:
        DataFrame: The node data with the relevant depth values updated
    """
```

```python

    for i in range(len(path) - 1):
        from_node = path[i]
        to_node = path[i+1]

        from_depth = nodes.at[from_node, "depth"]
        # Use the edge set to get the conduit index
        length = edges.at[edge_set.index(set([from_node, to_node])), "length"]
        new_to_depth = from_depth + min_slope * length

        # Only update the depth if the new depth is deeper than the current depth
        if new_to_depth > nodes.at[to_node, "depth"]:
            nodes.at[to_node, "depth"] = new_to_depth

    return nodes

def uphold_max_slope(nodes: pd.DataFrame, edges: pd.DataFrame,\
                     edge_set: list[set[int]], max_slope: float):
    """Checks if the conduits uphold the max slope rule, and alters/lowers the relevant nodes
    when this isn't the case

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        edge_set (list[set[int]]): A list of sets of all the "from" "to" node combos
        of the conduits
        max_slope (float): The value of the maximum slope [m/m]

    Returns:
        DataFrame: The node data with the depth value updated were needed
    """

    for _, node in nodes.iterrows():
        path = node.path

        for i in range(len(path)-1):
            # Move backwards through the list as the depth can only become greater
            lower_node = path[-1-i]
            higher_node = path[-2-i]
            # Use the edge set to get the conduit index
            length = edges.at[edge_set.index(set([lower_node, higher_node])), "length"]

            # Only update the depth if the current slope is greater than the max slope
            if abs(nodes.at[lower_node, "depth"] - nodes.at[higher_node, "depth"])\
                 / length > max_slope:
                nodes.at[higher_node, "depth"] = nodes.at[lower_node, "depth"] - length *
                    max_slope

    return nodes


def reset_direction(nodes: pd.DataFrame, edges: pd.DataFrame):
    """Flips the "from" and "to" columns for all conduits where needed if depth is reversed

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network

    Returns:
        DataFrame: Conduit data with the "from" "to" order flipped were needed
    """

    for i, edge in edges.iterrows():
        if nodes.at[edge["from"], "depth"] > nodes.at[edge["to"], "depth"]:
            edges.at[i, "from"], edges.at[i, "to"] = edge["to"], edge["from"]

    return edges


def flow_amount(nodes: pd.DataFrame, edges: pd.DataFrame, settings: dict):
    """Calculate the amount of flow through each conduit
```

```

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        settings (dict): Network parameters

    Returns:
        tuple[DataFrame, DataFrame]: Node and conduit data with the inflow and flow
        values added
    """

    nodes = nodes.copy()
    edges = edges.copy()

    nodes["inflow"] = nodes["area"] * (settings["peak_rain"] / (10**7))\
         * (settings["perc_inp"] / 100)
    edges["flow"] = 0
    edge_set = [set([edges["from"][i], edges["to"][i]]) for i in range(len(edges))]

    for _, node in nodes.iterrows():
        path = node["path"]

        for j in range(len(path)-1):
            edge = set([path[j], path[j+1]])
            edges.at[edge_set.index(edge), "flow"] += node["inflow"]

    return nodes, edges


def diameter_calc(edges: pd.DataFrame, diam_list: list[float]):
    """Determine the needed diameter for the conduits from a given list of diameters using the
    calculate flow amount

    Args:
        edges (DataFrame): The conduit data for a network
        diam_list (list[float]): List of the different usable diameter sizes for the
        conduits [m]

    Returns:
        DataFrame: Conduit data with diameter values added
    """

    edges["diameter"] = None

    for i, edge in edges.iterrows():
        precise_diam = 2 * np.sqrt(edge["flow"] / np.pi)

        if edge["flow"] == 0:
            edges.at[i, "diameter"] = 0

        # Special case if the precise diameter is larger than the largest given diameter
        elif precise_diam > diam_list[-1]:
            edges.at[i, "diameter"] = diam_list[-1]
            print(f"WARNING: Conduit between node {int(edge['from'])} and {int(edge['to'])} \
requires a larger diameter than is available ({round(precise_diam, 3)} m). \
Capped to {diam_list[-1]}")

        else:
            for size in diam_list:
                if size - precise_diam > 0:
                    edges.at[i, "diameter"] = size

                    break

    return edges


def cleaner_and_trimmer(nodes: pd.DataFrame, edges: pd.DataFrame):
    """Remove the columns from the node and conduit dataframes which were only needed for the
    attribute calculations. Also round off the calculated values to realistic presicions
```

```python

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network

    Returns:
        tuple[DataFrame, DataFrame]: Cleaned up nodes and conduit data
    """

    nodes = nodes.drop(columns=["considered", "path", "connections"])

    # Special condition if data was obtained from a csv (only for testing purposes)
    if "Unnamed: 0" in nodes.keys():
        nodes = nodes.drop(columns=["Unnamed: 0"])
        edges = edges.drop(columns=["Unnamed: 0"])

    # cm precision for x, y and depth
    # m^2 precision for area
    # L precision for inflow
    nodes.x = nodes.x.round(decimals=2)
    nodes.y = nodes.y.round(decimals=2)
    nodes.area = nodes.area.round(decimals=0)
    nodes.depth = nodes.depth.round(decimals=2)
    nodes.inflow = nodes.inflow.round(decimals=3)

    # cm precision for length
    # L precision for flow
    edges.length = edges.length.round(decimals=2)
    edges.flow = edges.flow.round(decimals=3)

    # Drop the conduits with 0 flow
    edges = edges[edges.flow != 0]
    edges = edges.reset_index(drop=True)

    return nodes, edges


def add_outfalls(nodes: pd.DataFrame, edges: pd.DataFrame, settings: dict):
    """Add extra nodes for the selected outfall and overflow nodes. Connect them up with new
    conduits

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        settings (dict): Parameters for the network

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data with extra nodes and conduits
        for the outfalls and overflows
    """

    for outfall in settings["outfalls"]:
        new_index = len(nodes)
        nodes.loc[new_index] = [nodes.at[outfall, "x"] + 1,
                                nodes.at[outfall, "y"] + 1,
                                0,
                                nodes.depth.max(),
                                "outfall",
                                0]

        edges.loc[len(edges)] = [outfall,
                                 new_index,
                                 1,
                                 0,
                                 settings["diam_list"][-1]]

    for overflow in settings["overflows"]:
        new_index = len(nodes)
        nodes.loc[new_index] = [nodes.at[overflow, "x"] + 1,
                                nodes.at[overflow, "y"] + 1,
                                0,
```

```python
                                settings["min_depth"],
                                "overflow",
                                0]

        edges.loc[len(edges)] = [new_index,
                                 overflow,
                                 1,
                                 0,
                                 settings["diam_list"][-1]]

    return nodes, edges


def loop(nodes: pd.DataFrame, edges: pd.DataFrame, settings: dict):
    """Runs the main attibute calculations loop for a given network

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        settings (dict): Parameters for the network

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data which the attribute values updated
    """

    nodes, edges = flow_and_depth(nodes, edges, settings)
    nodes, edges = flow_amount(nodes, edges, settings)
    edges = diameter_calc(edges, settings["diam_list"])

    return nodes, edges


def attribute_calculation(nodes: pd.DataFrame, edges: pd.DataFrame, settings: dict):
    """Does the complete attribute calculation step for a given network

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        settings (dict): Parameters for the network

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data with newly added and updated
        attribute values
    """
    nodes, voro = voronoi_area(nodes)

    nodes_copy = nodes.copy()
    edges_copy = edges.copy()

    nodes, edges = loop(nodes, edges, settings)
    print("Main attribute calculations completed, moving on to overflow diameter calculations...")

    loop_setting = settings.copy()
    for overflow in settings["overflows"]:
        loop_setting["outfalls"] = [overflow]
        _, loop_edges = loop(nodes_copy, edges_copy, loop_setting)

        for i in range(len(edges)):
            if edges.at[i, "diameter"] < loop_edges.at[i, "diameter"]:
                edges.at[i, "diameter"] = loop_edges.at[i, "diameter"]
                edges.at[i, "flow"] = loop_edges.at[i, "flow"]

        print(f"Calculations for the overflow at node {overflow} completed...")

    nodes, edges = cleaner_and_trimmer(nodes, edges)
    nodes, edges = add_outfalls(nodes, edges, settings)

    return nodes, edges, voro

```

```python
def tester():
    """Only used for testing purposes
    """
    print("attribute_calculator script has run")


if __name__ == "__main__":
    tester()
```

Main

```python
"""Main script of APDUDS. Running this script (with either main or tester) starts
the entire software.

This script requires that `matplotlib` and `pandas` be installed within the Python
environment you are running this script in, as well as all the packages required
by the modules `osm_extractor`, `plotter`, `terminal`, `swmm_formater` and `attribute_calculator`.

This file contains the following functions:

    * step_1 - Runs the network creation step of the software
    * step_2 - Runs the attribute calculation step of the software
    * step_3 - Runs the SWMM file creation step of the software
    * main - Starts the software in it's entirity. Run this function to run the entire software
    * tester - Only used for testing, can also be used for a terminal-skipping run of the software
"""

import warnings
from pandas import DataFrame
from swmm_formater import swmm_file_creator
from osm_extractor import extractor, cleaner, splitter
from plotter import network_plotter, voronoi_plotter, height_contour_plotter, diameter_map
from terminal import step_1_input, step_2_input, step_3_input, area_check
from attribute_calculator import attribute_calculation
from matplotlib import pyplot as plt
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)


def step_1(coords: list[float], space: int, block: bool = False):
    """Preform the network creation step of the software by running the appropriate functions.
    Also display some graphs which are relevent to the results of these functions

    Args:
        coords (list[float]): The north, south, east and west coordinates of the desired area
        space (int): The maximum allowable manhole spacing
        block (bool, optional): Decides wether displaying the graph pauses the run.
        Defaults to False.

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data of the created network
    """

    print("\nStarting the OpenStreetMap download. This may take some time, please only close the \
software after 5 minutes of no response....")
    nodes, edges = extractor(coords)

    print("Completed the OpenStreetMap download, starting the data cleaning...")
    filtered_nodes, filtered_edges = cleaner(nodes, edges)

    print("Completed the data cleaning, started the conduit splitting...")
    split_nodes, split_edges = splitter(filtered_nodes, filtered_edges, space)

    print("Completed the conduit splitting, plotting graphs...")
    _ = plt.figure()
    network_plotter(split_nodes, split_edges, 111, numbered=True)
    plt.show(block=block)

    return split_nodes, split_edges


def step_2(nodes: DataFrame, edges: DataFrame, settings: dict, block: bool = False):
    """Preform the attribute calculation step of the software by running the appropiate functions.
    Also display some graphs which are relevent to the results of these functions
```

```

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        settings (dict): The parameters for a network

    Returns:
        tuple[DataFrame, DataFrame, freud.locality.voronoi]: Node and conduit data with
            updated
        values for the attributes, as well as a voronoi object for use in the SWMM file
            creation
    """

    print("\nStarting the attribute calculation step...")
    nodes, edges, voro = attribute_calculation(nodes, edges, settings)
    print("Completed the attribute calculations, plotting graphs...")

    fig = plt.figure()
    voronoi_plotter(nodes, voro, 221)
    height_contour_plotter(nodes, edges, 222, fig)
    diameter_map(nodes, edges, 223)

    fig.tight_layout()
    plt.show(block=block)

    return nodes, edges, voro

def step_3(nodes: DataFrame, edges: DataFrame, voro, settings: dict):
    """Preform the SWMM file creation step of the software by running the appropriate
        functions.

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        voro (freud.locality.voronoi): Voronoi object of the nodes of a network
        settings (dict): The parameters for a network
    """

    print("\nStarting the SWMM file creation...")
    swmm_file_creator(nodes, edges, voro, settings)
    print("Completed the SWMM file creation.")


def main():
    """Running this function starts the software in its entirety.
    Run this function if you want to use the software in the intended way
    """

    coords, space = step_1_input()
    area_check(coords, 5)
    nodes, edges = step_1(coords, space)

    settings = step_2_input()
    nodes, edges, voro = step_2(nodes, edges, settings)

    settings.update(step_3_input())
    step_3(nodes, edges, voro, settings)



def tester():
    """Only used for testing, but can also be used as a way to run the program as intended,
    while skipping the terminal interaction stage.
    """

    test_coords = [51.9291, 51.9200, 4.8381, 4.8163]
    test_space = 120

    area_check(test_coords, 5)
    nodes, edges = step_1(test_coords, test_space, block=True)

```

```
    test_settings = {"outfalls":[111],
                     "overflows":[23, 65, 118],
                     "min_depth":1.1,
                     "min_slope":1/500,
                     "peak_rain": 36,
                     "perc_inp": 50,
                     "diam_list": [0.25, 0.5, 0.6, 0.75, 1.0, 1.25, 1.5, 2.0, 2.5],
                     "filename": "test_swmm",
                     "max_slope": 1/450,
                     "duration": 2,
                     "polygons": "n"}

    nodes, edges, voro = step_2(nodes, edges, test_settings, block=True)

    step_3(nodes, edges, voro, test_settings)


if __name__ == "__main__":
    main()
```

OSM Extractor

```python
"""Defines the OpenStreetMap extraction and conversion functions.

This script defines the functions that facilitate the downloading and reformating
of road map data from OpenStreetMap (OSM) (using osmnx for the downloading part).
It also defines the function that splits the obtained edges into smaller equal section
as per the user defined maximum edge length.

This script requires that `osmnx`, `pandas` and `numpy` be installed within the Python
environment you are running this script in.

This file contains the following functions:

    * extractor - Downloads the wanted area from OpenStreetMap
    * cleaner - Cleans and standerdizes the data downloaded by the extractor
    * splitter - Splits the obtained conduits according to the given parameters
    * tester - Only used for testing purposes
"""

import osmnx as ox
import pandas as pd
import numpy as np
ox.config(use_cache=False)

def extractor(coords: list, aggregation_size=15):
    """Downloads the road network from OpenStreetMap, and filters out the unwanted data

    Args:
        coords (list): noth, south, east and west coordinates of the desired bounding box
        aggregation_size (int, optional): Max distance by which to aggrigate nearby nodes.
        Defaults to 15.

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data of the network
    """

    # Download osm data, reproject into meter-using coordinate system, consolidate nearby nodes
    osm_map = ox.graph_from_bbox(coords[0], coords[1], coords[2], coords[3], network_type="drive")

    osm_projected = ox.project_graph(osm_map)
    osm_consolidated = ox.consolidate_intersections(osm_projected,
                                                    tolerance=aggregation_size,
                                                    dead_ends=True)

    # Seperate the nodes and edges, and reset multidimensional index
    osm_nodes, osm_edges = ox.graph_to_gdfs(osm_consolidated)
    nodes_reset = osm_nodes.reset_index()
    edges_reset = osm_edges.reset_index()

    # Create new nodes and edges dataframe which only contain the desired data
    int_from = [int(edges_reset.u[i]) for i in range(len(edges_reset))]
    int_to = [int(edges_reset.v[i]) for i in range(len(edges_reset))]
    edges = pd.DataFrame({"from":int_from,
                          "to":int_to,
                          "length":edges_reset.length})
    nodes = pd.DataFrame({"x":nodes_reset.x,
                          "y":nodes_reset.y,})

    return nodes, edges


def cleaner(nodes: pd.DataFrame, edges: pd.DataFrame):
    """Standerdizes and cleans the data downloaded from OpenStreetMap by the extractor

    Args:
        nodes (pd.DataFrame): The node data for a network
        edges (pd.DataFrame): The conduit data for a network

```

```python
    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data with updated and cleaned
            values
    """

    nodes = nodes.copy()
    edges = edges.copy()

    # Reset the x and y values of the nodes to start from 0
    nodes.x = nodes.x - nodes.x.min()
    nodes.y = nodes.y - nodes.y.min()

    # And then make the center of the network the center of the axes
    nodes.x = nodes.x - (nodes.x.max() / 2)
    nodes.y = nodes.y - (nodes.y.max() / 2)

    nodes.x = nodes.x.round(decimals=2)
    nodes.y = nodes.y.round(decimals=2)

    # Duplicate edges may exist. These need to be filtered out
    combos = []
    filtered_edges = pd.DataFrame(columns=["from", "to", "length"])
    for _, line in edges.iterrows():
        if line["from"] != line["to"]:
            combo = set([line["from"], line["to"]])

            if combo not in combos:
                filtered_edges.loc[len(filtered_edges)] = [line["from"], line["to"], line["
                    length"]]
                combos.append(combo)

    filtered_edges.length = filtered_edges.length.round(decimals=2)
    filtered_edges[["from", "to"]] = filtered_edges[["from", "to"]].astype(int)

    return nodes, filtered_edges


def splitter(nodes: pd.DataFrame, edges: pd.DataFrame, max_space: int):
    """Splits conduits which exceed a certain lenght into equally sized section,
    and connects them back up by adding nodes in the splits.

    Args:
        nodes (pd.DataFrame): The node data for a network
        edges (pd.DataFrame): The conduit data for a network
        max_space (int): The maximum allowable manhole spacing by which the conduits will be
            split

    Returns:
        tuple[DataFrame, DataFrame]: The node and conduit data with extra nodes and conduits
            added
        where conduits needed to be split.
    """

    nodes = nodes.copy()
    # Create a new dataframe to be filled with the correct edges
    new_edges = pd.DataFrame(columns=["from", "to", "length"])
    for _, line in edges.iterrows():
        # If the line is below the max_length, just add it to the new dataframe
        if line.length <= max_space:
            new_edges.loc[len(new_edges)] = [line["from"], line["to"], line["length"]]

        # Otherwise it needs to be split
        else:
            from_node = nodes.iloc[int(line["from"])]
            to_node = nodes.iloc[int(line["to"])]

            # Amount of splits, new lenght of resulting pipe sections, and x,y stepsize
            amount = int(np.ceil(line["length"] / max_space) - 1)
            new_length = line["length"] / (amount + 1)

            # Determine the direction in which to advance the x and y coords
```

```
            x_step_size = (to_node.x - from_node.x) / (amount + 1)
            y_step_size = (to_node.y - from_node.y)  / (amount + 1)

            # Special case for the first node and edge
            index_i = len(nodes)
            nodes.loc[index_i] = [from_node.x + x_step_size, from_node.y + y_step_size]
            new_edges.loc[len(new_edges)] = [line["from"], index_i, new_length]

            # Add new nodes and edges for the needed nodes in the middle
            if amount > 1:
                for i in range(2, amount+1):
                    index_i = len(nodes)
                    nodes.loc[index_i] = [from_node.x + x_step_size * i, \
                        from_node.y + y_step_size * i]
                    new_edges.loc[len(new_edges)] = [index_i - 1, index_i, new_length]

            # Special case for the last edge
            new_edges.loc[len(new_edges)] = [index_i, line["to"], new_length]

    # Clean up and round of the newly constructed data
    nodes.x = nodes.x.round(decimals=2)
    nodes.y = nodes.y.round(decimals=2)
    new_edges.length = new_edges.length.round(decimals=2)
    new_edges[["from", "to"]] = new_edges[["from", "to"]].astype(int)
    return nodes, new_edges


def tester():
    """Only used for testing purposes"""
    print("osm_extractor script has run")

if __name__ == "__main__":
    tester()
```

Plotter

```python
"""Defining file for all plotting functions

This script requires that `matplotlib`, `numpy` and `pandas` be installed within the Python
environment you are running this script in.

This file contains the following major functions:

    * network_plotter - Creates a plot containing the nodes (as points)
     and the conduits (as lines)
    * voronoi_plotter - Creates a plot of the nodes of a network, and the
    subcatchment area polygons
    * height_contour_plotter - Creates a plot containing a network, and a filled in contour plot
    of the depth values of the nodes
    * diameter_map - Creates a plot of the conduits of a network, with the thickness of the lines
    corresponding to the relative diameter size
"""

from matplotlib import pyplot as plt
import numpy as np
from pandas import DataFrame

def network_plotter(nodes: DataFrame, edges: DataFrame, subplot_number: int, numbered=False):
    """Plots the nodes and conduits for a network as points and lines respectivly

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        subplot_number (int): Ax to plot to
    """

    axes = plt.subplot(subplot_number)
    plt.plot(nodes.x, nodes.y, "o")

    for _, line in edges.iterrows():
        x_coord = [nodes.at[int(line["from"]), "x"], nodes.at[int(line["to"]), "x"]]
        y_coord = [nodes.at[int(line["from"]), "y"], nodes.at[int(line["to"]), "y"]]
        plt.plot(x_coord, y_coord, "#1f77b4")

    if numbered:
        for index, node in nodes.iterrows():
            axes.annotate(str(index), xy=(node.x, node.y), color="k")

    axes.set_title("Initial Pipe Network")
    axes.set_xlabel("Longitudinal Size")
    axes.set_ylabel("Latitudinal Size")
    plt.axis('scaled')


def voronoi_plotter(nodes: DataFrame, voro, subplot_number: int):
    """Plot the nodes as points and the subcathment areas a colored polygons

    Args:
        nodes (DataFrame): The node data for a network
        voro (freud.locality.voronoi): freud voronoi instance containting polygon information
        subplot_number (int): Ax to plot to
    """

    axes = plt.subplot(subplot_number)
    points = np.array([[nodes.x[i], nodes.y[i], 0] for i in range(len(nodes))])

    voro.plot(ax=axes, color_by_sides=False)
    axes.scatter(points[:, 0], points[:, 1])

    axes.set_title("Subcatchment Area for each Node")
    plt.axis("scaled")


```

```python
def height_contour_plotter(nodes: DataFrame, edges: DataFrame, subplot_number:int, fig):
    """Creates a subplot of a contourmap of the depth of the nodes, with the conduit
    network laid overtop.

    Args:
        nodes (DataFrame): The node data for a network
        edges (DataFrame): The conduit data for a network
        subplot_number (int): Ax to plot to
    """

    axes = plt.subplot(subplot_number)

    for _, node in nodes.iterrows():
        if node.role == "node":
            axes.plot(node.x, node.y, "bo")

    for _, line in edges.iterrows():
        x_coord = [nodes.at[int(line["from"]), "x"], nodes.at[int(line["to"]), "x"]]
        y_coord = [nodes.at[int(line["from"]), "y"], nodes.at[int(line["to"]), "y"]]
        plt.plot(x_coord, y_coord, "b")

    outfalls = nodes.index[nodes['role'] == "outfall"].tolist()
    for outfall in outfalls:
        # Special first case for adding a label
        if outfall == outfalls[0]:
            axes.plot(nodes.at[outfall, "x"], nodes.at[outfall, "y"], "rv", label="Outfall")

        else:
            axes.plot(nodes.at[outfall, "x"], nodes.at[outfall, "y"], "rv")

    overflows = nodes.index[nodes['role'] == "overflow"].tolist()
    for overflow in overflows:
        # Special first case for adding a label
        if overflow == overflows[0]:
            axes.plot(nodes.at[overflow, "x"], nodes.at[overflow, "y"], "r^", label="Overflow
                ")

        else:
            axes.plot(nodes.at[overflow, "x"], nodes.at[overflow, "y"], "r^")

    # Add the colored contours
    x_coords = nodes.x[(nodes.role == "node") | (nodes.role == "outfall")]
    y_coords = nodes.y[(nodes.role == "node") | (nodes.role == "outfall")]
    depths = nodes.depth[(nodes.role == "node") | (nodes.role == "outfall")]
    contourf = axes.tricontourf(x_coords, y_coords, depths)

    # Add extra points on the end to get a larger graph extent
    axes.scatter([nodes.x.min()-50, nodes.x.max()+50],
                 [nodes.y.min()-50, nodes.y.max()+50],
                 color="white")

    cbar = fig.colorbar(contourf, ax=axes)
    cbar.set_label("Depth below ground [m]")

    axes.set_title("Contour Map of the Needed Node Depth")
    axes.legend()
    plt.axis("scaled")


def diameter_map(nodes: DataFrame, edges: DataFrame, subplot_number:int):
    """Creates a subplot of the conduits of the system, with the line thickness corresponding
     to
    the diameter size.

    Args:
        nodes (DataFrame): The node data of a network
        edges (DataFrame): The conduit data of a network
        diam_list (list[float]): List of the viable diameters (in [m])
        subplot_number (int): Ax to plot to
    """

```

```python
    axes = plt.subplot(subplot_number)
    scalar = edges.diameter.max()

    outfalls = nodes.index[nodes['role'] == "outfall"].tolist()
    outfalls.extend(nodes.index[nodes['role'] == "overflow"].tolist())
    for _, line in edges.iterrows():
        if line["from"] not in outfalls and line["to"] not in outfalls:
            x_coord = [nodes.at[int(line["from"]), "x"], nodes.at[int(line["to"]), "x"]]
            y_coord = [nodes.at[int(line["from"]), "y"], nodes.at[int(line["to"]), "y"]]
            plt.plot(x_coord, y_coord, "#1f77b4", linewidth=line["diameter"] * 8 / scalar)

    outfalls = nodes.index[nodes['role'] == "outfall"].tolist()
    for outfall in outfalls:
        # Special first case for adding a label
        if outfall == outfalls[0]:
            axes.plot(nodes.at[outfall, "x"], nodes.at[outfall, "y"], "rv", label="Outfall")

        else:
            axes.plot(nodes.at[outfall, "x"], nodes.at[outfall, "y"], "rv")

    overflows = nodes.index[nodes['role'] == "overflow"].tolist()
    for overflow in overflows:
        # Special first case for adding a label
        if overflow == overflows[0]:
            axes.plot(nodes.at[overflow, "x"], nodes.at[overflow, "y"], "r^", label="Overflow
                ")

        else:
            axes.plot(nodes.at[overflow, "x"], nodes.at[overflow, "y"], "r^")


    axes.set_title("Relative Diameters of the Conduits")
    axes.legend()
    plt.axis("scaled")


def tester():
    """Only used for testin purposes"""
    print("The plotter script has run")


if __name__ == "__main__":
    tester()
```

SWMM Formater

```python
"""Defining file for creating a swmm file

This script requires that `pandas` be installed within the Python
environment you are running this script in.

This file contains the following major functions:

    * swmm_file_creator - Creates a txt file of the networkwhich can be used in the swmm
      software
    * tester - Only used for testing purposes
"""

from datetime import datetime
import pandas as pd

def swmm_file_creator(nodes: pd.DataFrame, edges: pd.DataFrame, voro, settings: dict):
    """Creates a .txt file which follows the System Water Management Model format (SWMM),
    so that the created network can be used in that software

    Args:
        nodes (DataFrame): The node data of a network
        edges (DataFrame): The conduit data of a network
        voro (freud.locality.voronoi): voronoi object of the nodes of a network
        settings (dict): Parameters for a network
        filename (str): Desired name for the SWMM file
    """

    with open(f"{settings['filename']}.txt", 'w', encoding="utf8") as file:

        title = create_title()
        file.write('\n'.join(title))

        date = datetime.today().strftime('%m/%d/%Y')
        options = create_options(date)
        file.write('\n'.join(options))

        evaporation = create_evaporation()
        file.write('\n'.join(evaporation))

        raingage = create_raingage()
        file.write('\n'.join(raingage))

        subcatchments = create_subcatchments(nodes, settings)
        file.write('\n'.join(subcatchments))

        subareas = create_subcatchement_subareas(nodes)
        file.write('\n'.join(subareas))

        infiltration = create_subcatchement_infiltration(nodes)
        file.write('\n'.join(infiltration))

        junctions = create_junctions(nodes)
        file.write('\n'.join(junctions))

        outfalls = create_outfalls(nodes)
        file.write('\n'.join(outfalls))

        conduits = create_conduits(edges)
        file.write('\n'.join(conduits))

        xsections = create_cross_section(edges)
        file.write('\n'.join(xsections))

        timeseries = create_timeseries(settings, date)
        file.write('\n'.join(timeseries))

        report = create_report()
        file.write('\n'.join(report))

```

```python
        tags = create_tags()
        file.write('\n'.join(tags))

        map_settings = create_map_settings(nodes)
        file.write('\n'.join(map_settings))

        coordinates = create_junctions_coordinates(nodes)
        file.write('\n'.join(coordinates))

        if settings["polygons"] == "y":
            polygons = create_subcatchment_polygons(nodes, voro)
            file.write('\n'.join(polygons))

        symbols = create_symbols(nodes)
        file.write('\n'.join(symbols))


def create_title():
    """Returns a list of strings for the title section"""

    title = ["[TITLE]",
             ";;Project Title/Notes",
             "\n"]
    return title


def create_options(date: str):
    """Returns a list of strings for the options section"""

    options = ["[OPTIONS]",
               ";;Option              Value",
               "FLOW_UNITS            CMS",
               "INFILTRATION          HORTON",
               "FLOW_ROUTING          DYNWAVE",
               "LINK_OFFSETS          DEPTH",
               "MIN_SLOPE             0",
               "ALLOW_PONDING         NO",
               "SKIP_STEADY_STATE     NO",
               "",
              f"START_DATE            {date}",
               "START_TIME            00:00:00",
              f"REPORT_START_DATE     {date}",
               "REPORT_START_TIME     00:00:00",
              f"END_DATE              {date}",
               "END_TIME              12:00:00",
               "SWEEP_START           1/1",
               "SWEEP_END             12/31",
               "DRY_DAYS              0",
               "REPORT_STEP           00:05:00",
               "WET_STEP              00:01:00",
               "DRY_STEP              00:10:00",
               "ROUTING_STEP          0:00:10",
               "RULE_STEP             00:00:00",
               "",
               "INERTIAL_DAMPING      PARTIAL",
               "NORMAL_FLOW_LIMITED   BOTH",
               "FORCE_MAIN_EQUATION   H-W",
               "VARIABLE_STEP         0.75",
               "LENGTHENING_STEP      0",
               "MIN_SURFAREA          0",
               "MAX_TRIALS            0",
               "HEAD_TOLERANCE        0",
               "SYS_FLOW_TOL          5",
               "LAT_FLOW_TOL          5",
               "MINIMUM_STEP          0.5",
               "THREADS               1",
               "\n"]
    return options


def create_evaporation():
```

```python
    """Returns a list of strings for the evaporation section"""

    evaporation = ["[EVAPORATION]",
                   ";;Data Source    Parameters",
                   ";;-------------- ----------------",
                   "CONSTANT         0.0",
                   "DRY_ONLY         NO",
                   "\n"]
    return evaporation


def create_raingage():
    """Returns a list of strings for the raingage section"""

    raingages = ["[RAINGAGES]",
                 ";;Name           Format    Interval SCF      Source ",
                 ";;-------------- --------- ------ ------ ----------",
                 "General          INTENSITY 0:05     1.0      TIMESERIES Design_Storm",
                 "\n"]
    return raingages


def create_subcatchments(nodes: pd.DataFrame, settings: dict):
    """Returns a list of strings for the subcatchments section"""

    subcatchments = ["[SUBCATCHMENTS]",
                     ";;Name           Rain Gage        Outlet           Area     %Imperv  \
Width    %Slope   CurbLen  SnowPack",
                     ";;-------------- ---------------- ---------------- -------- -------- \
-------- -------- -------- ----------------"]

    for node_index, node in nodes.iterrows():
        if node.role == "node":
            nr_length = len(str(node_index))
            catchment = "sub_" + str(node_index) + (17 - 4 - nr_length) * " "
            catchment += "General" + (17 - 7) * " "
            catchment += "j_" + str(node_index) + (17 - 2 - nr_length) * " "
            catchment += str(round(node.area / 10000, 4)) + \
(9 - len(str(round(node.area * 0.0001, 4)))) * " "
            catchment += str(settings['perc_inp']) + (9 - len(str(settings['perc_inp']))) * " "
            catchment += "500      0.5      0"

            subcatchments.append(catchment)
    subcatchments.append("\n")
    return subcatchments


def create_subcatchement_subareas(nodes: pd.DataFrame):
    """Returns a list of strings for the subareas section"""

    subareas = ["[SUBAREAS]",
                ";;Subcatchment   N-Imperv   N-Perv     S-Imperv   S-Perv     PctZero    \
RouteTo    PctRouted",
                ";;-------------- ---------- ---------- ---------- ---------- ---------- \
---------- ----------"]

    for node_index, node in nodes.iterrows():
        if node.role == "node":
            nr_length = len(str(node_index))
            subarea = "sub_" + str(node_index) + (17 - 4 - nr_length) * " "
            subarea += "0.01       0.1        0.05       0.05       25         OUTLET"

            subareas.append(subarea)
    subareas.append("\n")
    return subareas


def create_subcatchement_infiltration(nodes: pd.DataFrame):
    """Returns a list of strings for the infilatrion section"""

```

```
    infiltration = ["[INFILTRATION]",
                    ";;Subcatchment   Param1     Param2     Param3     Param4     Param5",
                    ";;-------------- ---------- ---------- ---------- ---------- ----------"]

    for node_index, node in nodes.iterrows():
        if node.role == "node":
            nr_length = len(str(node_index))
            infil = f"sub_{node_index}" + (17 - 4 - nr_length) * " "
            infil += "3.0        0.5        4          7          0"

            infiltration.append(infil)
    infiltration.append("\n")
    return infiltration


def create_junctions(nodes: pd.DataFrame):
    """Returns a list of strings for the junctions section"""

    junctions = ["[JUNCTIONS]",
                 ";;Name           Elevation  MaxDepth   InitDepth  SurDepth   Aponded",
                 ";;-------------- ---------- ---------- ---------- ---------- ----------"]

    for node_index, node in nodes.iterrows():
        if node.role == "node":
            nr_length = len(str(node_index))
            junc = "j_" + str(node_index) + (17 - 2 - nr_length) * " "
            junc += "-" + str(node.depth) + (10 - len(str(node.depth))) * " "
            junc += str(node.depth) + (11 - len(str(node.depth))) * " "
            junc += "0          0          0"

            junctions.append(junc)
    junctions.append("\n")
    return junctions


def create_outfalls(nodes: pd.DataFrame):
    """Returns a list of strings for the outfalls section"""

    outfalls = ["[OUTFALLS]",
                ";;Name           Elevation  Type       Stage Data       Gated    Route To",
                ";;-------------- ---------- ---------- ---------------- -------- \
----------------"]

    for index, node in nodes.iterrows():
        if node.role in ["outfall", "overflow"]:
            out = "j_" + str(index) + (17 - 2 - len(str(index))) * " "
            depth = "-" + str(nodes.at[index, 'depth'])
            out += depth + (10 - len(depth)) * " "
            out += "FREE                        NO"

            outfalls.append(out)
    outfalls.append("\n")
    return outfalls


def create_conduits(edges: pd.DataFrame):
    """Returns a list of strings for the conduits section"""

    conduits = ["[CONDUITS]",
                ";;Name           From Node        To Node          Length     Roughness  \
InOffset   OutOffset  InitFlow   MaxFlow",
                ";;-------------- ---------------- ---------------- ---------- ---------- \
---------- ---------- ---------- ----------"]

    for edge_index, edge in edges.iterrows():
        conduit = "c_" + str(edge_index) + (17 -2 - len(str(edge_index))) * " "
        conduit += "j_" + str(int(edge['from'])) + (17 - 2 - len(str(int(edge['from'])))) * " "
        conduit += "j_" + str(int(edge["to"])) + (17 - 2 - len(str(int(edge["to"])))) * " "
        conduit += str(edge.length) + (11 - len(str(edge.length))) * " "
```

```
        conduit += "0.01       0          0          0          0"

        conduits.append(conduit)
    conduits.append("\n")
    return conduits


def create_cross_section(edges: pd.DataFrame):
    """Returns a list of strings for the xsections section"""

    xsections = ["[XSECTIONS]",
                 ";;Link           Shape        Geom1            Geom2      Geom3      \
Geom4      Barrels    Culvert",
                 ";;-------------- ------------ ---------------- ---------- ---------- \
---------- ---------- ----------"]

    for edge_index, edge in edges.iterrows():
        x_sec = "c_" + str(edge_index) + (17 - 2 - len(str(edge_index))) * " "
        x_sec += "CIRCULAR     "
        x_sec += str(edge.diameter) + (17 - len(str(edge.diameter))) * " "
        x_sec += "0          0          0          1"

        xsections.append(x_sec)
    xsections.append("\n")
    return xsections


def create_timeseries(settings: dict, date: str):
    """Returns a list of strings for the timeseries section"""

    timeseries = ["[TIMESERIES]",
                  ";;Name           Date       Time       Value",
                  ";;-------------- ---------- ---------- ----------"]

    for time in range(0, settings["duration"]*60, 5):
        step = "Design_Storm     "
        step += date + (11 - len(date)) * " "

        hours, minutes = int(time // 60), int(time % 60)
        str_time = str(hours) + ":"
        if minutes == 0:
            str_time += "00"

        elif minutes == 5:
            str_time += "05"

        else:
            str_time += str(minutes)

        step += str_time + (11 - len(str_time)) * " "
        step += str(settings["peak_rain"] * 0.36)

        timeseries.append(step)
    timeseries.append("\n")
    return timeseries


def create_report():
    """Returns a list of strings for the report section"""

    report = ["[REPORT]",
              ";;Reporting Options",
              "SUBCATCHMENTS ALL",
              "NODES ALL",
              "LINKS ALL",
              "\n"]
    return report


def create_tags():
    """Returns a list of strings for the tags section"""
```

```
    tags = ["[TAGS]",
            "\n"]
    return tags


def create_map_settings(nodes: pd.DataFrame):
    """Returns a list of strings for the map settings section"""

    map_settings = ["[MAP]",
                   f"DIMENSIONS {round(nodes.x.min()-200, 2)} {round(nodes.y.min()-200, 2)} \
{round(nodes.x.max()+200, 2)} {round(nodes.y.max()+200, 2)}",
                    "Units      Meters",
                    "\n"]
    return map_settings


def create_junctions_coordinates(nodes: pd.DataFrame):
    """Returns a list of strings for the junctions coordinates section"""

    coordinates = ["[COORDINATES]",
                   ";;Node            X-Coord            Y-Coord",
                   ";;--------------- ------------------ -------------------"]

    for index, node in nodes.iterrows():
        coords = "j_" + str(index) + (17 - 2 - len(str(index))) * " "
        coords += str(node.x) + (19 - len(str(node.x))) * " "
        coords += str(node.y) + (18 - len(str(node.y))) * " "
        coordinates.append(coords)
    coordinates.append("\n")
    return coordinates


def create_subcatchment_polygons(nodes: pd.DataFrame, voro):
    """Returns a list of strings for the polygons section"""

    polygons = ["[Polygons]",
                ";;Subcatchment    X-Coord            Y-Coord",
                ";;--------------- ------------------ -------------------"]

    polytopes = voro.polytopes
    for index, node in nodes.iterrows():
        if node.role == "node":
            polygon = polytopes[index]

            for point in polygon:
                poly_point = "sub_" + str(index) + (17 - 4 - len(str(index))) * " "
                poly_point += str(round(point[0], 2)) + \
                    (19 - len(str(round(point[0], 2)))) * " "
                poly_point += str(round(point[1], 2)) + \
                    (18 - len(str(round(point[1], 2)))) * " "
                polygons.append(poly_point)

    polygons.append("\n")
    return polygons


def create_symbols(nodes: pd.DataFrame):
    """Returns a list of strings for the symbols section"""

    symbols = ["[SYMBOLS]",
                ";;Gage            X-Coord            Y-Coord",
                ";;--------------- ------------------ -------------------",]
    gage = "General          "
    gage += str(round(nodes.x.min()-100, 2)) + \
(19 - len(str(round(nodes.x.min()-100, 2)))) * " "
    gage += str(round(nodes.y.max()+100, 2)) + \
(19 - len(str(round(nodes.y.max()+100, 2)))) * " "
    symbols.append(gage)
    symbols.append("\n")
    return symbols
```

```python


def tester():
    """For testing purposes only"""
    print("The swmm_formater script has run")


if __name__ == "__main__":
    tester()
```

Terminal

```
"""Defining file for all terminal interaction functions

This script defines the functions that facilitate the interaction between the
user and the program via the terminal.

This file contains the following major functions:

    * area_check - Prints a warning if an area is above a certain threshold
    * yes_no_choice - Presents a yes no [y/n] input space to the user
    * step_1_input - Create the explanations and input space for the network creatin step
    * step_2_input - Create the explanations and input space for the attribute calculation
      step
    * step_3_input - Create the explanations and input space for the SWMM file creation step
    * tester - Only used for testing purposes
"""

def area_check(coords: list[float], threshold: int):
    """Checks wether a given area is larger than a certain threshold of km^2,
    and prints a warning if it is

    Args:
        coords (list[float]): north, south, east and west coordinates of an area
        threshold (int): The value to check against
    """

    vert = abs(coords[0] - coords[1]) * (40075 / 360)
    hor = abs(coords[2] - coords[3]) * (40075 / 360)

    area = vert * hor

    if area > threshold + 10:
        print("\n WARNING: The area you have selected may be larger than 5 km^2.\n\
This may cause a serious increase in runtime.")


def yes_no_choice() -> str:
    """Presents the user with a yes no choice input line

    Returns:
        str: either "y" or "n", the choice of the user
    """

    try:
        choice = input("[y/n]: ").lower()

    except ValueError:
        print("\nWrong input type, please try again:")
        choice = yes_no_choice()

    if choice not in ["y", "n"]:
        print("\nWrong input type, please try again:")
        choice = yes_no_choice()

    return choice


def coords_input() -> list[float]:
    """Present the user with the input space for the bounding box coordinates

    Returns:
        list[float]: north, south, east and west coordinates
    """

    try:
        north = float(input("Enter coordinates of the most northern point: "))
        south = float(input("Enter coordinates of the most southern point: "))
        east = float(input("Enter coordinates of the most eastern point: "))
        west = float(input("Enter coordinates of the most western point: "))
        coords = [north, south, east, west]
```

```python

    except ValueError:
        print("\nThe input was not in the correct format (ex: 51.592)\nPlease try again:\n")
        coords = coords_input()

    # If north was entered in the south entry space, swap them
    if coords[0] < coords[1]:
        coords[0], coords[1] = coords[1], coords[0]

    # Same for east and west
    if coords[2] < coords[3]:
        coords[2], coords[3] = coords[3], coords[2]

    print(f"\nThe coordinates you entered are {coords}. Are these correct?")
    choice = yes_no_choice()

    if choice == "n":
        print("")
        coords = coords_input()

    return coords


def manhole_space_input() -> int:
    """Present the user with a space to input the maximum allowable manhole spacing

    Returns:
        int: maximum allowable manhole spacing (in [m])
    """

    try:
        space = int(input("Maximum allowable manhole spacing: "))

    except ValueError:
        print("\nThe input was not in the correct format (ex: 20)\nPlease try again:\n")
        space = manhole_space_input()

    return space


def step_1_input():
    """Create the explanations and input space for the network creation step of the software

    Returns:
        tuple[list[float], int]: A list of the desired bounding box coordinates, and an integer
        value for the maximum allowable manhole spacing
    """


    print("\nWelcome To APDUDS!\n\n\
    To start please input the coordinates of the bounding box of the area \n\
    for which you want the preliminary design:\n\n\
    The inputs should be in degrees latitude and longitude, for example:\n\
    Enter coordinates of the most northern point: 51.9268\n")

    coords = coords_input()

    print("\nFor creating intermediate manholes, the maximum allowable space between\
 these manholes is needed.\nPlease specify this distance (in meters) (example: 100)\n")

    space = manhole_space_input()

    print("\nThe conduit network and manhole distribution for the area you selected will \
now be calculated.\nA figure will appear, after which you can proceed to the next step.")

    return coords, space


def step_2_input():
    """Create the explanations and input spaces for the attribute calculations step of the
```

```
    software

    Returns:
        dict: The parameters for the system as given by the user
    """

    settings = {}

    print("\nNow that the network has been generated, some attributes can be calculated.\n\
Please enter the described information to enable the next set of calculation steps:")

    print("\nThe index of the point you want to designate as an outfall/pumping point:\n\
(Should be a positive integer, for example: 78)\n")
    outfalls = input("Outfall point index: ").split()
    settings["outfalls"] = [int(x) for x in outfalls]

    print("\n\nThe indices of the points which you want to designate as overflow points:\n\
(Positive integers separate by space, for example: 23 65 118)\n")
    overflows = input("Overflows points indices: ").split()
    settings["overflows"] = [int(x) for x in overflows]

    print("\n\nThe minimum depth below the ground at which conduits can be installed:\n\
(Should be a positive integer or decimal number, for example: 1.1)\n")
    settings["min_depth"] = float(input("Minimum installation depth [m]: "))

    print("\n\nEnter the required minimum slope for the conduits:\n\
(Should be a positive decimal number, for example: 0.002)\n")
    settings["min_slope"] = float(input("Minimum slope [m/m]: "))

    print("\n\nDo you want to enter a maximum allowable slope as well?")
    choice = yes_no_choice()

    if choice == "y":
        print("\n\nMaximum slope should always be larger than the minimum slope\n")
        settings["max_slope"] = float(input("Maximum slope [m/m]: "))

    print("\n\nEnter the peak rainfall value for the design storm:\n\
(Should be a positive integer, for example: 23)\n")
    settings["peak_rain"] = int(input("The peak rainfall value [mm/h]: ")) / 0.36

    print("\n\nThe average percentage of impervious ground coverage of the area:\n\
(Should be a positive integer number between 0 and 100, for example: 25)\n")
    settings["perc_inp"] = int(input("Percentage of impervious ground [%]: "))

    print("\n\nA list of the available diameters of the conduits:\n\
(Should be a series of number separated by spaces, for example: 150 300 500 1000)\n")
    diam_list = input("List of available diameters [mm]: ").split()
    settings["diam_list"] = [int(x) / 1000 for x in diam_list]

    return settings


def step_3_input():
    """Create the explanations and input space for the SWMM file creation step of the
        software
    """

    settings = {}

    print("\n\nIf you are satisfied with the system that has been constructed,\n\
you can convert it into a System Water Management Model (SWMM) file. To do this,\n\
please give some final specifications:")

    print("\n\nA timeseries will be created from your given design storm value.\n\
Please specify the duration of this design storm in whole hours (for example: 2, max 12)\n")
    settings["duration"] = int(input("Design storm duration [hours]: "))

    print("\n\nA name for the SWMM file. This file will be a .txt file.\n\
The filename cannot contain any spaces or quotes (for example: test_file)\n")
    settings["filename"] = input("File name: ")

```

```python
    print("\n\nLastly, it is possible to show the subcatchment polygons in SWMM.\n\
Doing this for larger networks however may make the network difficult to view.\n\
Do you want to include the subcatchment polygons?\n")
    settings["polygons"] = yes_no_choice()

    print("\n\nThe file will now be created, and can be found in the main folder of \
APDUDS.\nPlease note, that in order to open this file in SWMM, you will need to select\n\
the 'all files' option in the folder explorer to be able to see the file in the directory.")

    print("\nThis concludes this use session of APDUDS, \
the software will close once the file has been created.")

    return settings


def tester():
    """Only used for testing purposes"""
    print("The terminal script has run")

if __name__ == "__main__":
    tester()
```