



Christian Lederhilger, BSc.

BIM in Geotechnics

-

Application to Road and Railway Construction

Master Thesis

To achieve the Degree

Master of Science

Submitted at the

Technische Universität Graz

Supervising Tutor

Ass.Prof. Dipl.-Ing. Dr.techn. Franz Tschuchnigg

Institute of Soil Mechanics, Foundation Engineering and Computational
Geotechnics

Technische Universität Graz

Graz, October 2018

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Arbeit identisch.

.....
Datum

.....
Unterschrift

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. This document is identical with the electronic version uploaded via TUGRAZonline.

.....
Datum

.....
Unterschrift

Kurzfassung

BIM in der Geotechnik – Anwendung im Straßen- und Eisenbahnbau

Das Hauptziel von Building Information Modeling ist es ein multi-dimensionales digitales Bauwerksmodell zu erschaffen, das alle projekt-relevanten Informationen vereint. Eine Schlüsselrolle in der Erstellung dieses digitalen Bauwerksmodells spielt das Ermöglichen eines effektiven Datenaustausches zwischen allen Projektbeteiligten und den Software-Anwendungen, die diese verwenden. Beim Planen von Straßen- und Eisenbahnprojekten stellen Verformungen, die während und nach dem Bau von Dammschnitten entstehen können eine nicht zu vernachlässigende Informationskomponente dar. Die vorliegende Masterarbeit beschreibt den Prozess der zum Ziel hatte den Informationsaustausch zwischen einer BIM-fähigen Verkehrswegeplanungs-Software und einer auf Geotechnik spezialisierten FE-Software zu erleichtern.

Abstract

BIM in Geotechnics - Application to Road and Railway Construction

The main goal in Building Information Modeling is to build a multi-dimensional digital building model that contains information from all project areas. A crucial part in getting to this digital representation of the building is to enable an effective exchange of information between people that are involved in the project and the software applications they use. In the planning of road and railway projects one component that cannot be neglected are the deformations that occur during and after the construction of an embankment. This master thesis describes the process that was followed in order to facilitate the information transfer between a BIM-capable traffic route engineering software and a FE-geotechnical software.

Acknowledgements

I would like to thank everyone that was involved in the development of this master thesis.

Special thanks go to Ass.Prof. Dipl.-Ing. Dr.techn. Franz Tschuchnigg for the dedication and guidance that he brought to the project. Moreover I want to thank MSc. Maria Westphal, Dipl.-Ing. Christoph Kellner and Marko Timic from Strabag for their instrumental support, as well as Dipl.-Ing. Jens Hoffmann for initializing the project in the first place.

I am also deeply grateful for the support that my family gave me during the writing of this thesis and entire course of my student career.

Table of Contents

1	Introduction: What is BIM?	1
2	BIM in Road and Railway Construction	3
3	Connection to Geotechnics	4
4	Project Definition	5
5	Evaluation of possibilities	7
5.1	ProVI Output Formats	7
5.2	Plaxis Import Tools	7
5.2.1	Plaxis Remote Scripting Interface	8
6	Development of Script	9
6.1	First Import Attempts	9
6.2	Further Analysis of Required Information and Explanation of Import Mechanisms	11
6.2.1	Geometry	11
6.2.2	Materials	15
6.2.3	Material Assignment	16
6.2.4	Additional Materials created by the Script	21
6.3	Additional Requirements	21
6.3.1	User Inputs	22
6.3.2	Stepwise Construction of Embankment	22
6.3.3	Geotechnical Measures	24
6.3.4	Automatic Phase Setup	31
6.3.5	Meshing and Mesh Refinement	39
6.4	Extracting the Calculation Results	40
6.4.1	Definition of Deformation Points	40
6.4.2	Writing Displaced 2D-Coordinates to a File	41
6.4.3	Export of Displaced 3D-Coordinates	42
6.5	Additional Problems occurring during the Script Development	45
6.5.1	Polygon Selection for Material Assignment and Mesh Refinement	45
6.5.2	User Inputs	47

6.5.3	Color Problem	47
6.6	Final Version of the Script	48
6.6.1	Intended Workflow	49
7	Alternative Script Version - Import of Material Properties	50
8	Visualization of Output Data in 3D	53
8.1	Analysis of Required Excess Material	55
9	Summary of the Script's Capabilities & Limitations	57
10	Conclusion	58
11	Literaturverzeichnis	59
Appendix A: User Manual		60
Appendix B: Source Code of Script		73

List of Figures

Figure 1:	Dimensions of BIM (modified: (Goger, et al., 2018)).....	1
Figure 2:	Communication Scheme between ProVI and Plaxis	6
Figure 3:	Basic Structure of Model Import	9
Figure 4:	Cross Section Modifications prior to Import	10
Figure 5:	Structure of “QP-file”	13
Figure 6:	Imported Sample Cross Section	15
Figure 7:	Imported Sample “BAUGRUND-file”	16
Figure 8:	Layer Thickness Modeling in ProVI.....	17
Figure 9:	Possible Forms of Layer Thickness Generation in Cross Section ...	19
Figure 10:	Flowchart of Material Assignment Process	19
Figure 11:	End Stadium of Material Assignment Process.....	20
Figure 12:	Definition of Reference Point for Embankment Sublayers	23
Figure 13:	Interpolated Boundary Points of First Sublayer.....	23
Figure 14:	Flowchart of Embankment Division.....	24
Figure 15:	Result of Embankment Division.....	24
Figure 16:	Boundary Conditions for Soil Replacement	25
Figure 17:	Inclination of Soil Replacement Slopes	26
Figure 18:	Boundary Conditions for Friction Bases.....	27
Figure 19:	Boundary Conditions for Drainage System	29
Figure 20:	Boundary Conditions for Overburden.....	30
Figure 21:	Bounding Box of Sample Polygon.....	32
Figure 22:	Phase Setup with added Safety and Consolidation Phases for each Sublayer	38
Figure 23:	Sample Setup for Mesh Refinement	39
Figure 24:	Sample Output Mesh.....	40
Figure 25:	Sketch of Result Point Interpolation	41
Figure 26:	Displaced Virgin Terrain in AutoCAD	42
Figure 27:	Point and Variable Definitions for Orientation Calculation	43
Figure 28:	Problem with wrongly assigned Materials.....	46
Figure 29:	Flowchart of Script.....	48
Figure 30:	Material Properties Import File.....	51
Figure 31:	Flowchart of Material Property Assignment Process	52
Figure 32:	Embankment on top of Virgin Terrain (without Displacements)	54
Figure 33:	Embankment Model with colored Contour Plot of Displacements u_y	54
Figure 34:	Contour Plot of Displacements u_y with Color Scale	55
Figure 35:	Displacements u_y along the Road Axis	55
Figure 36:	Folder Structure.....	60
Figure 37:	Extension of Cross_Section_File	61
Figure 38:	Extension of Subsoil_File	62
Figure 39:	Settings for Pegging_Data_File	62

Figure 40: Script Section: Password and Port Configuration	63
Figure 41: Configuration of Remote Scripting Server	63
Figure 42: Error-proofing of Inputs.....	65
Figure 43: Material Dialogue	65
Figure 44: Cross Section Input Loop	66
Figure 45: Construction and Consolidation Times Input.....	68
Figure 46: Mesh Dialogue	68
Figure 47: Phase Setup Scheme	69
Figure 48: Phase Input for Generation of Displacement File.....	70
Figure 49: AutoCAD Script File Name Convention	71
Figure 50: ProVI Import File Name Convention	71

List of Tables

Table 1:	Format of Material Test Import File	10
Table 2:	Advantages and Disadvantages of Import via DXF-Files	11
Table 3:	Advantages and Disadvantages of Import via “QP-file”	12
Table 4:	Key Parameters of “QP-file”	12
Table 5:	Format of Borehole Data Section in “BAUGRUND-file”	18
Table 6:	Unambiguous Sample Configuration of Material Assignment Data	20
Table 7:	Settings for Soil Replacement Phase	32
Table 8:	Settings for Friction Bases Phase	33
Table 9:	Settings for Drainage Construction Phase	34
Table 10:	Settings for Construction Phases	34
Table 11:	Settings for Consolidation Phase	35
Table 12:	Settings for Overburden Construction Phase	35
Table 13:	Settings for Overburden Consolidation Phase	35
Table 14:	Settings for Overburden Deconstruction Phase	35
Table 15:	Sample Phases Overview	37
Table 16:	Quadrant Classification based on Algebraic Signs of calculated Distances between Pegging Data Points	43
Table 17:	Description of Variables for Coordinate Transformation	44
Table 18:	Formulas for Quadrant-dependent Coordinate Transformation	44
Table 19:	Format of Output File for ProVI Import of 3D-Coordinates	45
Table 20:	Comparison of Required Excess Material for different Analysis Intervals	56

1 Introduction: What is BIM?

Since the digitalization started its advancement in the building sector, BIM, an acronym for Building Information Modeling, has become an often cited expression. Despite the frequent usage of the term, there is no definition for it that is universally agreed on.

The definition used by Austrian Standards can be translated as follows:

“Building Information Modeling (BIM) involves the optimized planning and construction of buildings with the help of suitable software. BIM is an intelligent digital building model that allows all persons that are involved in the project – from architect and client over technician to facility manager – to collectively work on this integral model and on its realization.” (Hirner, 2018)

Most descriptions of the term BIM emphasize that it has a wider meaning than only the digital model that is generated in the planning phase or the software that is used to do so. Building Information Modeling can also be understood as the whole innovative process that needs to be followed in order to arrive at the multi-dimensional digital building model. The key slogan that stands behind the planning with the BIM-method is “build digitally first”. This means that prior to the building’s construction; the whole construction process with all its necessary steps and sub-steps is already preempted by the digital building model. As a consequence it does not suffice for the model to include solely information about volumes and materials. Additionally, information regarding the time schedules of the construction needs to be also linked to the 3D-data. Thus, the model can be thought of as four-dimensional through the introduction of dates to it. By connecting the involved construction steps, volumes and materials to their costs, the building model can even be extended to five dimensions. (Goger, et al., 2018)

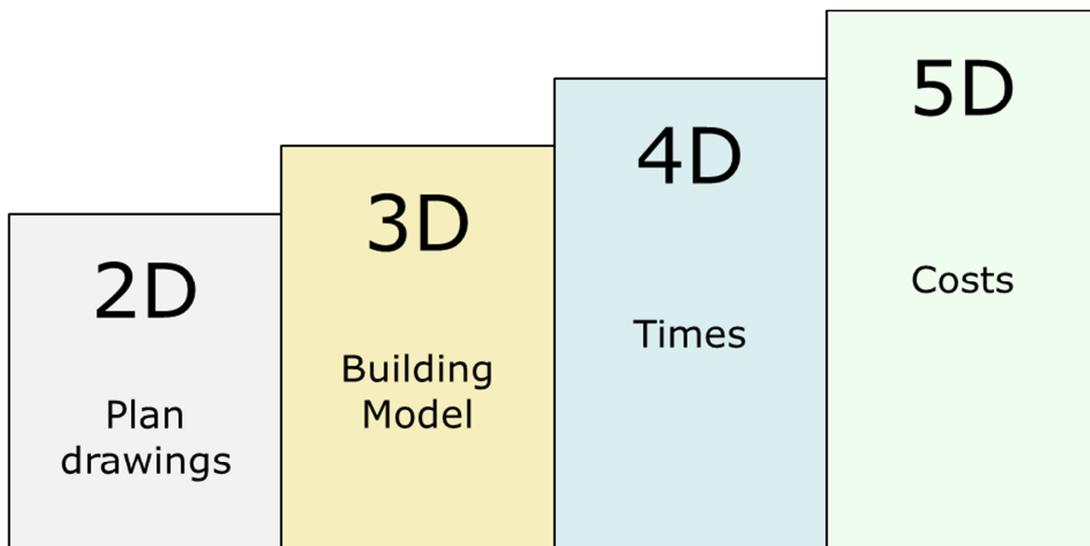


Figure 1: Dimensions of BIM (modified: (Goger, et al., 2018))

Resulting from the superordinate goal of BIM to create an integral digital building model, one aspect of great importance is the exchange of information between all parties that are involved in the project. (Borrmann, et al., 2015) In order to enable an efficient transfer process, between involved project parties and the different software packages they use, general data standards need to be established.

For the sector of infrastructure engineering, these data exchange standards, which would enable an efficient communication between different software tools, are often not yet available.

Specifically the lack of the ability of an efficient data transfer between a planning software for infrastructure projects and a geotechnical FE-analysis software was what sparked the idea that led to the development of this thesis – and the tool that was included in its scope.

2 BIM in Road and Railway Construction

In structural engineering the concept of Building Information Modeling already appeared in the early 2000s (Autodesk Inc., 2002) and the term was known even before that. In the field of infrastructure it took slightly longer for BIM to find usage in professional circles.

In 2015 the German Ministry for Traffic and digital Infrastructure (BMVI) released the so-called “Stufenplan Digitales Planen und Bauen”, which can roughly be translated as “Step-by-step Plan for Digital Planning and Construction”. The plan is structured in three phases. In the last stage, which starts in 2020, the plan prescribes that all infrastructure projects that fall under the field of responsibility of the BMVI are mandatory to be planned by using the BIM-method. (Doebler, 2018) Therefore BIM has started to increase its relevance in the field of infrastructure construction.

The overall goal of BIM is the same for structural design and infrastructure engineering: the establishment of one integral digital model that can be used to plan and analyze the entire lifecycle of a building. However, the boundary conditions for the usage of BIM-methodology are set differently for both of these engineering fields. In structural design the usage of BIM benefits from the geometric regularities that many building components have in common. Additionally, the building in terms of structural engineering forms one enclosed geometric entity that is not affected by surrounding boundary conditions as much as this is the case for infrastructure projects. Another factor that makes the application of BIM in infrastructure engineering – and especially in traffic route engineering – more difficult is the fact that the contractor is provided only seldom with digital data of the building models. In many cases it's the contractor's own responsibility to digitalize the data that has been provided in analog form by the client, which further complicates the establishment of a BIM-model. (Günther, et al., 2015)

Despite these complications in relation to the field of structural engineering, many companies have discovered that the market would stir towards the implementation of BIM in infrastructure projects. For example, the Austrian construction company Strabag SE began introducing BIM to their traffic route engineering sector in 2016. (Litsch, 2018)

In general, one can observe that the trend in traffic route engineering goes towards planning according to BIM methods. Resulting from the obvious connection that traffic routes have to the ground they are built on, this is also where one has to think of an effective inclusion of geotechnics to the BIM process.

3 Connection to Geotechnics

Due to various reasons – ranging from geomorphologic circumstances to limitations regarding the longitudinal gradient or logistics – there is often the need to construct roads and railways on artificial embankments. Since it is not uncommon for those embankments to reach lengths of hundreds of meters or even a few kilometers and heights of over ten meters, the involved earth movements can be quite significant.

However, when constructing dams of a few meters height on soils with limited stiffness, not only the primary earth movements need to be accounted for. High loads on soft soils lead to the occurrence of settlements and consolidation effects. Since in the construction of roads and railways the as-built height level needs to match the as-planned height level very accurately, those deformations need to be foreseen and counterbalanced with excess material.

Those excess material demands might quickly amass to quite significant numbers when we think of embankment sections with lengths of a few hundred meters. Knowing about the magnitude of those quantities therefore becomes of some importance regarding factors like costs and transport logistics.

Additionally, consolidation might take a long time to reach its final state, especially in soils with low permeability. Therefore, when planning roads or railways on embankments, consolidation times are an important aspect to consider.

When it comes to the planning and construction of an embankment project the geotechnical circumstances therefore not only influence material quantities, but also construction times and as direct consequence construction costs.

The inclusion of the occurring settlements and consolidation times is therefore necessary if one wants to establish an all-around 5D-BIM model of road projects. Currently there is no efficient way of transferring data between the software, where the road is planned, and the software, where the embankment's deformations are analyzed. The aim of the project that is described within this mater thesis is to enable the efficient inclusion of geotechnical calculation results in the BIM-model by facilitating the information transfer between both involved software tools.

4 Project Definition

Road and railway projects are generally planned using special software packages, which allow the consideration of many boundary conditions. While the ability to generate and import information about subsoil conditions like layering of soil materials is customary for that kind of software, it generally lacks the essential features to use the information from a geotechnical point of view to calculate settlements.

Embankment cross sections therefore need to be transferred from the planning software to the geotechnical software in order to calculate the occurring deformations. From there the deformations of embankment and subsoil need to be manually inserted into the planning software again.

In terms of BIM, where an all-around building model and an efficient data transfer between parties and softwares is the goal, this slow, manual data transfer is outdated and sought to be automatized.

The starting position for this is the following:

The planning software (in this case the AutoCAD-based ProVI from Obermeyer Planen + Beraten GmbH) has access to geotechnically relevant information under the whole construction area of a road project. For any given cross section of the embankment, this information includes:

- Geometry of the embankment
- Geometry of virgin terrain and subsoil layer boundaries
- Assignment of materials (defined through name and color) to those soil layers

The goal of this project is the automatized transfer of all those pieces of information from the planning software ProVI to a geotechnical software, where the embankment's displacements can be calculated – in this case the software Plaxis 2D.

Once the displacements that occur for any given cross section have been calculated in Plaxis – with all necessary intermediate steps to get there – those displacements should be transferred back to the planning software as efficiently as possible. As explained in Chapter 3, the displacements occurring as settlements and consolidation under an embankment influence quantities, times and costs. Therefore it is necessary to include this information in the road model of the planning software.

Figure 2 shows the intended flow of information between ProVI and Plaxis.

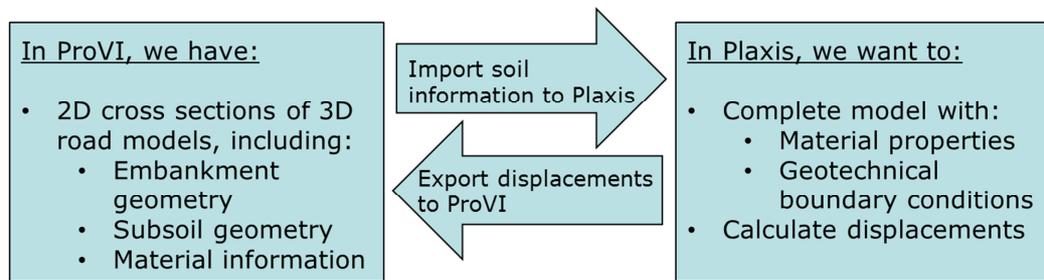


Figure 2: Communication Scheme between ProVI and Plaxis

The overall goal of the project can be summarized as enabling an elegant inclusion of three-dimensional subsoil deformation data in the BIM-models of traffic routes.

5 Evaluation of possibilities

Having worked out a clear project definition and knowing what information needs to be transported between the two applications ProVI and Plaxis, the next step was to analyze all possible ways to transfer the data.

Therefore ProVI was examined regarding supported output formats, while the focus was laid on available input formats in Plaxis.

5.1 ProVI Output Formats

Since ProVI is an AutoCAD-based application, it generally supports all formats that AutoCAD itself supports. Geometric data of cross sections can therefore easily be saved and transferred using the Drawing Interchange Format (DXF), or other similar standards.

However, seeing that there is not only the need to transfer geometric data but also information about materials and their assignment, the DXF-standard alone is not suitable for this kind of exchange. For this reason further examinations were carried out, that found that ProVI additionally supports the export of data via multiple tools. Besides the option to generate text based output files relying on XML or CTE type formats, there is also the possibility to write data to ASCII-files. Those ASCII text files have the advantage that they can be easily read by spreadsheet programs like Microsoft Excel. (OBERMEYER PLANEN + BERATEN GmbH, 2017)

Furthermore, ProVI also uses text based files for internal mechanisms. An example could be the creation of two-dimensional embankment cross sections from a three-dimensional road model. In order to get the actual cross section drawings, the user first has to generate a text based file called “QP-Datei”, which can contain coordinate and layer information for multiple cross sections. Only from this text file the user can then produce the actual drawings in AutoCAD. The memory location of these text files is in the ProVI working directory, to which the user has full access. Therefore it is also an option to use these native ProVI files for the data transfer to other applications.

5.2 Plaxis Import Tools

With Plaxis 2D 2017 access to the latest version of the program – at the point of writing – was provided. Additionally all features that are restricted to a Plaxis-VIP-license were also available to work with. Hence some features described here might not be available for all prior Plaxis versions.

The first option that comes to mind when trying to import data with Plaxis is the “Import Geometry”-tool, which is present in the structures mode of the

application. With it the user can import geometric data from external programs in Plaxis. Supported formats include (Plaxis BV, 2018):

- DXF: AutoCAD Interchange Format
- CSV: Comma Separated Value-files
- TXT: Text based files that are structured, according to a special convention
- Various other file formats of different software developers

While the import of geometric objects was tested and works generally fine with all of the mentioned formats above, the tool does not support the import of additional, non-geometric information.

So while the import of an embankment cross section available in the DXF-format would work quite fine for the geometry alone, an additional solution is needed for the read-in of material data.

5.2.1 Plaxis Remote Scripting Interface

The Plaxis Remote Scripting Interface provides the user with the possibility to write scripts in the programming language python and to run them in Plaxis. Through this python wrapper the user has the ability to use all features that are available in the Plaxis user interface. Furthermore it allows the usage of commands that are not even implemented in the GUI of Plaxis. Additionally the wrapper provides the opportunity to benefit from the full functionality of the programming language python (Plaxis BV, 2018). Python scripts for Plaxis are therefore a very powerful tool that allows the user to:

- Read-in text based files
- Analyze and modify the given information
- Build-up complete models for the Plaxis user interface
- Control meshing and refinement of the model
- Execute the calculation after a custom phase setup
- Analyze the data provided by the Plaxis Output module

All these capabilities make the Plaxis Remote Scripting Interface the perfect tool to import the geometric and material-related information from ProVI.

In order to use it, the user has to start the Remote Scripting Server directly in the Plaxis application. The SciTE editor, which is automatically installed simultaneously with Plaxis 2D 2017, can then be used to write custom user scripts. Those scripts can then be run directly from the SciTE-editor, provided the Plaxis Remote Scripting Server is active and the configurations for port and password are set correctly.

6 Development of Script

The planned working mechanism of data import via the script can be seen in the figure below:

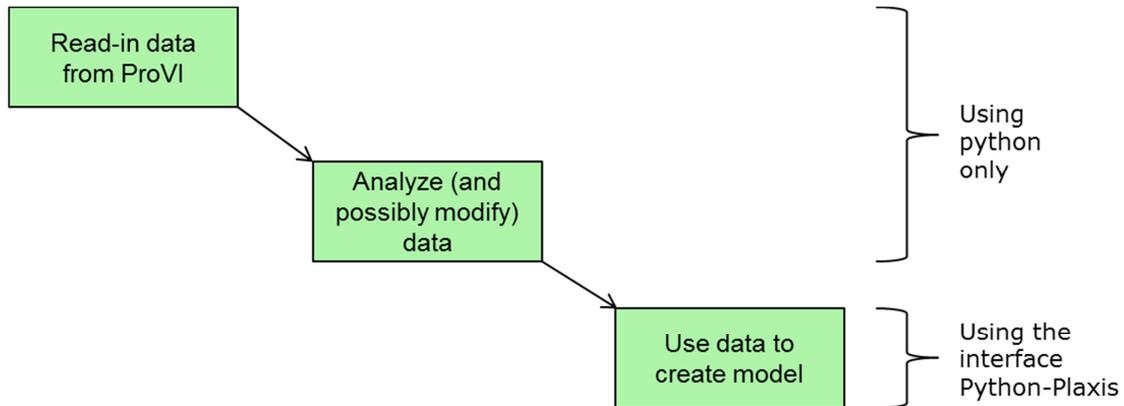


Figure 3: Basic Structure of Model Import

The development of the script was an iterative process that is described in the following chapters. The source code of the final script version can be found in Appendix B. Through the introduction of a color legend to the source code in the appendix it is attempted to roughly show, which code sections are responsible for the sub-tasks that are described in the following sections.

6.1 First Import Attempts

First tests to explore the capabilities of the Remote Scripting Interface started with the successful import of manually created embankment cross sections in DXF-formats. Therefore the “Import Geometry”-tool was used in Plaxis, only this time called by running a custom script via the Remote Scripting Server.

Moving on, the next step was to create cross sections of an actual road project in ProVI and trying to import them. However, cross sections generated in ProVI always come with textual data regarding heights and gradients directly included below the geometry of the cross section, as seen on the right-hand side of the figure below. When attempting to import a cross section model in Plaxis with the “Import Geometry”-tool, it is therefore necessary to first edit the geometry in AutoCAD. Firstly, all information that is not part of the embankment or the soilayers needs to be deleted, as seen on the left hand side of Figure 4. And even more importantly, the soil and embankment areas each need to explicitly form a closed boundary. This can be achieved via the AutoCAD command “Boundary” and by clicking in each of the closed polygons.

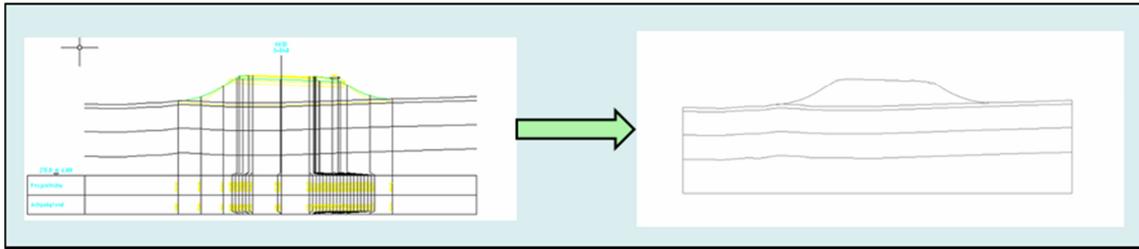


Figure 4: Cross Section Modifications prior to Import

Having now successfully transferred the geometric data of a road embankment cross section to Plaxis, the next objective was to import material related data. Since the format in which ProVI can produce this kind of material information was not known at this point, the first step was again to test the capabilities of the python wrapper by using a manually constructed file.

Therefore a CSV-type text file was created, having the format seen in

Table 1.

Table 1: Format of Material Test Import File

Material Number	Material Name	Material Color	Assigned to Polygon (No.)
1	Embankment	2238009	1
2	Topsoil	8841942	0
3	Clay	10283244	2
4	Sandy-Clay	16377283	3
5	Gravel	17820867	4

With the entries in each row separated by number signs (#) the data can easily be processed by the python script. Through calling commands in the Plaxis application those five soil materials are then also generated and visible in the user interface. The columns are defining the material names, colors (in RGB-format) and the number of the polygon in the Plaxis model to which the material should be assigned.

Already with this simple-structured input file some problems became apparent. Plaxis does not accept color input in RGB-format from the user interface. But this is not the case when setting multiple properties at once from either command line or remote scripting. The application then expects an entirely different number format that does not follow a scheme known to the user.

Additionally, the method of assigning materials to polygons based on their numbering in the Plaxis application proved to be quite suboptimal. Expectations were that polygons were numbered from top to bottom – or in any other traceable convention. But imports of multiple cross section geometries in DXF-formats showed that the numbering was different each time. Thus assigning materials to

regions solely by specifying the polygon's number soon proved to be an insufficient approach.

Although some complications occurred, those simple test imports generally demonstrated that with the usage of the python wrapper for Plaxis the data import can be designed very flexibly towards the user's needs. Moving forward therefore the next steps featured a further investigation of the information that needs to be imported.

6.2 Further Analysis of Required Information and Explanation of Import Mechanisms

In order to get a clear and thorough understanding of the data that needs to be imported, access to two complete road projects was granted by Strabag. Both of these projects have embankment sections with lengths of hundreds of meters. Therefore enough data was available to get a clear understanding of how the script needs to work. All mechanisms in the script were consequentially designed relying on the boundary conditions these projects share.

6.2.1 Geometry

Given that the aim of the project as a whole was, to make the deformation analysis of embankment cross sections as efficient as possible, the import via DXF-formats seemed far from optimal. The pre-processing that is necessary in AutoCAD prior to importing prevents an efficient data transfer. As a consequence it other options were investigated.

Of much interest became the so-called "QP-Datei" created in ProVI. As already mentioned briefly, the generation of cross section drawings in ProVI is achieved via the intermediate step of compiling this "QP-Datei". This text based file can contain – depending on the user-defined settings – cross sections in arbitrary intervals. For every cross section, the file then lists all lines that it is composed of. All lines in turn are defined via points that are specified in a local, cross section specific x-, y- coordinate system.

6.2.1.1 Comparison of Options

Table 2: Advantages and Disadvantages of Import via DXF-Files

Import via DXF-Files	
Advantages	Disadvantages
Easy modification of cross section geometry prior to import	Mandatory editing of cross section in AutoCAD prior to import
	Only one cross section at a time importable
	High time requirement

Table 3: Advantages and Disadvantages of Import via “QP-file”

Import via “QP-file”	
Advantages	Disadvantages
No editing of cross sections necessary prior to import	Custom modification of cross section geometry is more complicated
One file can contain multiple cross section geometries	
Highly time efficient when analyzing multiple models	

While at the beginning of the script-developing process both import via DXF-file and via “QP-file” was kept as a possibility, it became soon apparent that the second option is far superior. From the advantages and disadvantages listed in both tables above especially the convenience of zero versus a lot of necessary pre-processing tipped the scales towards the second option. At first the script still featured the possibility to import the geometry via DXF-format as well as via these text files. As development continued, it became obvious that the time efficient way of importing multiple cross sections via “QP-file” outweighed any negatives aspects by far. Therefore the idea of an import in DXF-format was completely abandoned. That way all resources could be invested in making the Import via “QP-file”-tool as efficient as possible.

6.2.1.2 Read-out of Geometry Data

The read-in of geometric data relies on an always constant format of the “QP-file”. An in-depth description of the structure of this text file can be found in the ProVI Benutzerhandbuch (2017). The script first virtually opens the text file and starts reading the data in it line-by-line. In order to analyze the data, it searches for the following key parameters:

Table 4: Key Parameters of “QP-file”

	Searching value	Position in line (columns)
Data type of line	66	1-2
Chainage of the cross section	Defined by user	4-18
Line number	21 (= virgin terrain) 42 (= subsoil layers) 51 (= earth filling)	20-23

Having identified the correct data type of a line, the script then looks for the chainage that the user wants to import. (Note: at this point of the development the value for chainage had to be defined directly within the script.) Having found the correct chainage, the script then searches for the line number 21, which forms the virgin terrain.

All these values are found in one text line. The next block following this text line then contains a list of point coordinates in x-, y- format. Each point is written in a separate line. The script then reads out the data of all these points and stores them in a list (in ascending order of x –coordinates) for later use. An exemplary data block can be seen below in Figure 5.

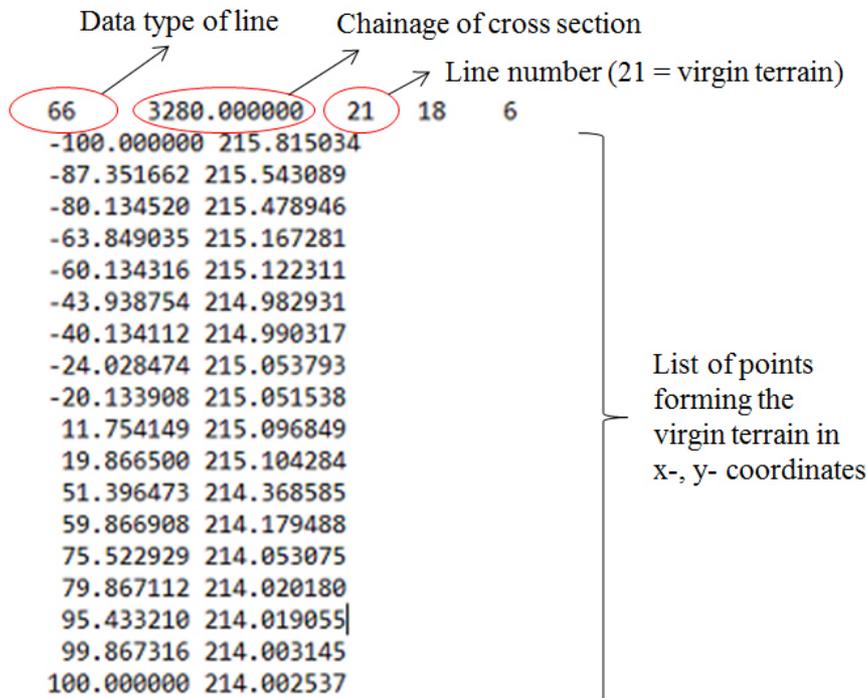


Figure 5: Structure of “QP-file”

After all entries for the line marked with number 21 have been read-out, the same process is done for the lines marked with the numbers 42 and 51. While the earth filling and the virgin terrain consist of only one line, the subsoil layers are generally formed by multiple lines. Each of these lines is therefore stored in a separate list.

6.2.1.3 Modification and Restructuring of Geometric Data

The way the model setup in Plaxis operates is through the creation of soil polygons. The polygons are created by using the command “polygon” and then specifying a list of point coordinates that form the polygon boundary. The order in which those points are given to the application is of great importance to the correct construction of the polygons. Only clockwise or counterclockwise organized boundary points will lead to the correct result. The data is therefore automatically reordered via the script. For example for the topsoil region, the polygon point list starts with the points of the uppermost subsoil boundary in ascending order. Once the end of that point list is reached, it is followed by the points forming the virgin terrain in descending order. Thus a strictly counterclockwise setup of polygon points is guaranteed, leading to the correct polygon setup.

Furthermore, as one can see in Figure 5, subsoil and virgin terrain cross section data generally ranges between -100 and 100 meters around the cross section axis. This range is unnecessarily wide in terms of deformation analysis with Plaxis, since no deformations will occur that far away from the embankment. Therefore polygons are vertically cut off at -50 and 50 meters, limiting the cross sections to 100 meters in width.

Another need for data modification comes specifically at the embankment area itself. The earth-filling-line that is listed in the “QP-file” sometimes contains a small part of soil replacement below virgin terrain. However the geotechnical measure of a soil replacement is implemented independently from the geometry import in the script – see Chapter 6.3.3.1. Therefore the geometry input is modified, removing the parts of earth filling that lies under the virgin terrain line.

Finally geometric data needs to be modified in order to eliminate data that would cause problems in the Plaxis application. When testing the geometry import on several cross sections it became apparent that Plaxis does not tolerate “dead ends” in its soil polygons. This means a polygon where a line goes from point A to point B and from there back to point A will cause the meshing procedure to fail. Since in the data delivered by the “QP-file” this constellation sometimes occurred it was necessary to delete one of those identical points. It is important to note that the overall geometry of the concerned polygon is not affected by this modification.

6.2.1.4 Result of the Geometry Import

A sample end product of the geometry input section in Plaxis might then look as shown in the figure below.

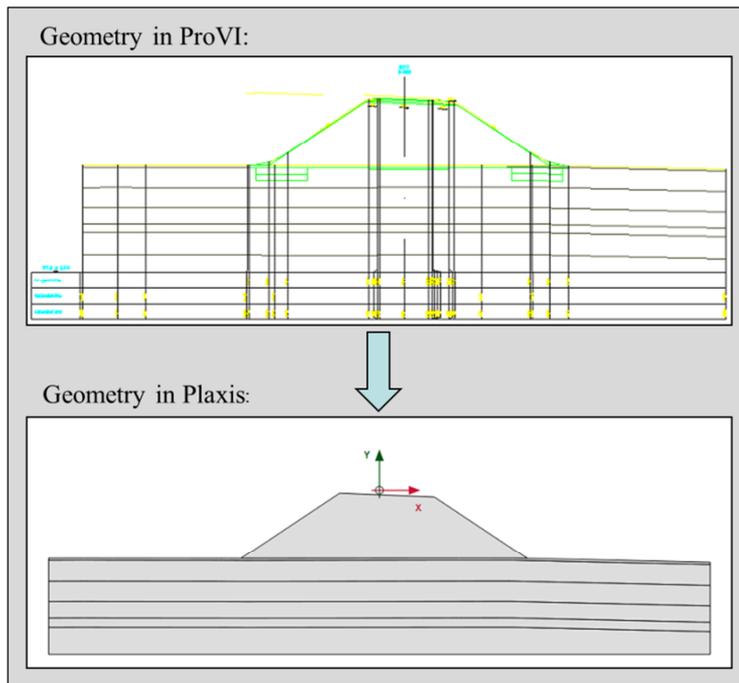


Figure 6: Imported Sample Cross Section

The upper part of the figure shows the geometry of a sample cross section with all data that is contained in the “QP-file”. The lower part shows the same cross section as seen after the import in Plaxis.

6.2.2 Materials

A first priority in the importing of materials was to find a suitable file in ProVI that contained all the relevant data. After a brief review it was found that the file called “BAUGRUND” (German for subsoil) by ProVI contains an overview of all materials that occur in the whole project. All data related to soil materials is stored in this file. Like the “QP-file” it can also be found in the working directory of ProVI and is a text-based file. In contrast to the “QP-file” though, the “BAUGRUND-file” is a CSV-type file. Every entry is separated from the next with a number sign. This makes it very easy to extract the data later in the python script. The upper part of Figure 7 shows a sample “BAUGRUND-file” as it occurs when opening it with a standard text editor from the working directory. Values of interest are here the material name (marked in red) and the material color (specified in R/G/B-format and marked in green). The material ID marked in light blue is later used for the assignment of materials to the soil polygons.

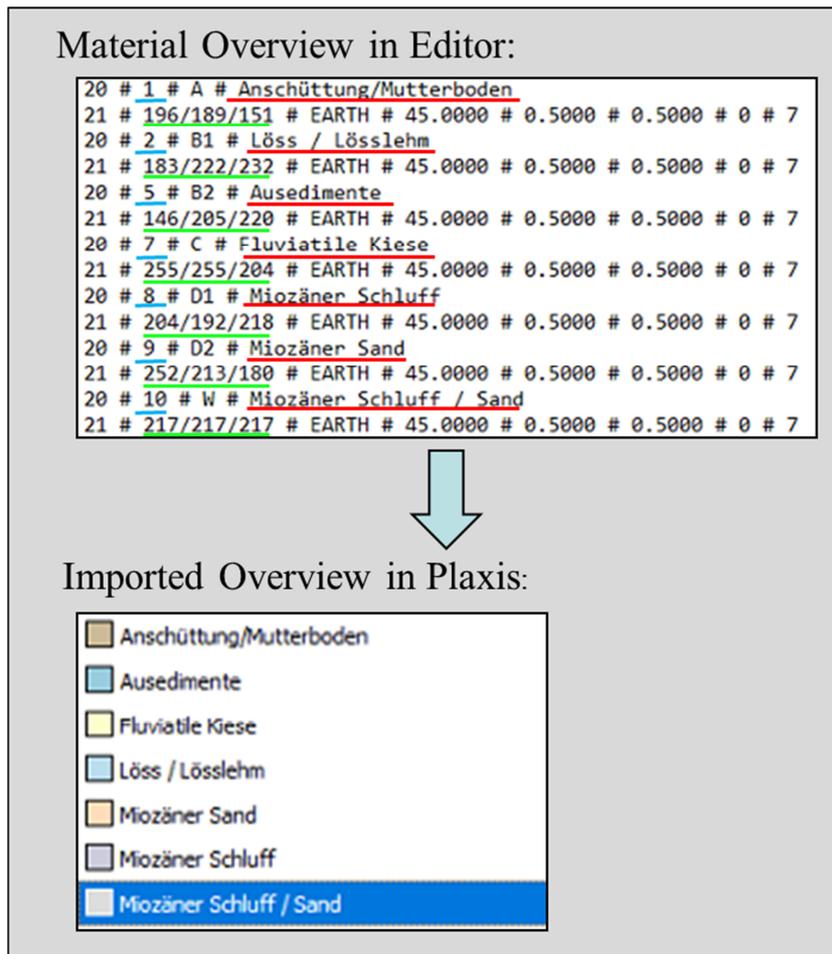


Figure 7: Imported Sample “BAUGRUND-file”

Those two properties are extracted by the script. For each material in the list, the script creates a soil material in Plaxis. The lower part of Figure 7 shows the material overview that was generated from the text file above, as seen in Plaxis.

6.2.3 Material Assignment

Having created the possibility to import the geometry of cross sections and an overview of all soil materials occurring in the project, the next task was to assign the materials to the right polygons. Analyzing the data in ProVI related to material assignment showed that the information cannot be extracted as easily as for the previous sections. While geometry and material overview could both be imported via the use of one text file and without much follow-up data analysis, this proved to be more difficult for material assignment. Of all the files that ProVI can produce none clearly states the information which material has to be assigned to which cross section. While earth fill and virgin terrain each have their own layer number in the “QP-file”, the subsoil materials are all bundled on one line number there.

The solution for the material assignment problem therefore necessitated a more in-depth analysis concerning the provided data. This was achieved by designing a

special process to remodel the method of material assignment in ProVI through the usage of borehole data.

6.2.3.1 Explanation of Material Assignment Process

First of all it is important to understand the way thicknesses of soil layers in cross sections are generated in ProVI. ProVI calculates the thickness of layers in cross sections by linearly interpolating the layer thicknesses in the boreholes before and after the cross section in longitudinal direction, as sketched in Figure 8. BH_left stands for the borehole to the left of the cross section, BH_right stands for the borehole to the right of the cross section.

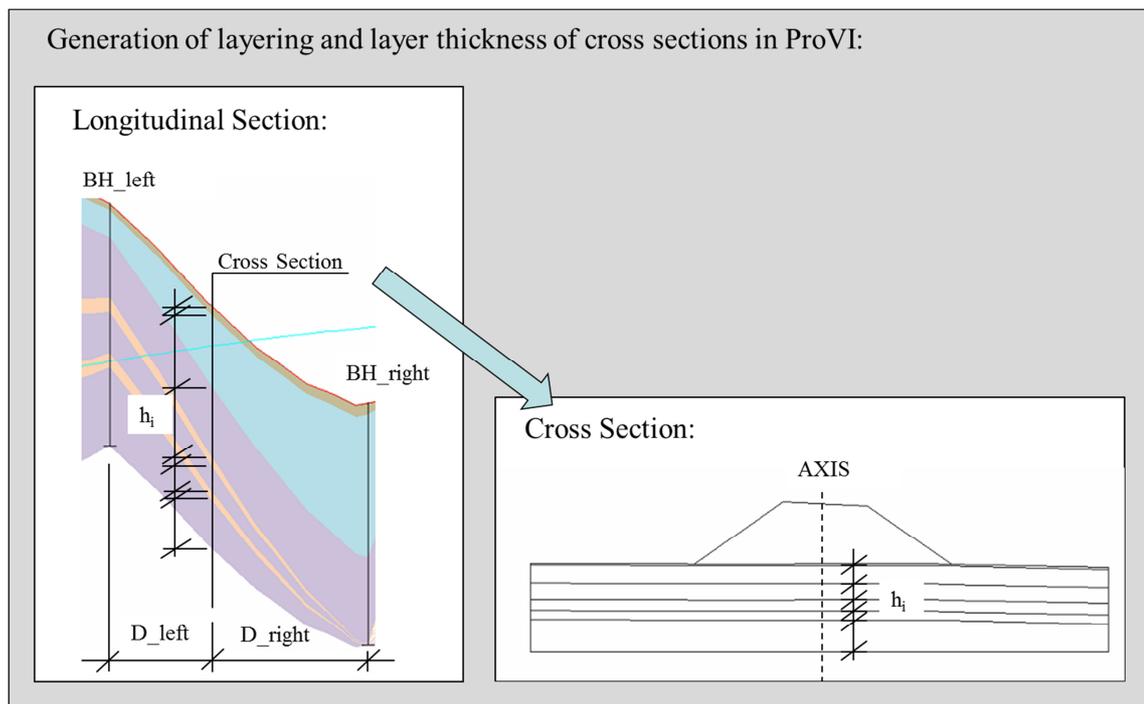


Figure 8: Layer Thickness Modeling in ProVI

When studying the longitudinal section of Figure 8 one can note three options possible options, how each layer of a cross section can be defined.

- Option A: The layer material occurs only in the borehole to the left of the cross section.
- Option B: The layer material occurs only in the borehole to the right of the cross section.
- Option C: The layer material occurs in both boreholes.

Note: Since the algorithm with which ProVI builds up the layer model from the borehole data is not known, the script was designed to investigate all three of these possibilities to decide which material belongs to which layer.

All borehole data related to the project is imported to the script. Conveniently, ProVI saves this data in the second section of the "BAUGRUND-file" that has

already been imported. The format of this borehole data section can be seen in the table below:

Table 5: Format of Borehole Data Section in “BAUGRUND-file”

Borehole Number A	Borehole Chainage A [m]
Material ID A1 [m]	Layer Thickness A1 [m]
Material ID A2 [m]	Layer Thickness A2 [m]
...	...
Material ID An [m]	Layer Thickness An [m]
Borehole Number B	Borehole Chainage B [m]
Material ID B1 [m]	Layer Thickness B1 [m]
Material ID B2 [m]	Layer Thickness B2 [m]
...	...
Material ID Bn [m]	Layer Thickness Bn [m]

It is important to note that it is possible for one material to occur more than once in one borehole as seen in the longitudinal section shown in Figure 8. That means for example the material with the ID “8” can occur multiple times in one borehole.

Having stored the borehole data in easy accessible lists, the script then analyzes it. Since the chainage of the imported cross section is known, the script can find the two boreholes closest to this chainage. It then computes the distances between the chainages of:

- BH_left and the cross section, called D_left from now on, and
- BH_right and the cross section, called D_right from now on.

Having gathered the information about these distances as well as the thicknesses of soilayers in both neighboring boreholes, the assignment process can start. Beginning with the top entry (for layer thicknesses) of both left and right borehole the script interpolates three possible thicknesses for the corresponding layer in the cross section. The three possibilities of how the layer thickness in cross sections can be formed can be seen in the sketch below:

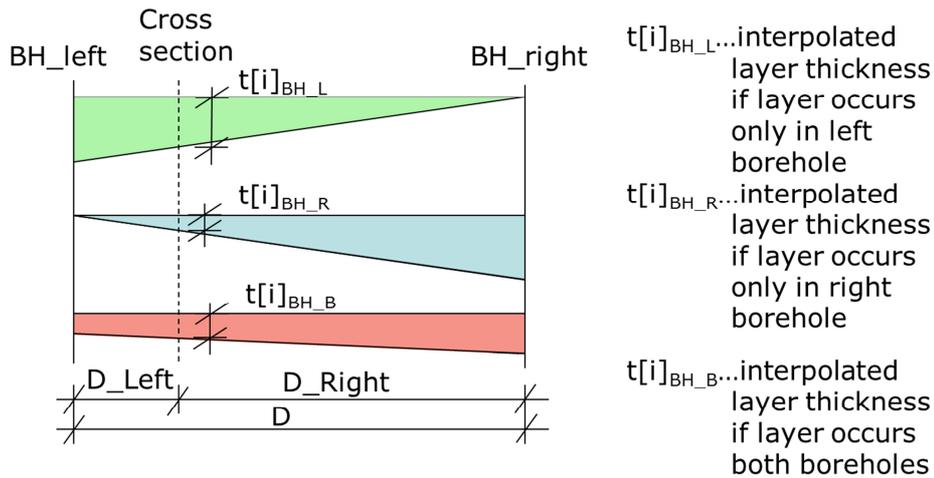


Figure 9: Possible Forms of Layer Thickness Generation in Cross Section

The calculated layer thickness (of these three options: $t[i]_{BH_L}$, $t[i]_{BH_R}$, $t[i]_{BH_B}$) that matches the actual layer thickness in the model ($t[i]_{model}$) is therefore proved to be the correct one – the same as in the ProVI model. The material with the corresponding ID gets then assigned to that soil polygon.

To make this process clearer, the flowchart of how this material assignment process is solved in the script can be seen in Figure 10.

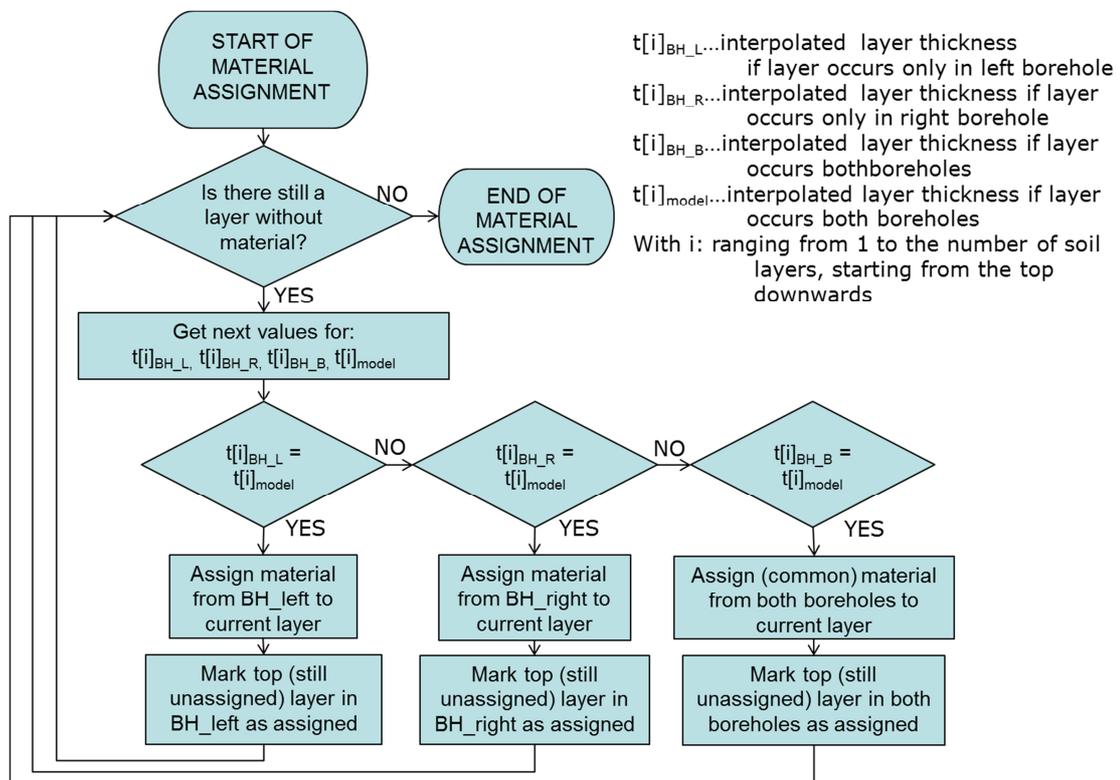


Figure 10: Flowchart of Material Assignment Process

A sample end stadium of the assignment process as seen in Plaxis can be studied in Figure 11. Every subsoil layer has been assigned its corresponding material data from the “BAUGRUND-file”. Since that file defines no material for the dam

area, an extra material called “Embankment” is created and assigned to that polygon.

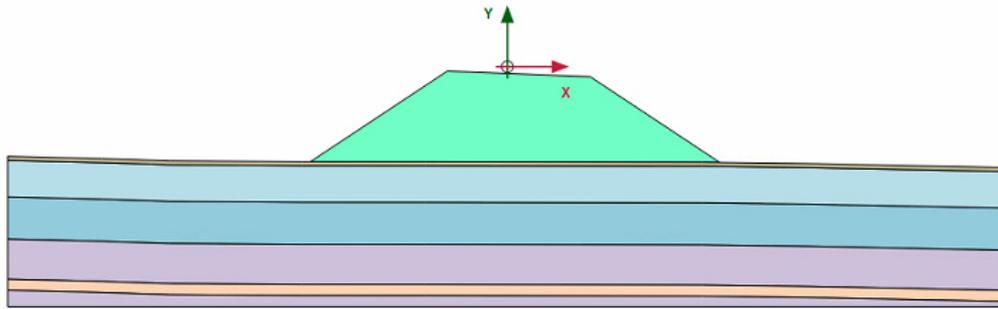


Figure 11: End Stadium of Material Assignment Process

6.2.3.2 Discussion on Uniqueness and Correctness of Material Assignment

Considering the approach to the material assignment process, one might point out that the chosen assignment method might not always lead to the correct result. For a very specific configuration of borehole data and cross section chainage materials could then be assigned to the wrong soil layers. A sample configuration, where this would be the case can be seen in the table below.

Table 6: Unambiguous Sample Configuration of Material Assignment Data

Cross section data	
Cross section chainage:	3200.0 [m]
Layer 1 – thickness in cross section	0.75 [m]
Borehole data:	
Chainage of BH_left	3100.0 [m]
Chainage of BH_right	3500.0 [m]
Material 1 - thickness in BH_left	1.0 [m]
Material 2 - thickness in BH_right	3.0 [m]

When interpolating the possible thicknesses of Layer 1 in the cross section, one comes up with the following values:

$$t_{1BH_L} = 1.0 - \frac{(1.0 - 0)}{(3500 - 3100)} \times (3200 - 3100) = 0.75[\text{m}] \quad (1)$$

$$t_{1BH_R} = 3.0 - \frac{(3.0 - 0)}{(3500 - 3100)} \times (3500 - 3200) = 0.75[\text{m}] \quad (2)$$

$$t_{1model} = 0.75[\text{m}] \quad (3)$$

Due to the structure of the material assignment process – Layer 1 in this cross section is assigned Material 1 from the left borehole. As seen in the process flowchart in Figure 10, the script first checks whether the interpolated thickness from the left borehole matches exactly the layer thickness in the cross section. Since this is the case, Material 1 gets assigned to Layer 1. However this might be the wrong assignment, seeing that t_{1BH-R} also exactly matches t_{1model} . If this is the case, this will lead to lower layers getting assigned no material at all, since the model layer thickness will then no longer match the interpolated thicknesses from the borehole data. This model could therefore not be used for a full automatic import.

However, the configuration seen above is only a very theoretical possibility. The borehole data of the projects that were analyzed for the development of this script had their chainages always defined to at least three but up to six decimal places. With this highly precise chainage definition, it becomes extremely unlikely that more than one interpolated thickness exactly match the actual thickness in the model. With more than one hundred cross sections that have been tested with the script this problem has never occurred and is therefore considered a purely theoretical possibility. Consequentially no further steps were taken in order to eliminate this problem.

6.2.4 Additional Materials created by the Script

The “BAUGRUND-file” imported from ProVI only lists subsoil materials. On a basic level the model needs at least one additional material for the earth fill of the embankment. Since some later sections of the script and the embankment models in general need further materials in order to accurately represent the reality, the script is already at this point used to create a list of extra materials – each one of them for a special purpose. The list includes:

- Embankment Material –as mentioned briefly in 6.2.3.1
- Soil Replacement Material
- Drainage Layer Material
- Friction Base Material
- Five empty material sets that the user can modify to include custom materials

6.3 Additional Requirements

With the import of geometry and materials and their correct assignment to soil polygons, the script is at this point already capable of the basic functions that it was designed for. In order to make it a more efficient tool for the import of embankment cross sections, it was agreed to extend the script’s scope of

operation by including some additional tools. In accordance with Strabag the inclusion of the following features was aimed at:

- Increasing the script's user-friendliness with the introduction of user inputs
- An automatic division of the embankment in sublayers and their stepwise construction
- A list of geotechnical measures, which the script's user can arbitrarily apply to cross sections. Those measures include:
 - Soil replacement
 - Friction bases
 - Drainage system
 - Temporary overburden
- The automatic setup of calculation phases
- Automatic meshing and refinement of the mesh

The in-depth information how these features were implemented can be found in the following chapters.

6.3.1 User Inputs

Looking at the way that geometry is imported via the script, the introduction of user inputs to the program became of great interest. With one "QP-file" storing multiple cross sections, it is necessary to allow the user to specify which cross sections he wants to import to Plaxis. Therefore the SciTE-Editor in which the script was written (and is run), provides a console part, from which textual user inputs can be directly transferred to the script. Originally defined as "string", the script can then change the input's data type to a numerical format. This makes offers the possibility to search the "QP-file" for the same number (in this case the chainage) and import the requested cross section – if present.

While originally user inputs were thought to be only used for this definition of the cross section chainage, it was later decided to increase the script's flexibility by allowing inputs in more areas, as explained in the following chapters.

6.3.2 Stepwise Construction of Embankment

In accordance with Strabag it was defined that embankments should be divided in sublayers. This way the actual construction process of the dam can be accurately remodeled with Plaxis. The height of each sublayer was specified with two meters. Regardless of the inclination of the virgin terrain, the sublayers are modeled always horizontally. The reference point for the sublayer generation is defined at the intersection of the embankment axis ($x=0$) with the virgin terrain, as seen in Figure 12.

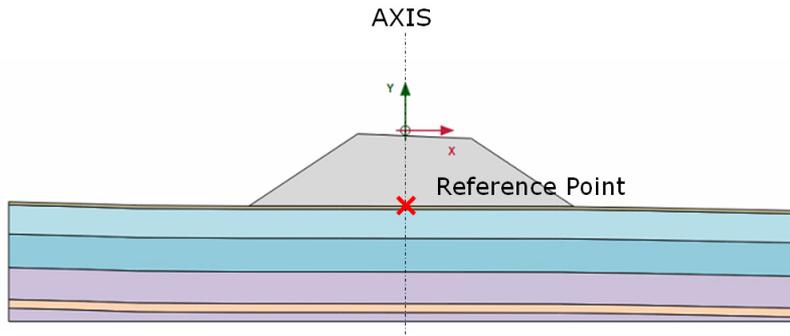


Figure 12: Definition of Reference Point for Embankment Sublayers

The x-coordinate of this point is defined with zero by its position on the dam axis. The y-coordinate is extracted by the script through analyzing the point coordinates of the virgin terrain-line and linearly interpolating the value for $x=0$ using the two neighboring line points.

Adding two meters to this y-coordinate then gives the y-value for the horizontal upper boundary line of the first sublayer. The script then analyzes the list which consists of all points forming the boundary of the embankment polygon. It starts from the point with the lowest y-coordinate. From there it checks the following points both clockwise and counter-clockwise. It adds all points to a newly generated sublayer list until the target y-value has been reached. There, it then linearly interpolates the x-values of both the left and right point that lie on the embankment boundary line. This relationship is visualized in Figure 13.

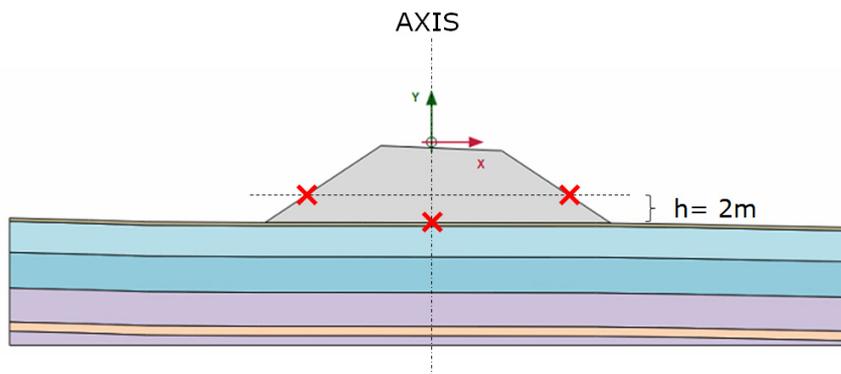


Figure 13: Interpolated Boundary Points of First Sublayer

Those two points are then also added to the point list of the first sublayer. By calling the command “polygon” in Plaxis and passing the coordinates of all boundary points, Plaxis then forms the first sublayer polygon in the model.

This process is now repeated by adding again two meters to the top y-coordinate of the previous sublayer, and forming again the next list of sublayer coordinates. This is done until this target y-value is higher than any point of the embankment polygon, thus indicating that the top of the embankment has been reached. The script then adds all embankment boundary points that are still left and forms the top subpolygon with these points. The flowchart of how this process of dividing

the embankment in sublayers is implemented in the script can be studied in Figure 14. The usage of the plural form for points and coordinates - in the text labels marked in green - indicates that here both points following clockwise and counterclockwise are concerned.

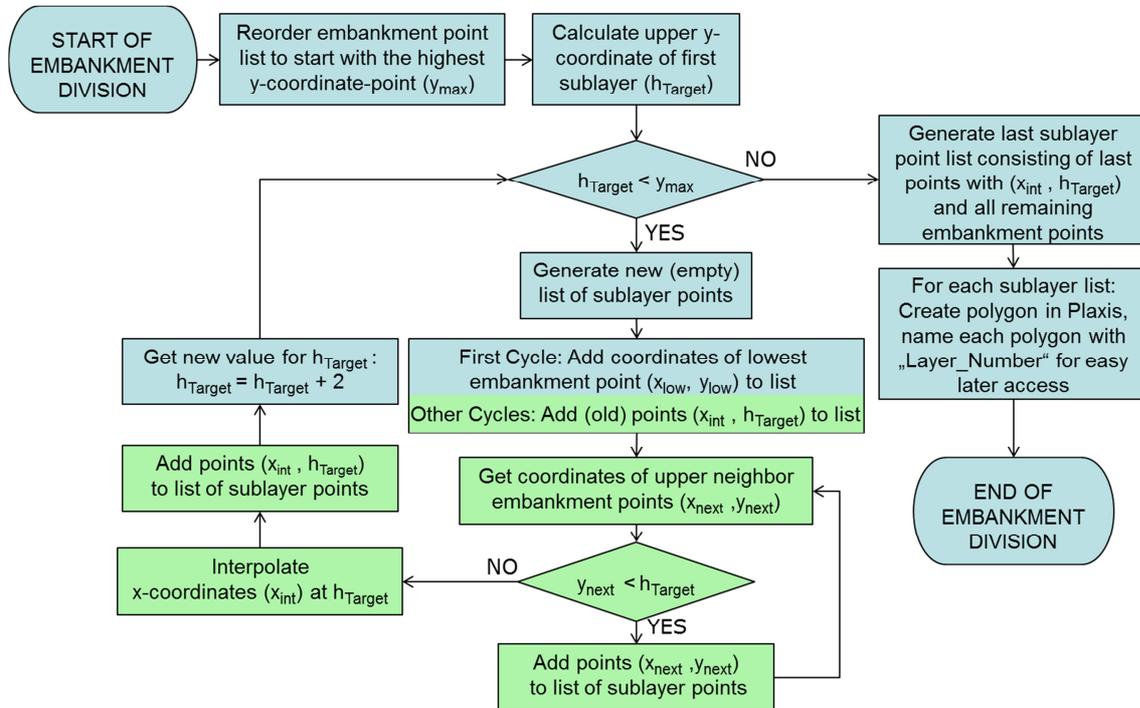


Figure 14: Flowchart of Embankment Division

A sample result of this division process can be seen in the figure below. While all lower layers have a height of two meters, this is generally not the case for the top layer. This one just has the height of the rest of the embankment even if this might be only a few centimeters and not over the whole crown area of the dam.

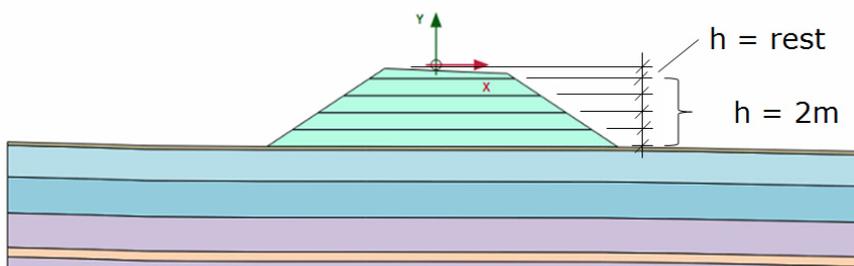


Figure 15: Result of Embankment Division

6.3.3 Geotechnical Measures

In order to be able to accurately model the actual construction and deformation processes of a road embankment, it was agreed to add a number of geotechnical measures to the script. The goals of these measures range from reducing deformations, over increasing stability to an acceleration of settlements and consolidation. With this automatic modeling one cannot achieve the same

flexibility as with manual modeling directly in Plaxis. But by enabling the user to define key model values via the script console, very quick model setup while also maintaining a high degree of flexibility is possible.

6.3.3.1 Soil Replacement

The replacement of soil under the road embankment is a geotechnical measure that aims at increasing the stability of the structure. It finds application in areas, with soft soils, that would lead to the occurrence of high deformations under the embankment area. Therefore soil with disadvantageous properties is removed and replaced by a material that provides increased stability. The depth of soil replacement may range from a few decimeters to more than one meter. For the purposes of this script it is assumed to be always constructed over the full width of the embankment.

As reference points for the geometric setup of soil replacement serve the two embankment toe points. The script identifies those two points as the embankment points with the smallest and the highest x-coordinate. The thickness has to be specified by the user and is added to both of these points vertically downward as seen below.

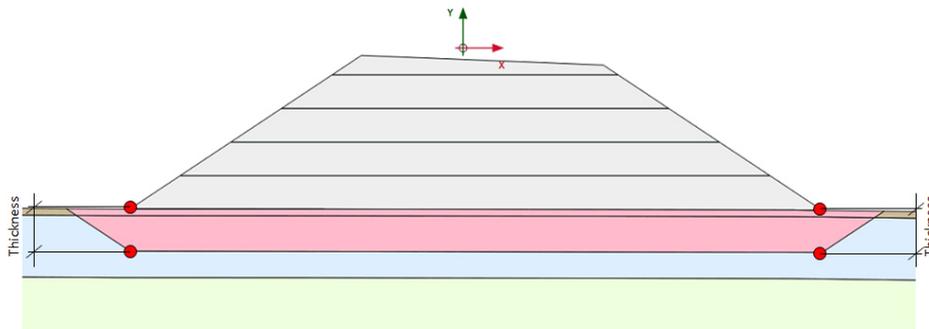


Figure 16: Boundary Conditions for Soil Replacement

A straight line connecting those two points below the embankment toes serves as the bottom boundary of the soil replacement. The slopes of the soil replacement excavation can also be specified by the user as a ratio in the format delta horizontal to delta vertical (e.g. 1.5/1) as seen in Figure 17.

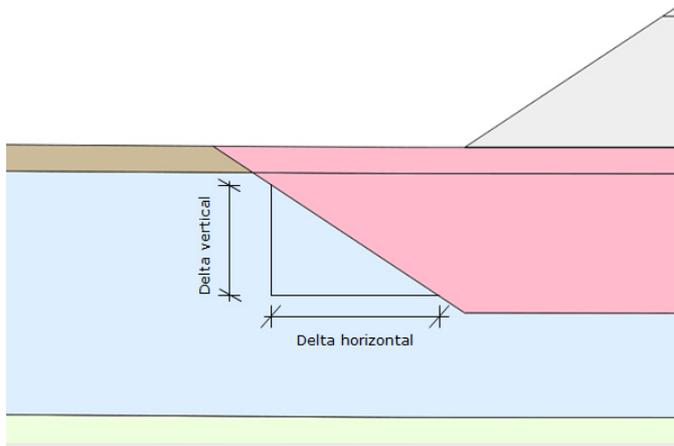


Figure 17: Inclination of Soil Replacement Slopes

With the specification of both thickness and inclination of slopes, the script has gathered all needed information. The geometry of the soil replacement is completed with the inclined slopes directly following after the base line. On the Plaxis-side the lines are generated with the command “line” and the specification of starting and end point coordinates. For the two upper points of the slope, the highest model y-coordinate is defined as their y-coordinate in order to make sure, the line definitely crosses the virgin terrain. The corresponding x-coordinate is then calculated in order to fit the given inclination.

6.3.3.2 Friction Bases

Similarly to the soil replacement, the aim of friction bases is to provide a stable foundation for the embankment. But while soil replacement is usually carried out over the full embankment width, friction bases are only a local measure at the embankment toe points. The working principle is here to increase the shear stability in the area surrounding the toe point of the dam. This is achieved by replacing the subsoil with materials with higher shear resistances like gravel or edged rocks.

Since the geometric boundary conditions are the same for both the right and the left friction base, the explanation is done only for one side. The properties the user has to specify for the friction bases are:

- Base length of the friction base
- Portion of the base length „outside” of the embankment area
- Thickness of the friction base
- Inclination of the friction base slopes

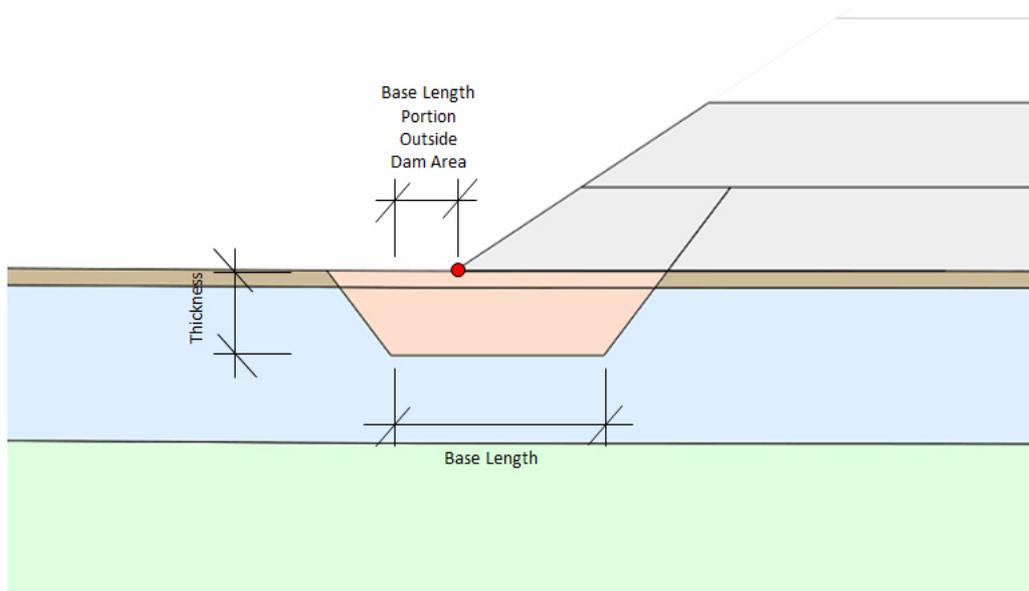


Figure 18: Boundary Conditions for Friction Bases

As reference points serve again the embankment toe points – again identified by the script as the embankment points with the minimum and maximum x-coordinates. The base line is always modeled horizontally, its horizontal position specified through its portion „outside” of the embankment area. The constant y-coordinate of the base line is defined as the toe’s y-coordinate minus the given thickness. Following the end points of the base length, the slopes are defined the same way as for the soil replacement – seen in Figure 17. The upper end points of the slope lines are again set specifically to ensure a flawless setup. Therefore the slope line to the outside of the embankment area is again only ended at the model’s highest y-coordinate. To the inside of the dam, the slope line has its ending point just below the first two-meter-sublayer line. This way it is certain that the virgin terrain line is pierced, while the first sublayer line is not. Thus no additional polygon is created in the model, which would lead to problems in later sections of the script, when setting up the stepwise embankment construction. An example for this geometric relationship can be studied for the left friction base in Figure 18.

6.3.3.3 Drainage System

While soil replacement and friction bases both aim at a reduction of occurring deformations, the aim of drainage lies more in the area of accelerating the occurring deformation. This way the final state of displacements is reached quicker which is beneficial in reducing construction times. The working principle of this measure is to increase the vertical permeability of the subsoil. Therefore excess pore water pressures induced by the embankment construction can be reduced faster. Thus the effects of consolidation are accelerated and following construction steps that rely on the deformations to have already occurred can start sooner. This measure is utilized in subsoils with low natural permeability

like for example clays. Here the construction of vertical drains has great effect in increasing the permeability. Depending on the permeability properties of the embankment fill material the drainage system can either be constructed with or without a separate drainage layer at the bottom of the dam.

The properties the user has to specify for the drainage system are:

- Thickness of the drainage layer in the range from 0 (meaning no drainage layer) to two meters
- Spacing of the vertical drains
- Length of the vertical drains

The optional drainage layer is always modeled with a horizontal surface. The thickness is therefore measured from the point where the dam axis intersects the ground surface. However, since the drainage layer is supposed to cover the whole embankment area, the thickness is automatically adjusted to a high enough value in case the horizontal drainage layer would intersect the ground surface before reaching the slopes of the dam. The first sublayer of the rest of the embankment is constructed from the top of the drainage layer. In this case the height of the first layer is not modeled with a thickness of two meters. Instead this value is selected with two meters minus the height of the drainage layer, so that the two layers combined reach a height of two meters again.

In case the embankment has sufficiently high permeability and the drainage layer is therefore omitted, the user can just enter „0” as layer thickness. Then there is no longer a horizontal drainage layer, the drains then all start from ground surface level.

From the entered value for drain spacing the script then automatically computes the number of drains. All drains are furthermore evenly distributed with leaving the same remaining length drain-free (marked red in Figure 19) on the outside of the embankment.

The length of drains is then measured from the top y-Coordinate of the drain downward. If a drainage layer is included all drains have their top point at the same height (the top of the drainage layer) and also share the same y-Coordinate for their bottom ends. If the drainage layer is omitted this is no longer the case since all drains start at ground level, at generally varying elevations.

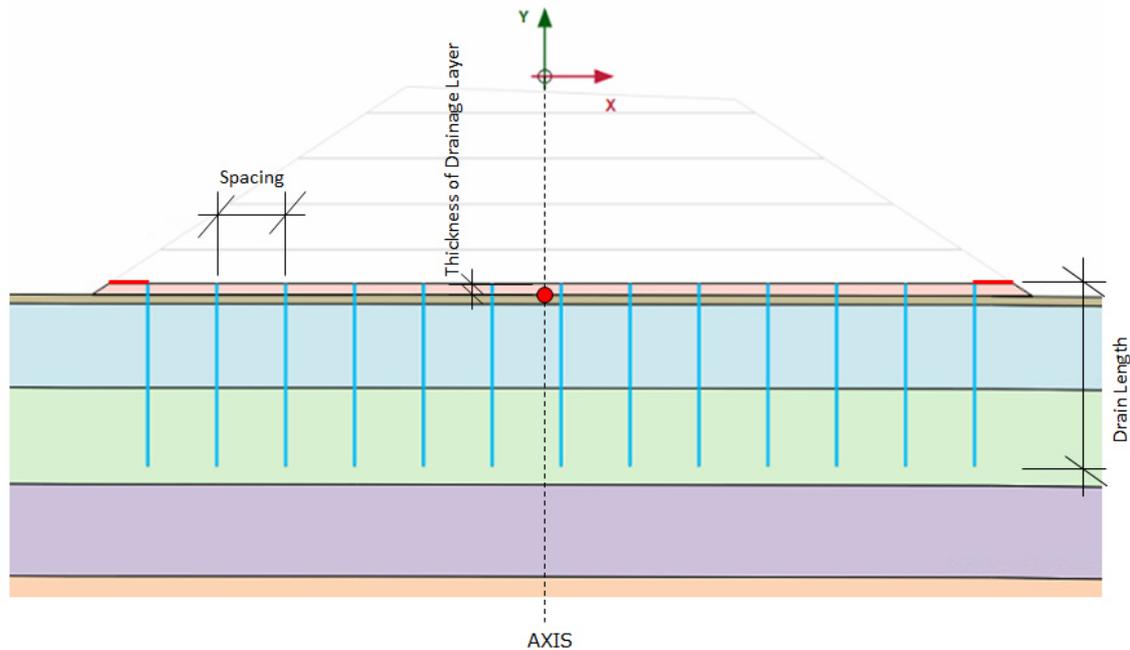


Figure 19: Boundary Conditions for Drainage System

On the script-side of this measure, drains are constructed as straight vertical lines using the Plaxis-command “drain”. The start and end point coordinates this command requires, are calculated by the script according to the boundary conditions described and seen above. Due to safety reasons the bottom y-coordinates of the drains are checked in regard to their absolute height. In case the drain length was selected to long by the user for a specific cross section, the script will require the user to change the input. Vertical drains exceeding the bottom boundary of the model would cause the calculation in Plaxis to fail.

6.3.3.4 Temporary Overburden

The construction of a temporary overburden on top of the final earth fill is a measure that also aims at accelerating the occurrence of deformations. By applying a load higher than the one at the end of the construction process, settlements and consolidation effects take less time to reach its final state. As for the drainage, this can be useful in saving overall construction time in case other steps rely on these deformations to have already taken place.

The properties the user has to specify for the temporary overburden are:

- Thickness of the overburden
- Construction time of the overburden
- Time the overburden is left to consolidate until the deconstruction

The geometric setup of the temporary overburden requires the script to automatically identify the two dam shoulder points (the two upper points marked in red in the drawing below). This is done by analyzing the angles that occur on the embankment surface. Starting from the embankment point with the highest y-

Coordinate all angles to subsequent points are calculated both clockwise and counterclockwise. Once an inclination of the embankment surface higher than 25 degrees has been detected, thus indicating that the embankment slope has been reached the script recognizes that point as one dam shoulder point. The same procedure is followed for the other shoulder point at the opposing side.

Having identified those two shoulder points, the overburden geometry is defined by intersecting the continuation of the two dam slope lines with a – possibly imaginary – connection between both dam shoulder points in the distance of the specified thickness. This connection line might be imaginary, because the embankment top does not have to be a straight line between both shoulder points. The top line of the overburden is therefore not necessarily parallel to the top line of the embankment. But – caused by these geometric boundary conditions – it is always parallel to the connection line of the dam shoulder points.

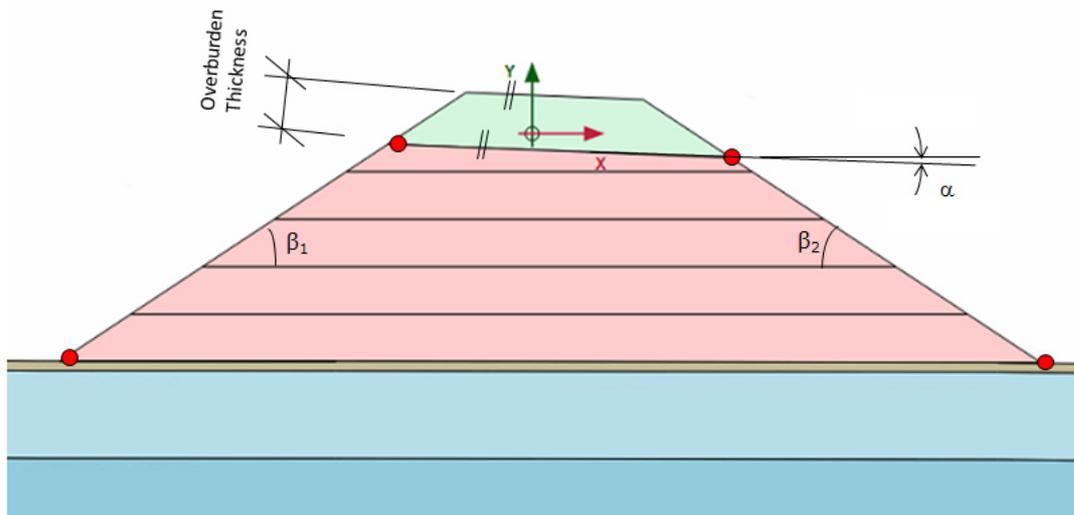


Figure 20: Boundary Conditions for Overburden

Regarding the phase setup of the overburden measure, the overburden construction begins after the consolidation of the final embankment layer. Then the overburden is left to consolidate for the amount of time specified by the user. Finally the deconstruction of the overburden is – due to simplicity reasons – assumed to consume the same amount of time as the overburden construction.

6.3.3.5 Combination of Measures

Generally the geotechnical measures described above are available in all combinations. The script is designed to allow a soil replacement as well as a drainage system and a temporary overburden all in the same embankment model. The user can specify for each measure via console inputs whether he wants to include it or not. The only exception is the combination of soil replacement and friction bases, which does not make sense from a technical point of view.

6.3.4 Automatic Phase Setup

The cross sections that the script can import from ProVI can thus far be modeled in Plaxis including:

- A complete geometry with embankment and subsoil layers,
- All subsoil materials that are involved in the project,
- The embankment divided in sublayers,
- Various geotechnical measures.

The next point of interest in designing the script is now an automatic setup of calculation phases depending on these model constituents. Therefore the script accesses the Plaxis application in the stages mode and creates a separate phase for every step in the construction of the embankment.

For every phase that needs to be created in Plaxis the script now has to perform the following tasks:

- Name the phases correctly
- Activate/Deactivate the correct soil polygons and other model features
- Apply the correct materials to each soil polygon in case of a change
- Apply the correct settings for phase duration
- Apply the correct calculation settings to each phase

As a blueprint for how this phase setup should be done, served a number of Plaxis models for actual road projects provided by Strabag. The phase setup as a whole and all parameters for each phase can be either traced back to those models or are designed to be set by the user while the script is running.

6.3.4.1 Excursus on Polygon Attributes

Both the activation and deactivation of soil polygons and the action of changing the material of a polygon rely on the script to extract data about these polygons. In Plaxis this information is stored as attributes to polygons. The attribute that is needed for the purpose of the phase setup (and also later for the mesh refinement) is called “BoundingBox” and is only present in the calculation modes of Plaxis. The “BoundingBox” is the geometric envelope of the polygon in x- and y-coordinates. It is therefore represented by four values:

- The minimum x-value of the polygon
- The maximum x-value of the polygon
- The minimum y-value of the polygon
- The maximum y-value of the polygon

Those four coordinate values then form rectangular window, in which the polygon is situated, as exemplarily sketched for an irregularly shaped polygon in Figure 21.

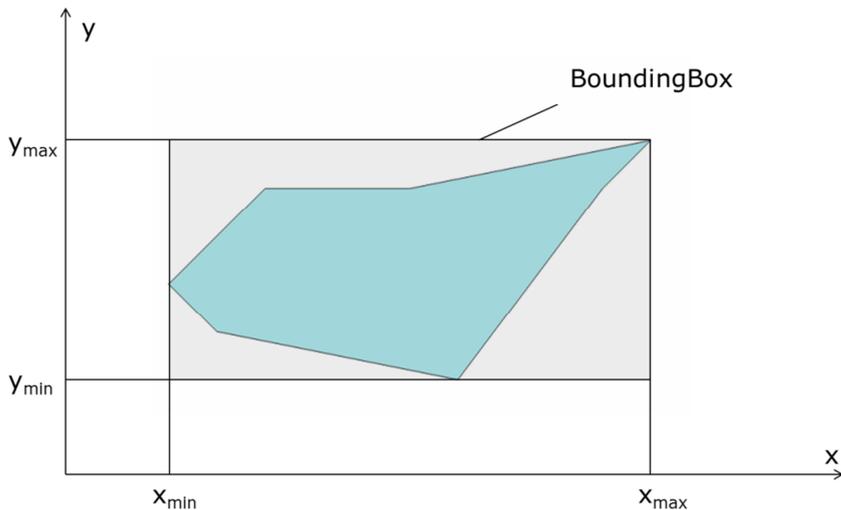


Figure 21: Bounding Box of Sample Polygon

The script can access this information regarding these polygon boundaries by calling “Polygon_1.BoundingBox”. In this case “Polygon_1” would be the name of the polygon. Plaxis then passes those four extremal values back to the script. There the values are then analyzed and used to decide whether or not the polygon needs to be activated, deactivated or have its material changed based on always different criteria.

6.3.4.2 Initial Phase

The initial phase is already present when switching to the stages mode in Plaxis. The naming is left unchanged for this phase. By default, when first switching to the stages mode, all soil polygons are deactivated. The polygons that need to be activated here are all soil polygons which form the subsoil body of the cross section. Their common characteristic by which those polygons are differentiated from all other regions, is that they stretch over the whole width of the cross section from -50 meters to 50 meters. In order to identify those polygons, the script uses the above-described polygon attribute “BoundingBox”. In this case the program cycles through all polygons of the model and activates only those which have “BoundingBox”-values of $x_{\min} = -50$ and $x_{\max} = 50$. Since no other polygons in the model share that characteristic, this is the ideal criteria for this layer activation. All other settings are left unchanged from the standard settings with which Plaxis creates the initial phase.

6.3.4.3 Optional Phase 1: Soil Replacement

In case the user specified to model a soil replacement for the cross section at hand, the script then automatically adds this phase after the initial phase. The parameters that are set for this phase can be seen in the table below.

Table 7: Settings for Soil Replacement Phase

Phase Identification	Soil Replacement
----------------------	------------------

Time Interval	2 [days]
Deformation Calculation Type	Consolidation

All other properties are left to Plaxis' standard settings. The polygons, where the soil should be replaced are defined as all those polygons, whose BoundingBoxes fulfill the following criteria - :

- $x_{\min} > -49$
- $x_{\max} < 49$
- $y_{\min} \geq$ the minimum y-coordinate of both soil replacement base points
- $y_{\max} <$ the y-coordinate of the first embankment sublayer line

Through the specification of those boundary values it is ensured that whole subsoil layers and parts of the embankment are excluded from the replacement. Only the polygons lying between the three soil replacement boundary lines and the virgin terrain line are therefore assigned the material "Soil Replacement".

6.3.4.4 Optional Phase 2: Friction Bases

In case the user specified to model friction bases for the cross section at hand, the script then automatically adds this phase after the initial phase. The parameters that are set for this phase can be seen in the table below.

Table 8: Settings for Friction Bases Phase

Phase Identification	Friction Bases
Time Interval	2 [days]
Deformation Calculation Type	Consolidation

All other properties are left to Plaxis' standard settings. The polygons, whose BoundingBoxes fulfill the listed criteria get assigned the material "FrictionBase":

- $x_{\min} >$ the minimum x-coordinate of the points forming the friction base lines
- $x_{\max} <$ the maximum x-coordinate of the points forming the friction base lines
- $y_{\min} \geq$ the y-coordinate of the base line of the friction base

In this case no value for y_{\max} needs to be specified, because the boundary is already uniquely defined by those three values and no other polygons are in danger of getting wrongly assigned the friction base material.

6.3.4.5 Optional Phase 3: Drainage Construction

The drainage construction phase can either follow directly after the initial phase or after one of the two previous optional phases. The parameters that are set for this phase can be seen in the table below.

Table 9: Settings for Drainage Construction Phase

Phase Identification	Drainage Construction
Time Interval	2 [days]
Deformation Calculation Type	Consolidation

All other properties are left to Plaxis' standard settings. In case it was selected to include a drainage layer in the model, again the BoundingBox attribute is used to identify the correct polygons with the script. The polygons, whose BoundingBoxes fulfill the following criteria are assigned the material "Drainage Layer":

- y_{\max} = the y-coordinate of the horizontal top of the drainage layer
- $x_{\min} \geq$ the minimum x-coordinate of all embankment points
- $x_{\max} \leq$ the maximum x-coordinate of all embankment points

6.3.4.6 Phases for each Embankment Sublayer

Following these optional phases now again comes a block of obligatory phases that – like the initial phase – are present in every model. For every two-meter-sublayer the script now adds two phases:

- A construction phase, where the layer is activated in the Plaxis model
- A consolidation phase, where the layer is left to consolidate for a certain amount of time, before the next layer is constructed

In contrast to the geotechnical measures – where two days were defined as standard duration in order to keep the user inputs at a reasonable level – here the user is required to set the duration of both these phase types. As usual this is done via a console input. The script then applies the specified durations to each phase of the same type. Because the embankment generally consists of multiple sublayers the setup of phases is done with a loop until all layers have been correctly activated. In contrast to the previously discussed activation technique with BoundingBoxes, a different approach was chosen for the embankment sublayers. Here every sublayer is only formed by exactly one polygon. The activation can therefore be done by using the Plaxis command "activate" and specifying directly the name of the polygon that was set during the embankment division process –see Figure 14. The phase setup is then carried out with a while-loop on the script side. Starting at the bottom of the embankment each layer gets its own construction and consolidation phase with the following parameters:

Table 10: Settings for Construction Phases

Phase Identification	Embankment x m Construction
Time Interval	User input [days]
Deformation Calculation Type	Consolidation

Table 11: Settings for Consolidation Phase

Phase Identification	Embankment x m Consolidation
Time Interval	User input [days]
Deformation Calculation Type	Consolidation

The “x” in the phase identifications of both phase types represents the current construction height of the dam. It therefore starts with “2” for the first sublayer and gets increased by two for every new layer that is activated. The top layer of the embankment forms an exception. While time interval and calculation type are kept constant, the phase identification is changed to “Embankment Final Construction” and “Embankment Final Consolidation” respectively.

6.3.4.7 Optional Phases 4: Overburden Phases

In case an overburden is included in the model, three phases are needed to accurately remodel the steps of construction in reality. The first phase where the overburden is constructed follows directly after the “Embankment Final Consolidation”-phase has ended. Afterwards a consolidation phase and a separate phase for deconstruction of the overburden are added. A first attempt was made without this separate phase. But the fact that Plaxis does not accept the model to be changed in phases that calculate until a certain degree of consolidation has been reached made it necessary to include it, because the next phase is such a degree-of-consolidation-type phase (see 6.3.4.8). Since it is again only one polygon that has to be activated or deactivated, this is carried out by directly naming the corresponding polygon. The parameters these phases are set-up with can be found in the tables below.

Table 12: Settings for Overburden Construction Phase

Phase Identification	Overburden Construction
Time Interval	User input [days]
Deformation Calculation Type	Consolidation

Table 13: Settings for Overburden Consolidation Phase

Phase Identification	Overburden Consolidation
Time Interval	User input [days]
Deformation Calculation Type	Consolidation

Table 14: Settings for Overburden Deconstruction Phase

Phase Identification	Overburden Deconstruction
Time Interval	Equal to time interval of Overburden Construction [days]
Deformation Calculation Type	Consolidation

The material for the overburden construction is assumed to be the same as for the embankment construction.

6.3.4.8 Degree of Consolidation

Finally we have reached the end state of our model regarding material assignment and polygon activation. The aim is now to analyze how much the embankment will deform over time. The phase that needs to be added here is a degree-of-consolidation-type phase. While in all previous phases we had a fixed duration, in this stage the time interval is not defined explicitly. Instead the calculation is ended by Plaxis once the degree of consolidation has reached 90 percent. As many of previously used parameters this value is selected based on the Plaxis reference models that Strabag provided. The phase's parameters can be studied in the table below.

Phase Identification	U90
Time Interval	-
Deformation Calculation Type	Consolidation
Loading Type	Degree of Consolidation
Degree of Consolidation	90%

6.3.4.9 Safety Calculation Phase

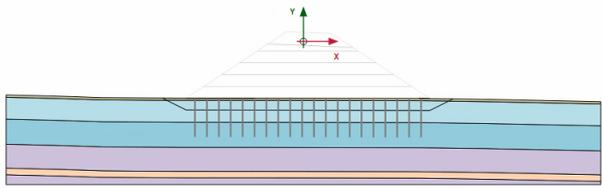
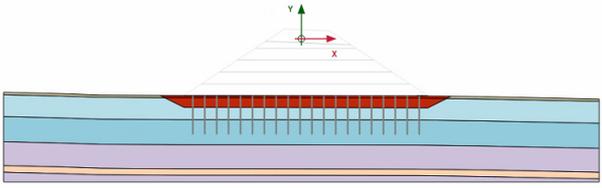
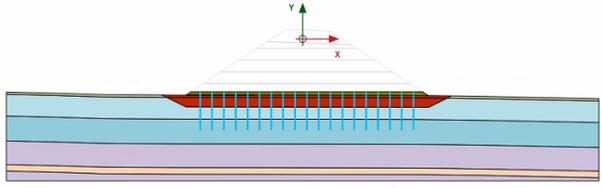
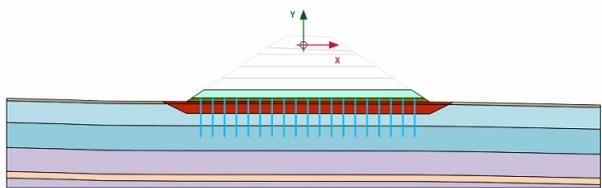
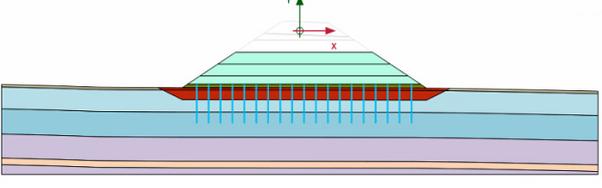
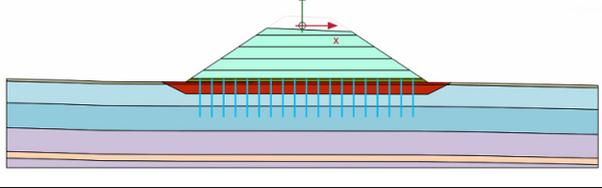
Lastly, an additional phase is added in order to investigate the embankments overall stability. However, the safety of the embankment is of interest directly after the construction and not only once the consolidation has already occurred. Therefore the phase is added as a daughter-phase to the phase "Embankment Final Construction" so that the safety can be directly analyzed for the just completed embankment. The properties of this phase are shown in the table below.

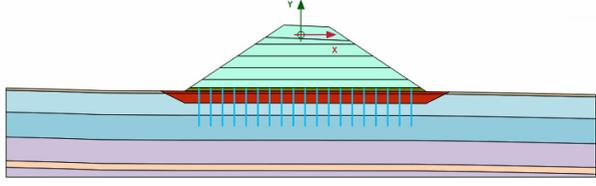
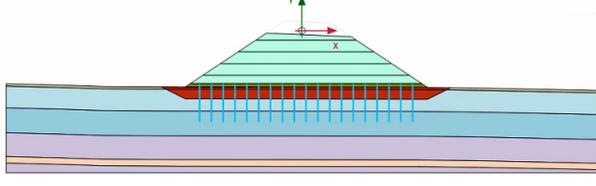
Phase Identification	Safety
Time Interval	-
Deformation Calculation Type	Safety
Loading Type	Incremental Multipliers
Msf	0.1

6.3.4.10 Phases Overview

Table 15 shows an exemplary overview of what the phases look like in the Plaxis application.

Table 15: Sample Phases Overview

Initial Phase	
Soil Replacement	
Drainage	
Embankment 2.0 m Construction	
Embankment 2.0 m Consolidation	
Embankment 4.0 m Construction	
Embankment 4.0 m Consolidation	
Embankment 6.0 m Construction	
Embankment 6.0 m Consolidation	
Embankment 8.0 m Construction	
Embankment 8.0 m Consolidation	
Embankment Final Construction	
Embankment Final Consolidation	

<p>Overburden Construction</p>	
<p>Overburden Consolidation</p>	
<p>Overburden Deconstruction</p>	
<p>U90</p>	
<p>Safety</p>	

6.3.4.11 Later-added Phases

Through most of the development phase of the python script, the phase setup looked just like the sample setup seen in Table 15. However towards the end of the script development it was decided to expand the phase setup quite significantly. In order to be able to analyze safety and also settlements for various embankment heights it was decided to include two additional daughter-phases for each embankment sublayer. Accordingly one phase of the type safety calculation was added directly after the construction phase of each sublayer and another phase of the degree-of-consolidation-type was added directly after the consolidation phase of each sublayer. These extra phases lead to a slightly more complicated phase tree, which can be studied for a sample model taken directly from the Plaxis application.

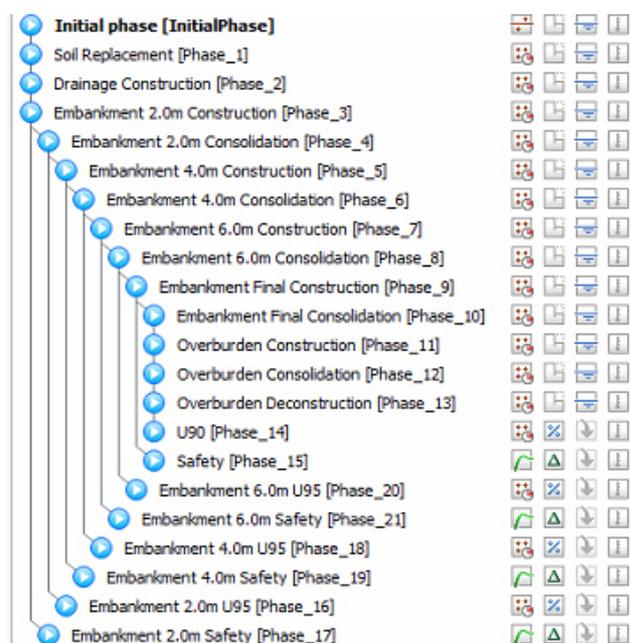


Figure 22: Phase Setup with added Safety and Consolidation Phases for each Sublayer

6.3.5 Meshing and Mesh Refinement

Similarly to the phase setup, the requirements regarding meshing and mesh refinement were defined by analyzing Plaxis models provided by Strabag. For the major part of the script development, the meshing was just handled internally in the script without the user being able to control it. In the final script version, however, the user is given the ability to control the mesh properties. Therefore the user can define the so-called “Relative Element Size Factor”, which then governs the total number of elements that are used in Plaxis. (Plaxis BV, 2018) Additionally, the script was designed to enable the user to easily refine the mesh in the areas of interest. Plaxis handles the refinement of the mesh in certain areas by changing the so-called “Local Coarseness Factor” for the polygons that lay in those regions. The regions, where the mesh should be refined in an embankment model are of course the embankment itself and the area below it. In order to be able to specifically target those regions three straight lines were added to the model in certain distance to the points that form the embankment envelope. Those three lines then form the boundary between refined and unrefined parts of the mesh – as seen in the figure below. The conditions for where these lines are drawn are different for each model, depending on the overall model height and whether or not drains are included in the model.

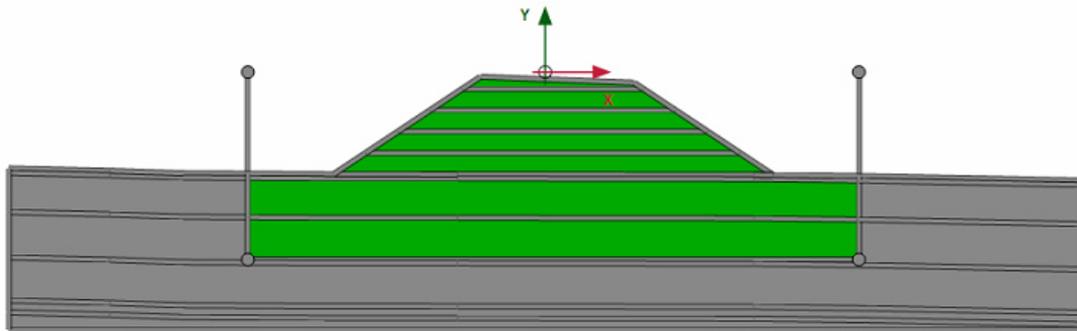


Figure 23: Sample Setup for Mesh Refinement

For the application of the “Local Coarseness Factor” the script uses again the BoundingBox attribute of the polygons. All polygons that are within the lines get applied the “Local Coarseness Factor” that the user needs to specify in the script’s console.

With a careful selection of both input parameters the script is capable of generating a well refined mesh for the purpose of the embankment deformation analysis as exemplarily shown in Figure 24.

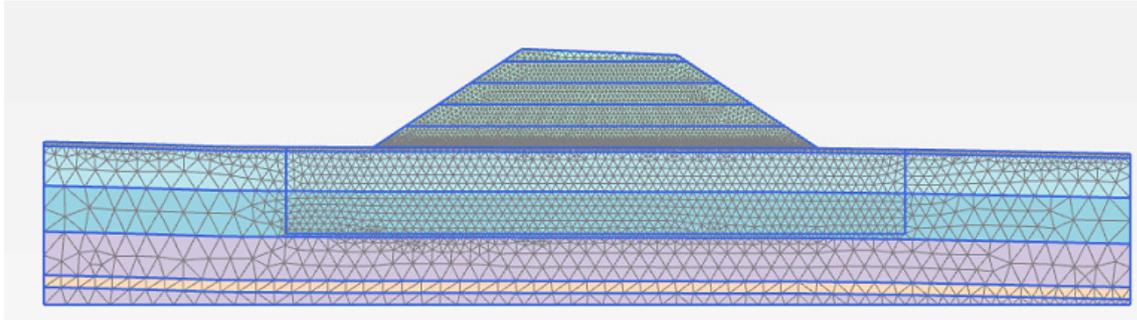


Figure 24: Sample Output Mesh

6.4 Extracting the Calculation Results

With all these steps it is now possible to import various embankment cross sections from ProVI and set up the model in Plaxis very efficiently. As discussed in the project definition the second project task is now to enable an easy retransfer of the calculation results to ProVI. In accordance with Strabag it was defined that specifically the deformation of the virgin terrain is of interest and should therefore be extracted.

Just as with the Plaxis Input application the script can also form a remote connection with the Plaxis Output application, making it possible to control and automatize all steps through remote scripting with the SciTE-editor.

6.4.1 Definition of Deformation Points

A first attempt to extract the deformations of the virgin terrain was made by selecting pre-calculation mesh points in Plaxis. However while it would be very easy to extract the occurring displacements in those points, this approach proved to be very impractical since a maximum of eight mesh points can be selected. For a model width of one hundred meters it is not possible to get a good impression of the deformation curve with such a small number of points. That is why this approach was dismissed. Instead, when going through the Plaxis Output Command Reference the command “getsingleresult” was discovered. This command requires the user to specify the x- , y- coordinates of a point and returns the results in that point. While many types of results can be returned with this command, in this case only the displacements in x- and y- direction (uy and ux) are of interest. Additionally, when working with this command the user can define the calculation phase for which one wants the results. This makes this command the perfect tool for the purposes of result extraction via the script.

Having found the right Plaxis command, a list of points that lay on the virgin terrain are required for it to work. Because of the way how the geometric data was imported from ProVI via the “QP-file” – see 6.2.1 – such a list of virgin terrain points is already stored within the python script. The only problem is that the points have very irregular spacing. Therefore it is necessary to interpolate

intermediate points between the already present virgin terrain points. The x-spacing that was selected for these intermediate points is one meter. The y-coordinates of those points are then interpolated between the two nearest points that define the virgin terrain. Figure 25 shows a sketch of the point interpolation.

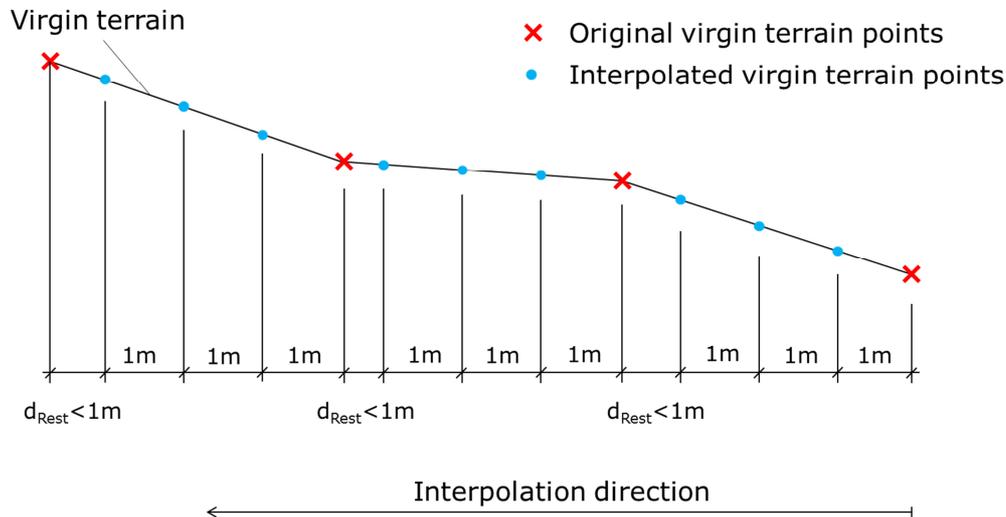


Figure 25: Sketch of Result Point Interpolation

With the spacing of one meter, this leads to around one hundred points for which displacements are read-out. The script is used to store all those points in a list. Then the script uses the “getsingleresult”-command in the Plaxis Output. Therefore the script-user needs to specify the phase for which he wants to generate the results – again through the use of the editor’s console. The Plaxis command then returns the displacements u_y and u_x for each point. By adding those displacements to the un-displaced point coordinates, the script then generates another list that contains the absolute 2D-coordinates of all virgin terrain points after the deformations have occurred (later used as x_{displ_2D} and y_{displ_2D} in chapter 6.4.3.2).

6.4.2 Writing Displaced 2D-Coordinates to a File

In order to be able to import the displaced points in ProVI, they need to be written to some kind of a file from the python script. Since ProVI is an AutoCAD based program, the first idea was to write the displaced point coordinates to an AutoCAD-script file. This is a text file with the extension “.scr” that lists certain commands that are then interpreted and carried out by the program. The python script therefore writes the commands “_multiple” and “_point” in the first two lines of the file followed by a list of the displaced 2D-point-coordinates, with each point in one line and x- and y- coordinates separated through commas. Once complete, the script saves this text file in the defined folder. From there it can be inserted to AutoCAD by using “Drag&Drop”. Figure 26 shows a sample drawing generated this way with the displaced point coordinates in red (in a very narrowly interpolated spacing of 0.2 meters) and the original cross section in black.

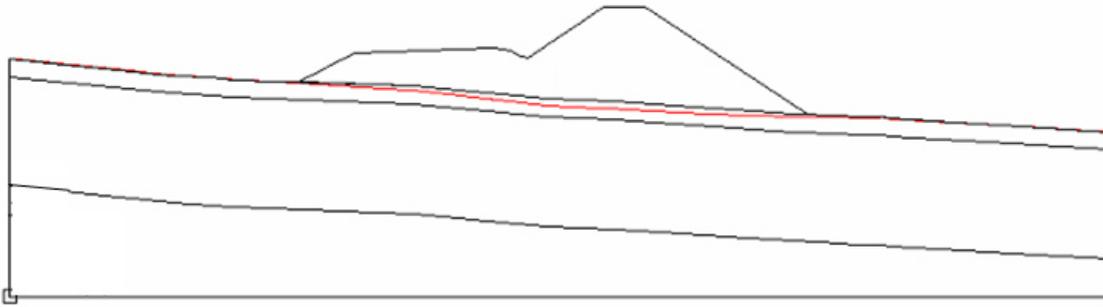


Figure 26: Displaced Virgin Terrain in AutoCAD

6.4.3 Export of Displaced 3D-Coordinates

However after consultation with Strabag it was decided that the re-import of 2D-point-coordinates is not sufficient for the purpose of this project. Instead a transformation from 2D to 3D coordinates within the python script was required. At this point neither of the files that have been imported from ProVI contained 3D-data of any fashion. Consequentially it soon became apparent that additional information needs to be imported from ProVI in order to be able to perform this coordinate transformation. For each cross section only two values are needed for that. On the one hand, the absolute 3D coordinates of the cross section's center point (with local x- and y-coordinates of zero) are required. On the other hand the orientation of the cross section in relation to the north in 3D-coordinates is also needed.

First ideas were to import a list of the projects alignment elements (straights, curves and clothoids) and to analyze their parameters in order to be able to calculate both of those needed values directly in the script. However this attempt proved to be impractical. Instead the chosen approach is an analysis of the pegging data of the road that is also present in ProVI.

6.4.3.1 Pegging Data Analysis

The pegging data of a road project is present in 3D-coordinates in ProVI. The idea behind this approach is to create a file with the pegging data for two points of each cross section. One point has to be the center point of the cross section, the point of intersection between road axis and cross section. With this point the absolute position of the cross section in 3D-project-coordinates is clearly defined. As second point can serve any point to the "right" of the axis – when looking in project direction. The sole purpose of this point is to be able to define the orientation of the cross section in relation to the project north.

The way this idea is now implemented is through importing another ProVI-generated file with the python script, as has been done for the "QP-file" and the

“BAUGRUND-file”. This file is called “ABSTECK-file” and needs to exactly match the required format that is described in this script’s user manual. For every cross section for which the script has carried out the calculation, the “ABSTECK-file” is then searched for the corresponding cross section chainage. Having identified the lines where that data is written, the script then extracts the 3D coordinates for both the cross section axis point and the point to the right of it. The absolute 3D-coordinates of the axis point can be directly used for the later coordinate transformation. The orientation of the cross section needs to be calculated. Figure 27 shows the geometrical relationship as described and the definition of variables for the orientation calculation.

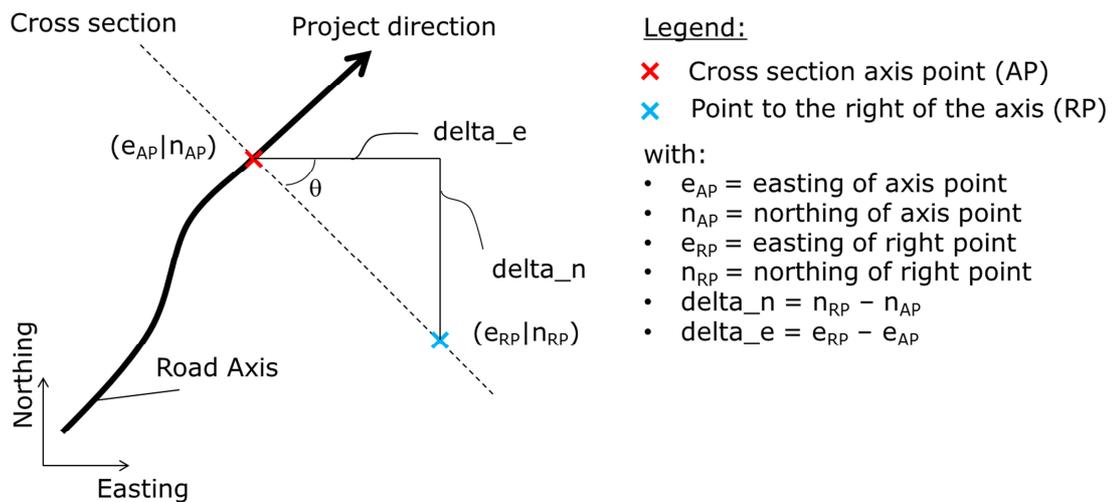


Figure 27: Point and Variable Definitions for Orientation Calculation

The angle theta as seen in the figure above is calculated with:

$$\theta = \arctan\left(\frac{(|\text{delta_n}|)}{(|\text{delta_e}|)}\right) \quad (2)$$

Here the absolute values of the differences delta_n and delta_e have been used. The angle theta is therefore always a positive value between zero and ninety degrees. In order to analyze in which quadrant the road direction is pointing for a specific cross section, now the algebraic signs of delta_n and delta_e are investigated. The conditions for this quadrant classification are summed up in Table 16.

Table 16: Quadrant Classification based on Algebraic Signs of calculated Distances between Pegging Data Points

Quadrant Number	Algebraic sign of delta_e	Algebraic sign of delta_n
Quadrant 1	+	-
Quadrant 2	-	-
Quadrant 3	-	+
Quadrant 4	+	+

6.4.3.2 Coordinate Transformation from 2D to 3D

With this information the script has now everything it needs to do the transformation from 2D-cross-section-coordinates (x , y) to 3D-project coordinates (easting, northing, height). Depending on the quadrant to which the road direction is pointing at a specific cross section, the transformation is done as seen in Table 18. Table 17 lists the variables that occur within the formulas that were used.

Table 17: Description of Variables for Coordinate Transformation

Variable	Definition
θ	Orientation of cross section (see 6.4.3.1)
e_{3D}	Easting of cross section point in 3D-coordinates
n_{3D}	Northing of cross section point in 3D-coordinates
h_{3D}	Height of cross section point in 3D-coordinates
e_{AP}	Easting of axis point in 3D-coordinates
n_{AP}	Northing axis point in 3D-coordinates
h_{AP}	Height of axis point in 3D-coordinates
x_{displ_2D}	Displaced x-coordinate of cross section point
y_{displ_2D}	Displaced y-coordinate of cross section point

Table 18: Formulas for Quadrant-dependent Coordinate Transformation

Quadrant 1	$e_{3D} = e_{AP} + x_{displ_2D} \times \cos(\theta)$	(4)
	$n_{3D} = n_{AP} - x_{displ_2D} \times \sin(\theta)$	(5)
	$h_{3D} = h_{AP} + y_{displ_2D}$	(6)
Quadrant 2	$e_{3D} = e_{AP} - x_{displ_2D} \times \cos(\theta)$	(7)
	$n_{3D} = n_{AP} - x_{displ_2D} \times \sin(\theta)$	(8)
	$h_{3D} = h_{AP} + y_{displ_2D}$	(9)
Quadrant 3	$e_{3D} = e_{AP} - x_{displ_2D} \times \cos(\theta)$	(10)
	$n_{3D} = n_{AP} + x_{displ_2D} \times \sin(\theta)$	(11)
	$h_{3D} = h_{AP} + y_{displ_2D}$	(12)
Quadrant 4	$e_{3D} = e_{AP} + x_{displ_2D} \times \cos(\theta)$	(13)
	$n_{3D} = n_{AP} + x_{displ_2D} \times \sin(\theta)$	(14)
	$h_{3D} = h_{AP} + y_{displ_2D}$	(15)

Those operations are carried out for every point for which the deformations of the virgin terrain have been previously calculated (see 6.4.1). Every result of these calculations is added to separate lists for easting, northing and height in order to store them for later use.

6.4.3.3 Writing displaced 3D-Coordinates to Files

At this point the script has succeeded in generating lists that contain the displaced 3D-coordinates for every point of the virgin terrain (in one meter interval) for the given cross section. The remaining goal is now to write this data to files that can then be read by the ProVI-application.

In order to be flexible regarding the import in ProVI, two similar file types are created. One is again an AutoCAD-script file like the one that was generated in chapter 6.4.2. Only this time the file consists of three columns for easting, northing and height instead of only two for x- and y-coordinates.

The other file is specifically shaped to allow an efficient input in ProVI. Therefore a CSV-type of file is created. The file consists out of four columns and has the format seen in Table 19.

Table 19: Format of Output File for ProVI Import of 3D-Coordinates

Point ID	Easting	Northing	Height
1	EE.EEEEEEE[m]	NN.NNNNNN[m]	HH.HHHHHH[m]
2	EE.EEEEEEE[m]	NN.NNNNNN[m]	HH.HHHHHH[m]
...
n	EE.EEEEEEE[m]	NN.NNNNNN[m]	HH.HHHHHH[m]

This file is then saved in the folder specified in the script and can be efficiently imported in ProVI for further use.

6.5 Additional Problems occurring during the Script Development

Many challenges that occurred during the development of the python script have been already discussed in previous chapters. However there were more problems that did not find reference yet and shall briefly be mentioned here.

6.5.1 Polygon Selection for Material Assignment and Mesh Refinement

Before the polygon attribute of BoundingBoxes was discovered (see chapter 6.3.4.1) different approaches were tested in order to change the assignment of materials to polygons during the mesh setup. One attempt consisted of accessing

the polygons depending on their names in the Plaxis application. However it was soon discovered, that the polygon naming that Plaxis uses, is different for the structure mode and the calculation mode. The polygons are internally renamed when switching from one mode to another. For example a polygon that was named “Polgon_2” in the structure mode gets assigned the name “Polygon_2_1” in the calculation modes. However if this polygon is divided into more than one subpolygons by the addition of lines to the model – like it is done for soil replacement and friction bases – the naming gets more complicated. Then every subpolygon has its own name “Polygon_2_X”. This makes it very unpractical to use polygon names for the purpose of material assignment, since the naming cannot always be predicted as it would be necessary.

When finally the BoundingBox-attribute to polygons was discovered, this did however not solve all problems right away. Tracing back to the fact that the BoundingBox consists of only four values that form the outer envelope of a polygon, there soon occurred wrong material assignments to polygons. As seen in the sketch below: When a drainage system and friction bases (or a soil replacement) were combined this problem appeared very often.

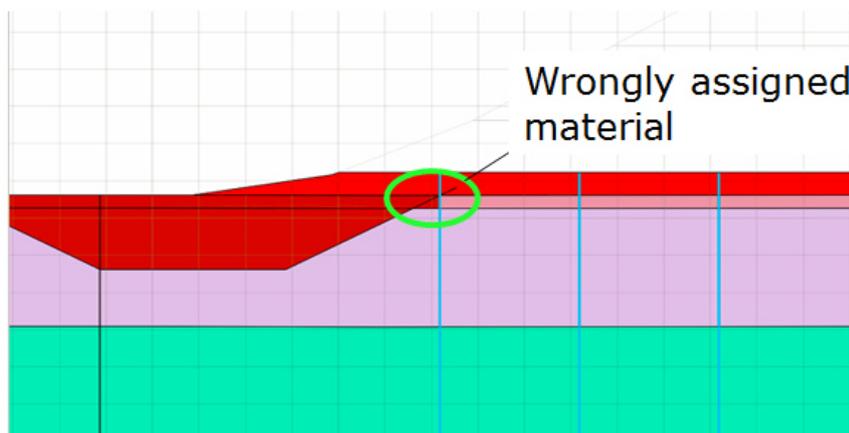
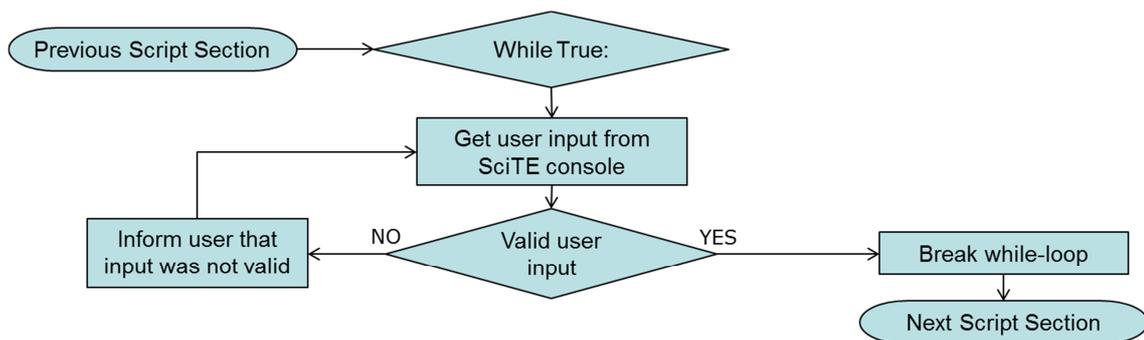


Figure 28: Problem with wrongly assigned Materials

For this problem the most effective solution proved to be a change in the order in which the model is set-up. Therefore first the boundary lines for soil replacement (or the friction bases) are drawn in the structures mode. Then the script commands Plaxis to switch to the stages mode. Here immediately the soil replacement phase is set-up – including the changed material assignment. At this point there is no problem with the BoundingBoxes, since those problematic small polygons are not formed yet by the addition of the drainage. Only then Plaxis is commanded to go back to the structures mode to build up the lines that form the vertical drains. By then going back again to the stages mode and completing the setup of the drainage phase and all other phases the problem has been eliminated.

6.5.2 User Inputs

Another complication in the development concerned the aspect of user inputs via the script's console in general. As discussed in detail in the user manual, the console is not able to detect the usage of the delete key. Therefore an attempt of the user to change his input in case he typed something wrong, had no effect and often ended in the termination of the script with an error message. While it was not possible to target the problem of the malfunctioning delete key itself, a safety net was introduced to all user inputs. For this reason all required user inputs are situated within a while-loop in the script. The script then tries to extract the needed data from the string sequence that the user typed in the console. If it fails to do so and normally an error would occur, this error is "caught" by an exception. The user is then informed that the script couldn't interpret the given input and required to repeat it. Only once the script has received a valid input the while-loop gets broken-up and the script continues with other sections. The flowchart of this safety measure around all user inputs can be seen below.



6.5.3 Color Problem

A smaller problem but one that persisted throughout most of the development process of the script was related to the colors of the imported soil materials. In ProVI those materials were already assigned a color in the RGB-color-format. For consistency reasons it was requested that the materials are depicted with the same color in Plaxis. In the user interface Plaxis does accept an input of tcolors in RGB-format. But when it comes to the setup of Plaxis materials with the command line or the remote scripting server by using the command "setproperties" Plaxis expects the colors to be defined in another format that is not known to the user. For the longest part during script development this resulted in the colors not matching the ones in ProVI. Only towards the end of the design process a similar command was found in the Plaxis command reference that allowed it to set material colors in the RGB-format. By using the command "setcolor" and accessing the right material the script is now capable of setting the correct RGB-color for all soil materials that were imported from ProVI.

6.6 Final Version of the Script

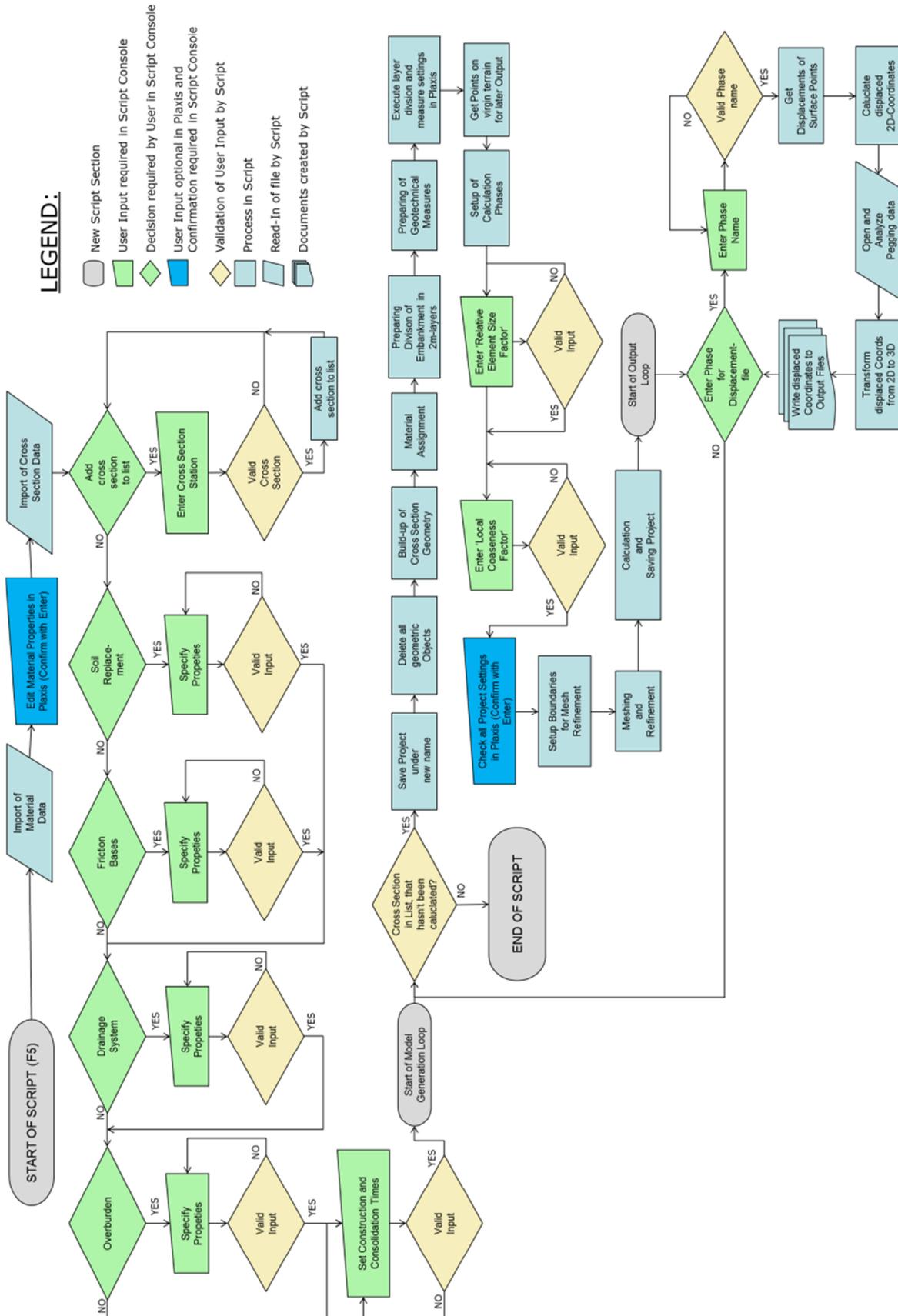


Figure 29: Flowchart of Script

6.6.1 Intended Workflow

Assuming that the three required ProVI-files that are needed for the import are present at the correct location and in the correct format, the workflow when using the script is designed as follows:

Once the script has been started the script immediately reads the “BAUGRUND-file” and imports all materials that are present in the project. The script is then paused to set all material properties that are necessary for the calculation. Once the script is continued by the user, the “QP-file” is read by the script. Next, a long input section is started, where the user is required to define:

- All cross section chainages that he wants to investigate in this session
- Which geotechnical measures – with all their parameters – he wants to include for those cross sections (they get applied to all of them)
- The construction and consolidation time of each 2m-sublayer

Once all of these parameters have been set, a loop within the script is started. For every cross section after another the steps listed below are followed by the script:

- (1) Complete model setup with all defined parameters
- (2) Request user input for meshing and mesh refinement properties
- (3) Pause the script in order to allow the user to check the whole model for its correctness
- (4) Calculation
- (5) Allow user to specify one after another all phases for which he wants to generate a file with displaced 3D-coordinates
- (6) Write and save the files for all required phases
- (6) Save Plaxis file
- (7) Create empty Plaxis model before the next cross section is imported.

After step seven is completed the loop-structure of the script causes the application to immediately begin with the setup of the next model. Looping over the steps (1) to (7) the script then develops and analyzes all cross sections that have been specified by the user. The material parameters as well as the geotechnical measures are hereby kept constant for all cross section. Only once the last cross section deformation analysis has been completed, the script is ended for good.

The setup of the script with this loop was designed in consultation with Strabag in order to enable an efficient import and calculation of not only one but multiple cross sections. The loop structure can also be studied in the flowchart depicted in Figure 29.

7 Alternative Script Version - Import of Material Properties

As described in chapter 6.6.1 the workflow with the script makes it almost possible to pass on using the Plaxis user interface. Apart from setting the material properties it is not necessary to switch to the application to analyze whole road embankment models. If it were possible to import not only material names and colors from ProVI, but also geotechnical material properties, the script would be capable of handling the complete model setup from beginning to end. Since ProVI does so far not support this setting of parameters, it was tried to find another solution in order to make the material setup more efficient.

Therefore the original script was kept as basis, but an additional section was added that enables the import of an extra text file. This text file was specially created to be easily importable with python and it contains material data sets, where the user can set all necessary material properties. Four different constitutive laws were therefore selected:

- Linear Elastic
- Mohr Coulomb
- Hardening Soil
- Hardening Soil with Small Stain Stiffness

For each of these material models the file has included five material slots that the user can fill. However, this number could be increased very easily if more materials of one model would be needed. In total the file can therefore contain up to twenty different materials. Furthermore each material model has of course different parameters that the user has to define. The required parameters for each model and also the structure of the file can be seen in

Figure 30. However while here is only depicted the first material data set (LE1, MC1, HS1 and HSS1) per constitutive model, the actual import file has five slots for each model.

```

#####
Soilmodel # LinearElastic # 1
#####
Material Set Number # LE1 # [-]
GammaSat # 20 # [kN/m³]
GammaUnsat # 18 # [kN/m³]
E' # 10000 # [kN/m²]
Poisson's Ratio # 0.2 # [-]
#####
Soilmodel # MohrCoulomb # 2
#####
Material Set Number # MC1 # [-]
GammaSat # 20 # [kN/m³]
GammaUnsat # 18 # [kN/m³]
Young's Modulus # 10000 # [kN/m²]
Poisson's Ratio # 0.2 # [-]
Friction Angle # 30 # [°]
Dilatancy Angle # 5 # [°]
Cohesion # 0.0 # [kN/m²]
K0 # 0.5 # [-]
R_inter # 1.0 # [-]
#####
Soilmodel # HardeningSoil # 3
#####
Material Set Number # HS1 # [-]
GammaSat # 20 # [kN/m³]
GammaUnsat # 18 # [kN/m³]
E_50_ref # 3000 # [kN/m²]
E_oed_ref # 3000 # [kN/m²]
E_ur_ref # 12000 # [kN/m²]
m # 1.0 # [-]
pref # 100 # [kN/m²]
K0nc # 0.5 # [-]
Friction Angle # 30 # [°]
Dilatancy Angle # 5 # [°]
Cohesion # 0.0 # [kN/m²]
OCR # 1.0 # [-]
POP # 0.0 # [kN/m²]
R_inter # 1.0 # [-]
#####
Soilmodel # HS_SMALL # 4
#####
Material Set Number # HSS1 # [-]
GammaSat # 20 # [kN/m³]
GammaUnsat # 18 # [kN/m³]
E_50_ref # 3000 # [kN/m²]
E_oed_ref # 3000 # [kN/m²]
E_ur_ref # 12000 # [kN/m²]
m # 1.0 # [-]
pref # 100 # [kN/m²]
K0nc # 0.5 # [-]
Friction Angle # 30 # [°]
Dilatancy Angle # 5 # [°]
Cohesion # 0.0 # [kN/m²]
OCR # 1.0 # [-]
POP # 0.0 # [kN/m²]
R_inter # 1.0 # [-]
G0_ref # 11000 # [kN/m²]
Treshh shear strain(0.7)# 0.1 # [-]

```

Figure 30: Material Properties Import File

In order to use this file efficiently the script is furthermore extended by an additional user input section. Here the script goes through all materials – that have already been imported at this point – and asks the user for every material whether he wants to assign the properties of a data set from the newly created file to it. If he decides to do so, he then has to specify the constitutive model and the number of the data set. The script then searches the file for the defined values and sets all parameters accordingly.

The flowchart of the whole added process can be seen in Figure 31.

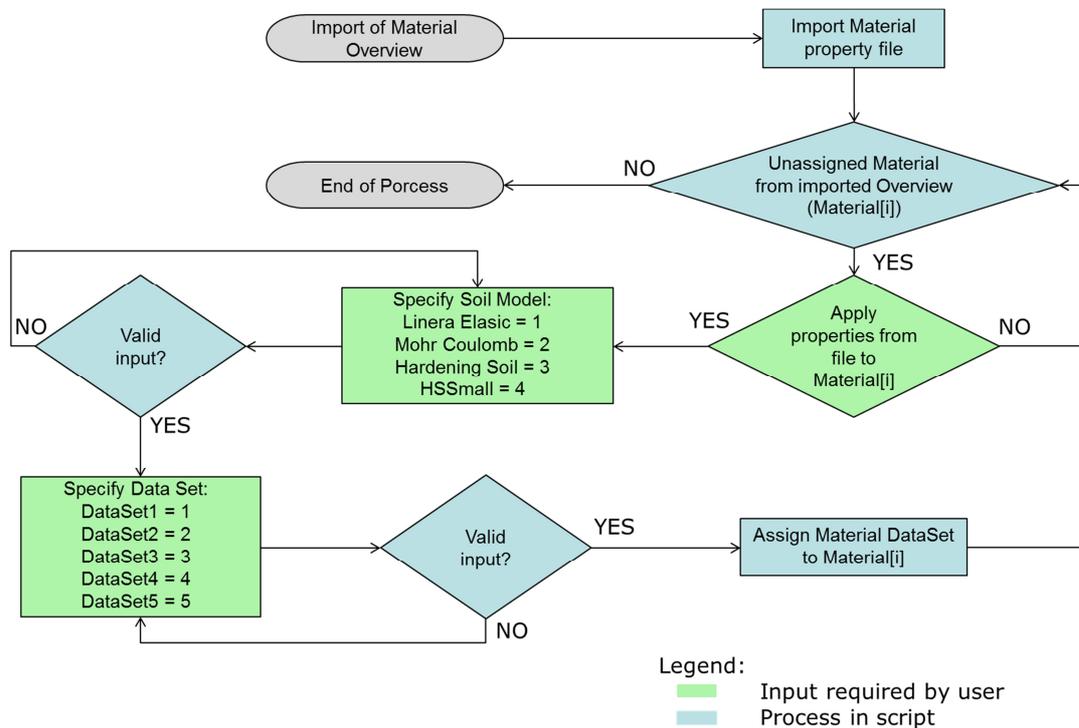


Figure 31: Flowchart of Material Property Assignment Process

With the addition of this process, the script can now – in theory – function completely independently from the graphic user interface of Plaxis.

8 Visualization of Output Data in 3D

As discussed in previous chapters, the script is quite efficient at analyzing multiple cross sections of a road embankment project. Displacement data can therefore be calculated for many chainages in relatively small intervals of a few meters. This provides the opportunity to create a 3D-visualization of the embankment and the occurring deformations.

In order to test this 3D-visualization a sample road project was select that includes an embankment section with the length of 300 meters. In order to induce significant displacements the material properties of a soft soil were defined for the displacement-governing soilayers. Furthermore the script was slightly modified. It now creates four output files with the point data for the following lines of every cross section:

- The virgin terrain in the original state – prior to displacements
- The virgin terrain after displacements have occurred
- The embankment on top of the virgin terrain –prior to displacements
- The displacements u_x , and u_y over the whole cross section width

In contrast to the original script version, this time the script appends all of this point data to just one file respectively. Thus, four large point-cloud-like files are generated that can be easily imported to a 3D-visualization software.

The interval in which the cross sections should be analyzed was set to 10 meters. With a total length of 300 meters embankment, this leads to 30 calculated cross sections.

The software that was used to visualize the data is called Surfer15. The advantage of this program is that it also allows the user to calculate the volume between two surfaces. Thus, one can easily compute an estimation of the excess material that is necessary due to settlements by analyzing the volume between original and displaced virgin terrain.

Four exemplary visualizations that can be generated with these files as basis can be seen in the figures below. Figure 32 shows the embankment surface as a wire mesh model. In Figure 33 the displacements u_y of the virgin terrain were included in the visualization. Here is clearly visible that the deformations increase with the height of the embankment. Figure 35 shows the displacements that occur along the longitudinal section of the road following the course of the axis, in the direction of the chainage as marked in Figure 34.

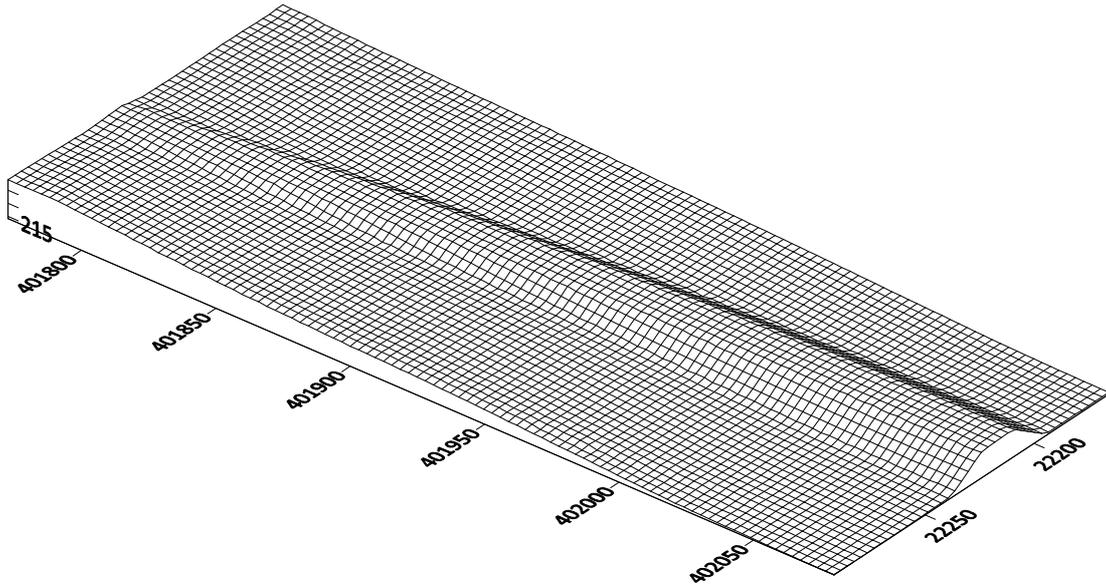


Figure 32: Embankment on top of Virgin Terrain (without Displacements)

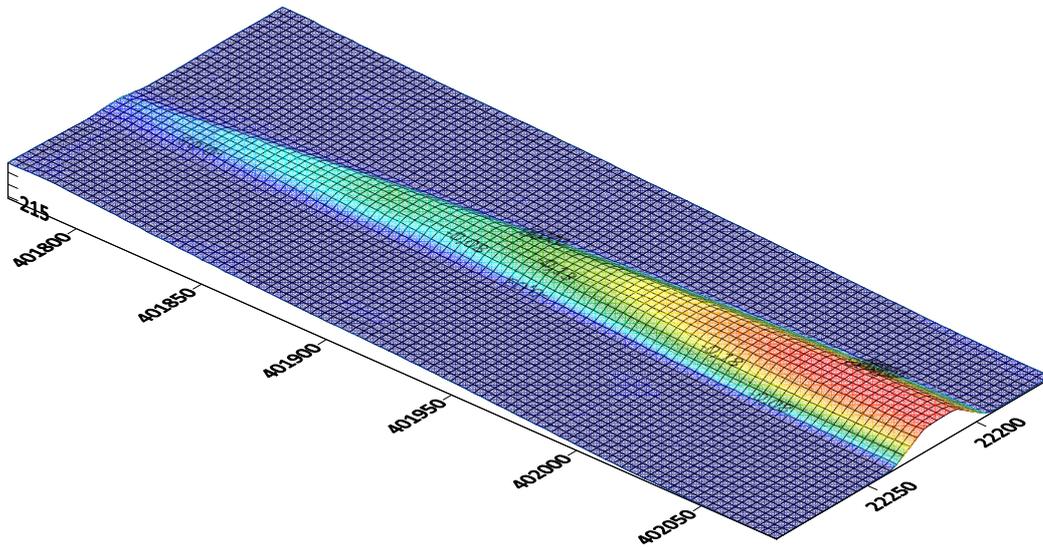


Figure 33: Embankment Model with colored Contour Plot of Displacements u_y

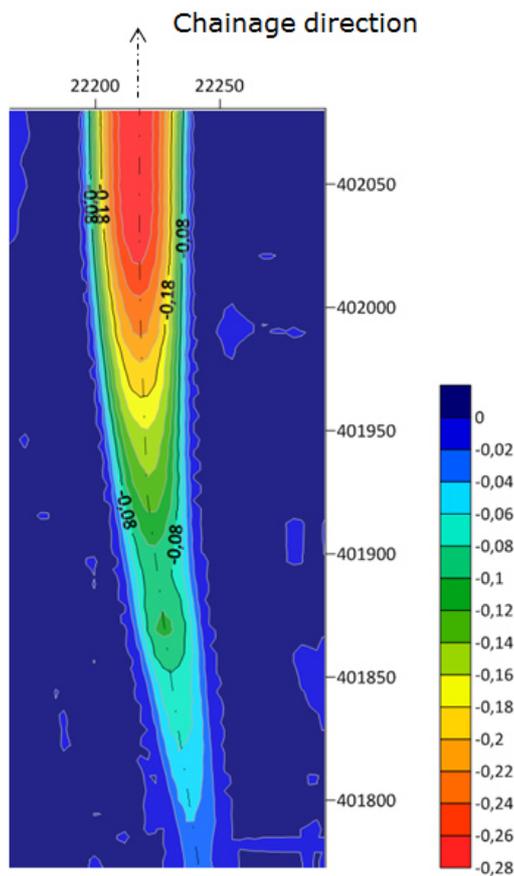


Figure 34: Contour Plot of Displacements u_y with Color Scale

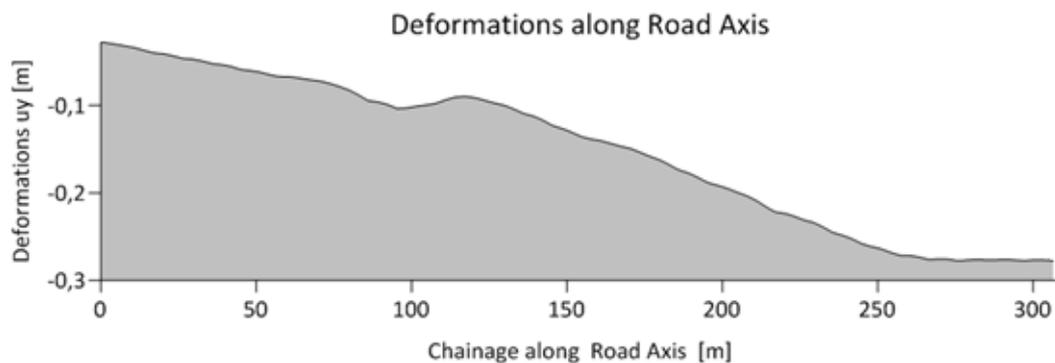


Figure 35: Displacements u_y along the Road Axis

8.1 Analysis of Required Excess Material

Additionally to those visualizations, Surfer15 was also used to compute an estimation of the excess material that is needed for the embankment construction. This is done by calculating the volume between the undisplaced virgin terrain and the virgin terrain after displacements have occurred. In order to investigate the influence of the interval in which the cross sections are analyzed, this calculation was carried out four times. Therefore the script was used to generate displacement files for cross sections in a 10 m-, 30 m-, 50 m-, and 100 m-

interval. Surfaces like the ones seen above were then generated for each of the intervals with Surfer15. The software is furthermore also used to extract the overall volume between the two mentioned surfaces through numerical integration. For the whole embankment length of 300 meters this volumes amount to the values seen in Table 20.

Table 20: Comparison of Required Excess Material for different Analysis Intervals

Analysis Interval	Required Excess Material [m³]	Deviation from 10m-Interval [%]
10m Cross Section Interval	1223.2	-
30m Cross Section Interval	1227.3	0.3
50m Cross Section Interval	1227.8	0.4
100m Cross Section Interval	1204.7	-1.5

The comparison of the calculated excess material quantities shows that even for the longest investigated interval of 100 meters the deviation from the assumed most exact value, calculated from the 10 meter interval, is only very small. This can be attributed to the fact that the height of the embankment increases quite smoothly over the whole model length. Consequentially even the 100 m-interval produces a quite accurate value in this case. However, if the height does not increase that smoothly, it might be necessary to decrease the analysis interval to much smaller values in order to receive reliable results.

9 Summary of the Script's Capabilities & Limitations

The final version of the described python script has the following capabilities on the input side of the Plaxis application:

- Import geometric data of various cross sections through the “QP-file”
- Modify that data to build a suitable model in Plaxis
- Import a material overview through the “BAUGRUND-file”
- Assign the soil materials to the correct regions through analysis of borehole data
- Create an automatic division of the embankment in sublayers
- Facilitate the setup of geotechnical measures, including:
 - Soil Replacement
 - Friction Bases
 - Drainage System
 - Temporary Overburden
- Automatic setup of calculation phases
- Automatic refinement and meshing

After the calculation has been carried out, the script can provide the following features for the user:

- Read-out of displacements of virgin terrain points for multiple calculation phases as specified by the user
- Transformation of the displaced 2D point coordinates to 3D point coordinates as used in ProVI
- Writing the displaced coordinates to a text file that can be imported in ProVI

However, there are also some limitations that come with the usage of the script:

- The embankment as well as each subsoil region has to form one closed polygon. That means the correct import of two separate embankment parts or subsoil layers with points of discontinuities is not possible with this script.
- The script relies on the input files having the exact format that it expects. Any deviation from this format might cause the import to fail.

10 Conclusion

The Python Remote Scripting Interface for Plaxis is a very powerful tool when it comes to the automatization of model generation. The usage of the programming language python enables an easy way to import and modify the data that is provided in textual form by ProVI. The script that has been created as part of this master thesis is a very specialized tool that was custom designed for the requirements of the people who will work with it at Strabag. The main purpose of the script is to provide an efficient way of transferring data between the planning software ProVI and the geotechnical software Plaxis. Additionally, the script is designed to facilitate and automatize the model setup in Plaxis. While on the one hand the model import should work as automatic as possible, another aim of the script was to maintain a high enough degree of flexibility. The outbalancing of these two oppositional goals was a main point of consideration over the course of the script's development.

As shown in previous chapters, the final version of the script can be used for a relatively time-efficient deformation analysis of numerous two dimensional cross sections. As shown in Chapter 8, this high density of calculation intervals can be used to transform this output data from 2D to 3D. While, originally, there were thoughts to include a separate 3D-import tool to the scope of this project, this idea was abandoned once it became clear that the 2D-import tool provides the indirect possibility for a three-dimensional analysis as well.

11 Literaturverzeichnis

- Autodesk Inc., 2002. *autodesk.com/buildinginformation*. [Online]
Available at: http://www.laiserin.com/features/bim/autodesk_bim.pdf
[Accessed 24 October 2018].
- Borrmann, A., König, M. & Beetz, J., 2015. *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. Wiesbaden: Springer Vieweg.
- Doebler, A., 2018. Aktivitäten der Bundesregierung zur Einführung von BIM im Infrastrukturbau. In: *Tagungsband Workshop Digitale Infrastruktur & Geotechnik*. Hamburg: Grabe, Jürgen, pp. 3-4.
- Goger, G., Piskernik, M. & Urban, H., 2018. *Studie: Potenziale der Digitalisierung im Bauwesen*, Vienna: Bundesministerium für Verkehr, Innovation und Technologie; Wirtschaftskammer Österreich Geschäftsstelle Bau.
- Günther, J. et al., 2015. *Expertengespräch: BIM in der Infrastrukturplanung* [Interview] 2015.
- Hirner, H., 2018. *Austrian Standards*. [Online]
Available at: <https://www.austrian-standards.at/infopedia-themencenter/infopedia-artikel/building-information-modeling-bim/>
[Accessed 23 October 2018].
- Litsch, J., 2018. BIM 5D im Verkehrswegebau / Erdbau. In: *Tagungsband Workshop Digitale Infrastruktur & Geotechnik*. Hamburg: Grabe, Jürgen, pp. 213-217.
- OBERMEYER PLANEN + BERATEN GmbH, 2017. *ProVI Benutzerhandbuch*. München: OBERMEYER PLANEN + BERATEN GmbH.
- Plaxis BV, 2018. *Plaxis 2D Reference Manual 2018*. Delft: Plaxis BV.

Appendix A: User Manual

A.1 System Requirements

A.1.1 Software

Please ensure that Plaxis 2D 2017 is installed on the PC that you want to use the script on. An installation of ProVI is not necessary as long as the files specified under A.2 are provided.

A.1.2 Folder Structure

The script can be saved in any folder you want. However, please make sure that the subfolders seen in Figure 36 are present in the same folder the script is saved in. Also please notice that upper and lower case and spelling of the folder names are needed to exactly match the requirements seen below.

Name	Änderungsdatum	Typ	Größe
OutputFiles	19.09.2018 12:06	Dateiordner	
ProjectFiles	19.09.2018 10:30	Dateiordner	
ProVIFiles	18.09.2018 18:02	Dateiordner	
PlaxisScript	19.09.2018 12:02	PY-Datei	68 KB

Figure 36: Folder Structure

A.2 Input Files

In order to be able to use the complete functional range of the script, please make sure, that the following files are saved in the subfolder “ProVIFiles”. Please take care that the file names exactly match the required format (also including upper and lower case).

A.2.1.1 “Cross_Section_File“

The so-called “Cross_Section_File” is based on a file format that occurs under the name “QP-Datei” in ProVI. It should be generated to contain the geometric data of all cross sections that you want to import in Plaxis. In order to generate a “Cross_Section_File” with ProVI that can be read by the python script, please follow these steps:

- Open the so-called “Querprofileditor” in ProVI (make sure that both axis and roadway layout are selected correctly).
- Click on “QP-Datei...”.
- Make sure that in the drop-down menu “Ausgabelinien”, the option “Alle Linien + Erdauftrag/Erdabtrag” is selected.

- Specify the values “von Station”, “bis Station” and “Intervall” so that all chainages you want to analyze in Plaxis are included. (Note: You can also include chainages, where no embankment is present in the cross section. The geometry of surface and subsurface soil layers will be imported nonetheless. However the calculation cannot be carried out using the script, since some features rely on an embankment line being present.)
- With clicking on “OK”, you have now generated a so called “QP-Datei”.
- You can check whether the just generated cross section file is correct by right-clicking on it in the project manager and selecting “Zeichnen”.
- In the now opening window you can select the tab “Linien” to see all lines that occur in the cross section file you have generated. Lines that have to be present for the cross sections to be importable are:
 - Line No. 21, called “Geländehöhe”, containing the geometric data of the surface
 - Line No. 42, called “Baugrund”, containing the geometric data of the subsoil layer boundaries
 - Line No. 51, called “Erdauftrag”, containing the geometric data of the embankment
- The cross section file you have just generated also appears in your ProVI working directory now.
- Open the file from the working directory with the windows text editor. Save it as “Cross_Section_File” in the subfolder “ProVIFiles” that you have generated in the script folder. Please also make sure that “.txt” is specified as file extension – as seen in Figure 37.

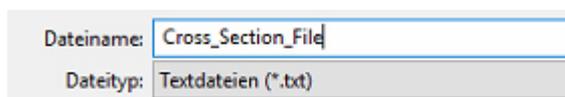


Figure 37: Extension of Cross_Section_File

A.2.1.2 “Subsoil_File”

The so-called “Subsoil_File” is based on the file called “Baugrunddatei” in ProVI. The “Subsoil_File” needs to contain:

- A list of all soil materials occurring in the project
- The borehole data that forms the basis for the subsoil layering

In order to obtain a valid and importable “Subsoil_File”, please follow these steps:

- Open the “Baugrunddatei” located in your ProVI working directory with the windows text editor. (By default the file name is of the format “BG+AXIS_NUMBER.EXTENSION” e.g. “BG001B.1”.)
- Save it as “Subsoil_File” in the folder ProVIFiles. Make sure that you have selected “.txt” as the file extension – see Figure 38

Dateiname:	Subsoil_File
Dateityp:	Textdateien (*.txt)

Figure 38: Extension of Subsoil_File

A.2.1.3 “Pegging_Data_File”

The “Pegging_Data_File” is based on the file called “ABSTECK” in ProVI. It is used in the python script in order to get the orientation of the cross sections for the conversion from 2D to 3D coordinates. The script relies on the “Pegging_Data_File” having one coordinate set for points lying on the axis and one coordinate set for points lying right to the axis (in project direction). In order to generate a valid “Pegging_Data_File” please follow the steps listed below:

- Right-click on the roadway project (“Trassenprojekt”) -> “Berechnen” -> “Trassenbuch”.
- In the pop-up window “Trassenbücher”, in the tab with the general setting, select the data for interval, starting chainage and end chainage. Consider that all chainages you want to analyze in Plaxis have to be included.
- Make sure the tick mark is selected for the “Absteckwerte”-box.
- Then switch to the “Absteckwerte”- tab and make sure that the two only ticked marks are the ones for the axis and for one part to the right of the axis, like shown in Figure 39.

Typ	ID	Achse	Beschreibung
<input type="checkbox"/> Grünstreifen	-4		
<input type="checkbox"/> Absatz	-3		
<input type="checkbox"/> Fahrbahn	-2	A900...	
<input type="checkbox"/> Fahrbahn	-1	A900...	
<input checked="" type="checkbox"/> Achse	0	A001B	
<input checked="" type="checkbox"/> Fahrbahn	1	A900...	
<input type="checkbox"/> Fahrbahn	2		
<input type="checkbox"/> Absatz	3		
<input type="checkbox"/> Bordstein	4		
<input type="checkbox"/> Bankett	5		
<input type="checkbox"/> Absatz	6		
<input type="checkbox"/> Beme	8		Lämschutzwand

Ausgabedatei: fortsetzen

Excel-Ausgabe

Punktausgabe Punktdatei:

Nachkommastellen bei Rechtswert/Hochwert:

Figure 39: Settings for Pegging_Data_File

- Also make sure that the tick mark for “Excel-Ausgabe” is set.

- When hitting “OK”, you have now created two files in your ProVI working directory. By default both of them are called “ABSTECK”. The one you want to use for the script is the one of the type “text file”.
- Save this text file with the name “Pegging_Data_File” in the script-subfolder ProVIFiles. Please also make sure that “.txt” is specified as file extension again.

A.3 Configure Script for First Time Use on Computer

In case you are using the script for the first time on a new computer you will need to configure the settings for password and port directly in the script. In order to do so, please open the script. The settings for ports and passwords have to be specified in the lines 20 and 21 for the Plaxis Input and the lines 23 and 24 for the Plaxis Output as seen below.

```

18 #Password and Port Settings
19 #Plaxis Input
20 Plaxis_Input_Port = 10000
21 Plaxis_Input_Password = "HGvkVQQci<v6Pxnr"
22 #Plaxis Output
23 Plaxis_Output_Port = 10001
24 Plaxis_Output_Password = "HGvkVQQci<v6Pxnr"

```

Figure 40: Script Section: Password and Port Configuration

Therefore open any already calculated Plaxis project. Select “Expert” -> “Configure remote scripting server” from the menu bar. A window similar to the one seen in Figure 41 will now appear on your screen.

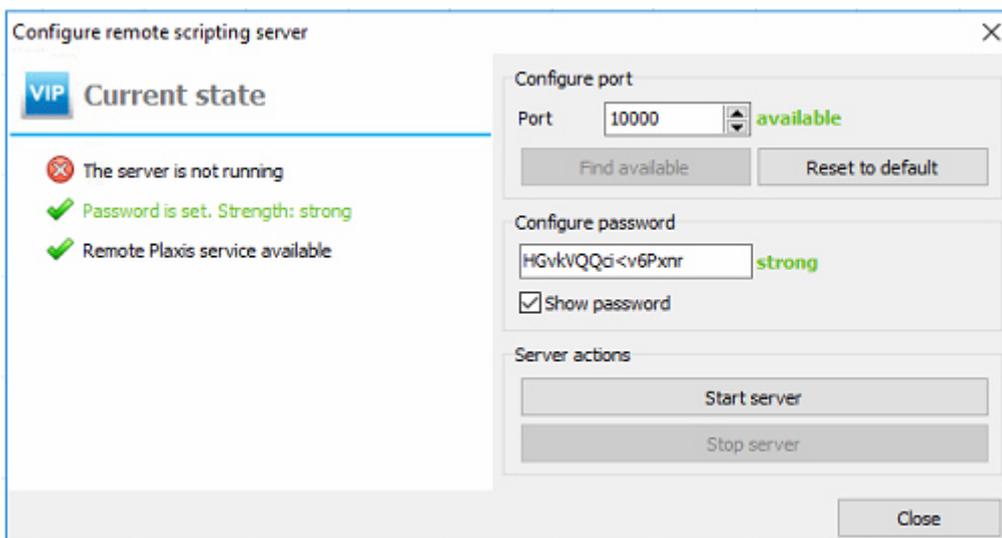


Figure 41: Configuration of Remote Scripting Server

Now copy the number specified there as port to line number 20 of the script (after the equal sign). Make sure to completely replace the number that has been written there. Then copy the character string seen in the “Configure password” field to line 21 between (!) both quotation marks. Make sure that both quotation marks are still present afterwards and that no additional character is included.

Now open the Plaxis Output for the same project and follow the same steps only this time replacing the values for “Plaxis_Output_Port” and “Plaxis_Output_Password” in the lines 23 and 24 of the script.

Once you have correctly configured those four values you can save the script and use it without any further actions anytime you work on the same computer.

A.4 Starting the Remote Scripting Server

In order to start the remote scripting server, please open Plaxis and click on “Expert” -> “Configure remote scripting server” in the menu bar. In the now opening window please keep the default settings – as they occur in your Plaxis version – for port and password and click on “Start Sever” (see Figure 41).

A.5 Starting the Script

In order to run the script, please open it from the location where it is saved, then proceed as follows:

- Start the script by either pressing “F5” or from the menu bar with “Tools”- > “Go”.
- Minimize the black window that has just occurred (but do not close it).
- On the right hand side a console field appears where you can specify all the inputs the script requires.

A.6 User Inputs

A.6.1 General Remarks on Inputs

Please note that much like the Plaxis command line the script also uses a point rather than a comma as decimal separator.

Another fact that is important to know about the user inputs required by the console is, that the console directly interprets all inputs that you enter via the keyboard rather than reading what is printed on the console. For example: When you want to enter the chainage of a cross section (e.g. 3450.0) and you enter “3459” by accident, hitting the “delete”-key once and entering “0” will not help you in getting the correct cross section. While on the console it looks like you entered “3450”, the script reads your input like “3459’delete‘0”. Therefore it will not recognize your input as a valid cross section and require you to repeat the input.

Consequentially also the console input via copying and pasting into the console line will not work, since the script will read your input as “Ctrl+V” rather than what the console line shows.

Since this means editing of already done inputs doesn't work, another safety measure was included. All inputs are therefore error-proofed using while-loops around them. Only once an input can be validated by the script, it will allow the user to continue.

For example, when by accident typing a letter when you are required to name the cross sections you want to import, the script console will show the dialogue seen in Figure 42 and require you to repeat the input.

```
For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
3834
Could not convert your input into an integer - please try again.
```

Figure 42: Error-proofing of Inputs

Similarly the same safety mechanism was introduced for all other user inputs.

A.6.2 Material Settings

Once you have started the script, it immediately begins reading the "Subsoil_File" located in the "ProVIFiles"-folder. All materials occurring in the file should now have been imported. The script is now paused, showing the message seen in Figure 43. The materials can now be edited directly in the Plaxis application. Once you have finished setting up all material properties, you can continue the execution of the script by pressing enter in the console line of the script editor. (Note: In case you want to include additional materials that were not imported via the "Subsoil_File" please edit one of the predefined materials named "CustomMaterial_Number". Do not add a new material, since the script relies on the materials being generated in the right order in later sections.)

```
Please edit the material properties now directly in Plaxis - press enter if finished.
```

Figure 43: Material Dialogue

A.6.3 Cross Section Chainages

Having set the materials and confirmed with enter, the script will now require you to name the chainages of all cross sections that you want to analyze in this session. Therefore the first two lines of dialogue seen below in Figure 44 will occur on your screen. You can now one after another enter all the cross section chainages that you want to import, creating a list that the script will store and later handle one at a time. A sample input might look something like the dialogue seen in Figure 44.

After having entered a cross section and confirmed with enter, the script will search the "Cross_Section_File" that was saved in the folder "ProVIFiles" for the given chainage. If the chainage is present in the file, the cross section will be

added to the list of cross sections to calculate. If not, the script will inform you that it couldn't find the cross section and ask for other cross sections to import.

Once you have added all cross sections to the list that you want to import, you can end the loop by entering “none”, “None”, “N” or “n” and confirming with enter.

```
For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
3000
Cross section 3000.0 added.

For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
31000
The cross section you entered (31000) was not found in the QP-file.

For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
3100
Cross section 3100.0 added.

For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
3200
Cross section 3200.0 added.

For which cross section(s) do you want to generate a model? Specify the station and press enter: (e.g. 3250.0)
Enter 'none' if finished.
none
```

Figure 44: Cross Section Input Loop

A.6.4 Geotechnical Measures

The next input section of the script deals with geotechnical measures. You can choose whether you want to include one or multiple geotechnical measures. Enter “Y” or “y” and confirm with enter in case you want to include the measure, just press enter if you don't want to include the measure in question. In case you want to add the measure, you will be asked additional questions where you can define its properties.

The measures you specify here are then applied to all cross sections you have entered in the section above.

A.6.4.1 Soil Replacement

In case a soil replacement should be added to the model, you have to specify the following parameters:

- Depth of the soil replacement in meters
- Inclination of the excavation slopes of the soil replacement (needs to be specified in the format delta horizontal/delta vertical e.g. „1/0.8”)

For more information regarding the standardized boundary conditions for soil replacements, see Chapter 6.3.3.1.

11.1.1.1 Friction Bases

This feature is for obvious reasons only available in case no soil replacement was specified to be modeled. In case friction bases should be added to the model, you have to specify the following parameters:

- Base length of both friction bases in meters (The base line is hereby always modeled horizontally.)
- Portion of the base length in meters, that is lying to the “outside” of the embankment area
- Depth of the friction bases in meters (measured vertically downward from the respective dam toe point)
- Inclination of the excavation slopes of friction bases (this needs to be specified in the format delta horizontal/delta vertical e.g. “1/0.8”)

For more information regarding the standardized boundary conditions for friction bases, see Chapter 6.3.3.2.

A.6.4.2 Drainage System

Regardless whether you have chosen to include either soil replacement, friction bases, or none of the two you have then the option to include a drainage system into your model(s). In case a drainage system should be added, you have to specify the following parameters:

- Thickness of the drainage layer in meters*
- Horizontal spacing of the vertical drains in meters
- Length of the vertical drains in meters (measured from their respective top coordinate, which might be different for each drain, in case no drainage layer is constructed)

* Since the intermediate embankment layers are assumed to be constructed horizontally the same boundary condition was assumed for the drainage layer. Starting from the (bottom) y-Coordinate of the embankment at its axis ($x=0$), the thickness specifies the upward offset of a horizontal line. This line is then forming the upper boundary of the drainage layer. Entering „0” as the thickness of the drainage layer is also possible and results in a dam model without a separate layer for drainage.

For more information regarding the standardized boundary conditions for friction bases, see Chapter 6.3.3.3.

A.6.4.3 Overburden

Independently from all previously listed measures you have the option to include a temporary overburden in your model in order to accelerate settlements. In case

a temporary overburden should be added to the model, you have to specify the following parameters:

- Thickness of the overburden in meters (measured perpendicular to an imaginary line connecting top left and top right point of the embankment cross section)
- Construction time for the overburden in days
- Consolidation time for the overburden in days (not including the construction time)
- For more information regarding the standardized boundary conditions for friction bases, see Chapter 6.3.3.4.

A.6.5 Construction and Consolidation Times

Following the geotechnical measures section you will now have to specify the construction and consolidation times for each 2m-intermediate-layer of the embankment. Both inputs are required in the unit days. An exemplary console dialogue can be seen in Figure 45.

```
How long does the construction of each 2m-layer take? [days]
3
How long is the consolidation time until the next 2m-layer is constructed? [days]
12.5
```

Figure 45: Construction and Consolidation Times Input

A.6.6 Mesh Properties

After the model geometry and the phase setup has been done via the script, the user then has to specify the properties of the mesh through the two values “Relative Element Size Factor” and “Local Coarseness Factor” after the following console dialogue occurs on the script:

```
How fine should the mesh be? - Enter a value for 'Relative Element Size Factor'
Values between 0.01 and 0.1 might lie in a reasonable range.
0.05
Please set a value for the 'Local Coarseness Factor' for the area around the embankment.
Values between 0.03125 (very fine!) and 1.0 (no refinement) might lie in a reasonable range.
0.4
Are all properties set correctly? - Please check all settings directly in Plaxis. Last pause before start of calculation.
Continue with enter
```

Figure 46: Mesh Dialogue

For detailed information regarding the mesh setup through those two properties, please check the Plaxis 2D Reference Manual. Once you have successfully specified both values, you have the last chance to make changes to the model directly in the Plaxis application. Once the last part of the script dialogue is confirmed with “Enter”, the script will continue with meshing.

A.7 Build-up of Model in Plaxis

Once the last user input is confirmed with Enter, the script starts with its internal mechanism of building up the model in the Plaxis application. The steps in which the model creation takes place can be seen in the following list or more in detail in Figure 29.

- Build-up of geometry, including soillayering and embankment geometry
- Assignment of soil materials to the soillayers and the embankment
- Division of the embankment in horizontal 2m-sublayers
- Build-up of geotechnical measures (as specified by the user)
- Setup of calculation phases in the Staged Construction mode following the scheme seen in Figure 47.

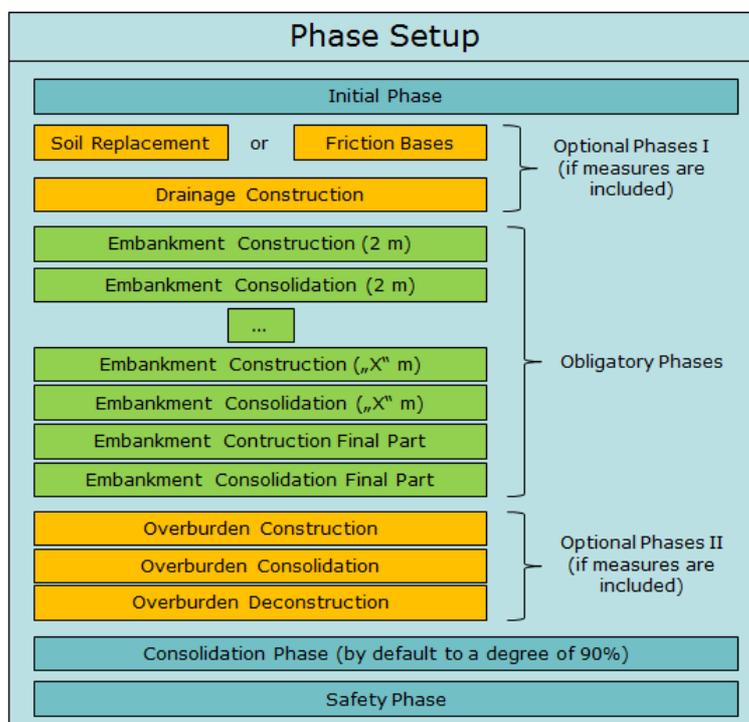


Figure 47: Phase Setup Scheme

- User Input: Define mesh fineness through “Relative Element Size Factor”
- User Input: Define degree of local refinement through “Local Coarseness Factor”
- Last pause in script to give the user a chance to check all model parameters
- Setup of refinement boundaries; all polygons within the following boundaries are refined:
 - Left boundary: 8 meters to the left of the left dam toe point
 - Right boundary: 8 meters to the right of the right dam toe point
 - Bottom boundary: 8 meters below the lowest dam y-coordinate, but at least 1 meter above the lowest y-coordinate of the whole model. If drains are present, the bottom boundary is moved to the point lying a fifth of the distance between the lowest drain coordinate and the bottom boundary of the model below the lowest drain coordinate

- Top boundary: no upward boundary regarding refinement
- Meshing
- Calculation
- Saving of the Plaxis Project File in the subfolder PlaxisFiles

A.8 Interaction of Script with Plaxis Output

Once the calculation is done the Plaxis Output Window is automatically opened by the script. The user is now required to specify the Calculation Phase for which he wants to get a list of displaced surface coordinates. The phase name is required to be of the format „Phase_Number”, as seen in Figure 48.

```
For which Phase do you want to generate a displacement-file? (enter e.g. 'Phase_12')
Enter 'none' if finished
Phase_18
```

Figure 48: Phase Input for Generation of Displacement File

Once a phase has been entered a list of surface coordinates with an x-coordinate interval of one meter is given to the Plaxis Output, which then calculates the displacements u_x and u_y for all listed coordinates. Adding the values for the displacements to the original surface coordinates then gives a list of displaced 2D-coordinates.

After the coordinate transformation back to 3D project coordinates (using the „Pegging_Data_File”), the displaced surface coordinates are then written to two different text files:

- AutoCAD-Script file (.scr-extension)
- ProVI-Import-File (.txt-extension)

Once the data is written to both these files they are saved to the subfolder OutputFiles. You then have the chance to type another phase name, in case you are interested in the displacements for another phase.

When a displacement file has been generated for all phases of interest, you can end the section with typing “none” or “None” and confirming with enter.

A.9 Continuation of Model Generation Loop

The script has now handled the first cross section of the list that you specified (see Chapter A.6.3). The loop is now continued by deleting all geometric objects that were present in Plaxis for the last model.

All material properties and the settings for geotechnical measures are kept the same for the next cross section.

Once all objects have been deleted the build-up of the next model starts, followed by all steps listed in the Chapters A.7 and A.8.

This process is repeated until all cross sections in the list are dealt with.

A.10 Explanations for the Output Files

A.10.1 AutoCAD Script File

The name convention of the file was defined as follows:

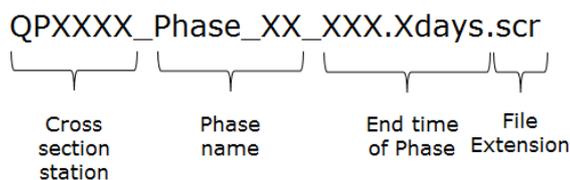


Figure 49: AutoCAD Script File Name Convention

The AutoCAD Script has the commands „_multiple” and „_point” written in the first two lines, followed by the coordinates of all displaced surface points in the convention „Easting” comma „Northing” comma „Altitude”. Each of those three values has been rounded to six decimal places prior to writing.

The script file can be used in AutoCAD by simply dragging and dropping the file into an opened AutoCAD drawing.

A.10.2 ProVI Import File

The name convention of the ProVI Import File was defined as follows:

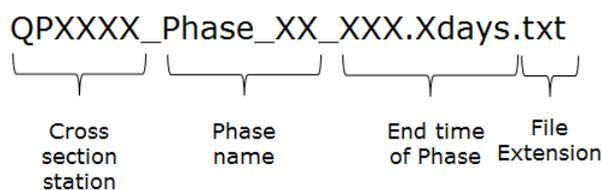


Figure 50: ProVI Import File Name Convention

The text file was internally structured in order to match a format that ProVI can easily import. Therefore the displaced point data was divided in four columns (Point_ID, Easting, Northing, and Altitude), each column separated from the next by a “#”-sign.

The text file can then be easily transformed into a ProVI Point File via the ProVI-program „DATKONV”.

A.11 Known Issues

One known issue of the script is a bug, internally called „BBox-error”. The script hereby tries to receive data from Plaxis regarding the bounding coordinates of polygons and fails to retrieve the required data.

Sadly the bug couldn't be solved yet since the error is not reproducible.

On the flip side, this however also means that setting the model with the exact same configuration again will result in the script running successfully.

Appendix B: Source Code of Script

Color Legend:

	Start of Server
	Help Functions for later use in Script
	Import of Material Data
	Creation of Materials in Plaxis
	Input Section
	Save & Prepare Plaxis for next Loop
	Geometry Import & Setup
	Material Assignment
	Division of Embankment
	Setup of Geotechnical Measures
	Setup of Phases
	Meshing & Refinement
	Preparations for Displacement Export
	Extract Displacements from Plaxis Output
	Coordinate Transformation
	Write to File

```
##### SCRIPT START #####

import codecs
import os

##### MODIFY PATH #####
originalPath = os.path.dirname(os.path.abspath(__file__))
partA = originalPath[0:2]
partB = originalPath[3:]
scriptPath = partA + "\\\" + partB

##### START SERVER #####

from plxscripting.easy import *
from decimal import *
from math import *

#Password and Port Settings
#Plaxis Input
Plaxis_Input_Port = 10000
Plaxis_Input_Password = "HGvkVQQci<v6Pxnr"
#Plaxis Output
Plaxis_Output_Port = 10001
Plaxis_Output_Password = "HGvkVQQci<v6Pxnr"

#Start Server
s_i, g_i = new_server('localhost', Plaxis_Input_Port, password= Plaxis_Input_Password)
s_i.new()

##### HELP FUNCTIONS #####

#HELP FUNCTION: get Point for specified x-coord "d"
def getPoint(coordsX, coordsY, d):
    for i in range(0, len(coordsX)):
        if coordsX[i] > d:
            iLeft = i - 1
            iRight = i
            break

    pX = d
    pY = coordsY[iLeft] + (coordsY[iRight] - coordsY[iLeft])\
```

```

*(d - coordsX[iLeft])/(coordsX[iRight]-coordsX[iLeft])
return px, py

```

```

#HELP FUNCTION: GET POLYGON COMMAND

```

```

def getPolygonCommand(PointListX, PointListY):

```

```

    start = "g_i.polygon("
    middle = ""
    for n in range(0, len(PointListX)):
        middle = middle + str(PointListX[n]) + "," + str(PointListY[n]) + ","
        if n == (len(PointListX) - 1):
            middle = middle[:-1]
    end = ")"
    command = start + middle + end
    return command

```

```

#HELP FUNCTION: GET POLYGON HEIGHTS AT x=0

```

```

def layerHeight(pN):

```

```

    x_coords = []
    y_coords = []

    for j in range(0, len(g_i.Polygons[pN].Points)):
        x_coords.append(g_i.Polygons[pN].x + g_i.Polygons[pN].Points[j].x)
        y_coords.append(g_i.Polygons[pN].y + g_i.Polygons[pN].Points[j].y)

```

```

    if (x_coords[0] > 0):
        for i in range(0, len(x_coords)):
            if x_coords[i] < 0:
                iLeft = i
                iRight = i-1
                xLeft = x_coords[iLeft]
                xRight = x_coords[iRight]
                yLeft = y_coords[iLeft]
                yRight = y_coords[iRight]
                break

        for i in range(iLeft, len(x_coords)):
            if x_coords[i] > 0:
                jRight = i
                jLeft = i-1
                xLeft2 = x_coords[jLeft]
                xRight2 = x_coords[jRight]
                yLeft2 = y_coords[jLeft]
                yRight2 = y_coords[jRight]
                break

            elif x_coords[(len(x_coords)-1)] < 0:
                xLeft2 = x_coords[(len(x_coords)-1)]
                xRight2 = x_coords[0]
                yLeft2 = y_coords[(len(x_coords)-1)]
                yRight2 = y_coords[0]

```

```

    elif (x_coords[0] < 0):
        for i in range(0, len(x_coords)):
            if x_coords[i] > 0:
                iLeft = i-1
                iRight = i
                xLeft = x_coords[iLeft]
                xRight = x_coords[iRight]
                yLeft = y_coords[iLeft]
                yRight = y_coords[iRight]
                break

        for i in range(iRight, len(x_coords)):
            if x_coords[i] < 0:
                jRight = i-1
                jLeft = i
                xLeft2 = x_coords[jLeft]
                xRight2 = x_coords[jRight]
                yLeft2 = y_coords[jLeft]
                yRight2 = y_coords[jRight]
                break

            elif x_coords[(len(x_coords)-1)] < 0:
                xLeft2 = x_coords[(len(x_coords)-1)]
                xRight2 = x_coords[0]
                yLeft2 = y_coords[(len(x_coords)-1)]
                yRight2 = y_coords[0]

```

```

    else:
        y2_0 = 0;
        if x_coords[1] > 0:
            for i in range(1, len(x_coords)):
                if x_coords[i] < 0:
                    iLeft = i
                    iRight = i-1
                    xLeft = x_coords[iLeft]
                    xRight = x_coords[iRight]
                    yLeft = y_coords[iLeft]
                    yRight = y_coords[iRight]
                    break

```

```

        else:
            for i in range(1, len(x_coords)):
                if x_coords[i] > 0:
                    iLeft = i
                    iRight = i-1
                    xLeft = x_coords[iLeft]
                    xRight = x_coords[iRight]
                    yLeft = y_coords[iLeft]
                    yRight = y_coords[iRight]
                    break

```

```

y1_0 = yLeft + (yRight - yLeft)/(xRight-xLeft)*(-xLeft)

```

```

if "y2_0" in locals():

```

```

    y2_0 = 0

```

```

else:

```

```

    y2_0 = yLeft2 + (yRight2 - yLeft2)/(xRight2-xLeft2)*(-xLeft2)

```

```

hLayer = abs(y1_0 - y2_0)

```

```

return(hLayer)

#HELP FUNCTION: REMOVE ELEMENT FROM LIST
def removeElement(List, ListIndex):
    return (List[:ListIndex] + List[ListIndex+1:])

#HELP FUNCTION: convert gon to radians
def gon_to_rad(alpha_gon):
    alpha_rad = alpha_gon*pi/200
    return alpha_rad

#HELP FUNCTION: GET BOUNDING BOX
def getBoundingBox(Polygon):
    a = str(Polygon.BoundingBox)
    #print(a)
    b = ""
    c = []
    i = 0
    while i < len(a):
        if a[i] == "-" or a[i].isdigit() or a[i] == ".":
            b = b + a[i]

            if a[i] == ";" or a[i] == ")" :
                #print(b)
                try:
                    c.append(float(b))
                except:
                    print("BBox error")
                    print(a)
                    print(b)
                    c.append(0)

                b = ""
                i +=1

    del c[2]
    del c[4]
    #print(c)
    return c

#HELP FUNCTION: GET ABSOLUTE POLYGON COORDINATES
def getAbsCoords(Polygon):
    x_abs = []
    y_abs = []
    for i in range(0, len(Polygon.Points)):
        x_abs.append(Polygon.x + Polygon.Points[i].x)
        y_abs.append(Polygon.y + Polygon.Points[i].y)

    return x_abs, y_abs

#HELP FUNCTION: GET INDEX OF POLYGON:
def getIndex(yMax, xMin, xMax):
    for i in range(0, len(g_i.Polygons)):
        BBox = getBoundingBox(g_i.Polygons[i])
        if abs(BBox[3] - yMax) < 0.01 and BBox[0] > (xMin-0.05) and BBox[2] < (xMax+0.05):
            break
    return i

#HELP FUNCTIONS: GET SINGLE RESULT COMMANDS
def getSingleResultXCommand(Phase_No, PointX, PointY):
    return "ux.append(float(g_o.getsingleresult(g_o."+str(Phase_No)+"", g_o.Soil.Ux," + str(PointX)\
+", "+str(PointY)+")))"

def getSingleResultYCommand(Phase_No, PointX, PointY):
    return "uy.append(float(g_o.getsingleresult(g_o."+str(Phase_No)+"", g_o.Soil.Uy," + str(PointX)\
+", "+str(PointY)+")))"

#####
##### Import material data #####
#####

#Delimiter
delimiter = "#"

#Material Lists
matData20 = []
matData21 = []
MaterialNumber = []
MaterialCode = []
MaterialName = []
MaterialColor = []
helpList = []

#Borehole Lists
boreholeDataSet = []
bhd = []
BHD = []

#Open the file in which the materials are stored and extract material data
with codecs.open (scriptPath + "\ProvIFiles\Subsoil_File.txt", 'r', 'utf-8' ) as mat_file:
    for line in mat_file:
        if (line[0] == str(2) and line[1] == str(0)):
            matData20.append(line)
        elif (line[0] == str(2) and line[1] == str(1)):
            matData21.append(line)
        elif (line == ""):
            mat_file.close()

```

```

#Continue reading file and extract borehole data
with open (scriptPath + "\\ProVIFiles\\Subsoil_File.txt") as mat_file:
    for line in mat_file:
        row = line.split(delimiter)
        if (line == ""):
            mat_file.close()
        if (line[0] == str(3) and line[1] == str(0)):
            boreholeDataSet.append("x"+row[2])
            boreholeDataSet.append(row[6].replace("\n", ""))
        elif (line[0] == str(3) and line[1] == str(1)):
            boreholeDataSet.append(row[1])
            boreholeDataSet.append(row[2].replace("\n", ""))

#Store borehole data in "array" BHD
i = 0
while (i < len(boreholeDataSet)):
    bhd.append(boreholeDataSet[i].replace("x", ""))
    i += 1
    while ((i < len(boreholeDataSet)) and (boreholeDataSet[i][0] != "x")):
        bhd.append(boreholeDataSet[i])
        i += 1

    BHD.append(bhd)
    bhd = []

#Split Material-Data in separate lists for Names, Colours and Assignment
for i in range (0, len(matData20)):
    row = matData20[i].split(delimiter)
    MaterialNumber.append(float(row[1]))
    MaterialCode.append(row[2])
    MaterialName.append(row[3].replace("\n", ""))
for i in range (0, len(matData21)):
    row = matData21[i].split(delimiter)
    helpList.append(row[1])
    MaterialColor.append(helpList[i])

#Split Colours in RGB
RGB = []
Color_delimiter = "/"
for i in range (0, len(MaterialColor)):
    RGB.append(MaterialColor[i].split(Color_delimiter))

for i in range (0, len(RGB)):
    if len(RGB[i]) < 3:
        RGB[i].append("0")
    if len(RGB[i]) < 3:
        RGB[i].append("0")

##### Creating Materials #####
Materials = []

for i in range (0, len(matData20)):
    Materials.append(g_i.soilmat())
    Materials[-1].setproperties("SoilModel", 3, "gammaUnsat", 18, "gammaSat", 20, "einit", 0.5,
    "E50ref", 10000, "EoedRef", 14107, "EurRef", 16500, "phi", 30, "cref", 3000, "k0nc", 0.680,
    "perm_primary_horizontal_axis", 0.08640E-3, "perm_vertical_axis", 0.08640E-3,
    "DrainageType", 1,
    "Gref", 10000, "MaterialName", MaterialName[i])
    g_i.Materials[-1].setcolour(int(RGB[i][0]), int(RGB[i][1]), int(RGB[i][2]))

#Add materials for later change
for i in range(0, 5):
    Materials.append(g_i.soilmat())
    Materials[-1].setproperties("SoilModel", 2, "Gref", 5000, "MaterialName", "CustomMaterial_"+str(int(i)))

#Create additional Material "FrictionBase"
Materials.append(g_i.soilmat())
Materials[-1].setproperties("SoilModel", 1, "gammaUnsat", 16, "gammaSat", 20,
    "perm_primary_horizontal_axis", 0.08640E-3, "perm_vertical_axis", 0.08640E-3,
    "Gref", 10000, "MaterialName", "Friction Base", "Colour", 600000)

#Create additional Material "Drainage Sand"
Materials.append(g_i.soilmat())
Materials[-1].setproperties("SoilModel", 1, "gammaUnsat", 16, "gammaSat", 20,
    "perm_primary_horizontal_axis", 8.640E-3, "perm_vertical_axis", 8.640E-3,
    "Gref", 10000, "MaterialName", "Drainage Layer", "Colour", 400000)

#Create additional Material "SoilReplacement"
Materials.append(g_i.soilmat())
Materials[-1].setproperties("SoilModel", 1, "gammaUnsat", 16, "gammaSat", 20,
    "perm_primary_horizontal_axis", 0.08640E-3, "perm_vertical_axis", 0.08640E-3,
    "Gref", 10000, "MaterialName", "Soil Replacement", "Colour", 600000)

#Create additional Material "Embankment"
Materials.append(g_i.soilmat())
Materials[-1].setproperties("SoilModel", 1, "gammaUnsat", 16, "gammaSat", 20,
    "perm_primary_horizontal_axis", 0.08640E-3, "perm_vertical_axis", 0.08640E-3,
    "Gref", 10000, "MaterialName", "Embankment", "Colour", 14876084)

```

```

##### OPEN QP-FILE #####
QPF = []
#Open QP-File
#Store data in list QPF
with open(scriptPath + "\\ProvIFiles\Cross_Section_File.txt", "r") as QP_file:
    for line in QP_file:
        QPF.append(line)

QP_file.close()

##### INPUT DIALOGUE #####

stations = []

print("Please edit the material properties now directly in Plaxis - press enter if finished.")
input()
print("")
copyMaterials = []
for i in range(0, len(g_i.Materials)):
    copyMaterials.append(g_i.Materials[i])

while True:
    print("For which cross section(s) do you want to generate a model?\nSpecify the station and press enter: (e.g. 3250.0)")
    print("Enter 'none' if finished.")
    inputString = input()

    if inputString == "none" or inputString == "None" or inputString == "N" or inputString == "n":
        print("")
        break
    else:
        try:
            QPstation = float(inputString)

            validQP = False
            for i in range(0, len(QPF)):
                if str(QPF[i][0:2]) == "66" and float(QPF[i][7:17]) == QPstation:
                    validQP = True
                    stations.append(QPstation)
                    print("Cross section " + str(QPstation) + " added.")
                    print("")
                    break
                else:
                    continue

            if validQP == False:
                print("The cross section you entered (" + inputString + ")\nwas not found in the Cross_Section_File.")
                print("")

        except ValueError:
            print("Could not convert your input into an integer - please try again.")
            print("")

##### Geotechnical Measures #####

##### Soil replacement #####
print("Do you want to perform a soil replacement? (Y/N)")
input1 = input()
print("")
if input1 == "Y" or input1 == "y":
    replace_soil = True
    while True:
        print("How many meters of soil should be replaced? [m](e.g. 3.0)")
        try:
            thickness_SR = float(input())
            print("")
            break
        except ValueError:
            print("Please enter a valid value for the depth of the soil replacement.")
    while True:
        print("What should be the inclination of the slopes? (horizontal/vertical) e.g.'5/3'")
        try:
            input2 = input()
            print("")
            value_hor = ""
            value_ver = ""
            i = 0
            while input2[i] != "/":
                value_hor = value_hor + input2[i]
                i += 1
            #last i-value is "/"
            for j in range(i+1, len(input2)):
                value_ver = value_ver + input2[j]
            inclination_SR = float(value_hor)/float(value_ver)
            friction_base = False
            break
        except:
            print("Please enter a valid inclination of the excavation slopes.")
    else:
        replace_soil = False

##### Friction Bases #####
if replace_soil != True:
    print("Do you want to include friction bases? (Y/N)")
    input3 = input()
    print("")
    if input3 == "Y" or input3 == "y":
        friction_base = True

```

```

while True:
    print("How long should the base length of both friction bases be? [m] (e.g. 2.5)")
    try:
        base_length_FB = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for the base length.")

while True:
    print("What length (of the base length) should be outside of the embankment area?")
    try:
        excess_length_FB = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for the outside proportion of the base length.")

while True:
    print("How deep should the friction base be? [m] (e.g. 1.0)")
    try:
        thickness_FB = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for the depth of the friction bases.")

while True:
    print("What should be the inclination of the slopes of the friction bases?\
(horizontal/vertical) e.g.'5/3'")
    try:
        input2 = input()
        print("")
        value_hor = ""
        value_ver = ""
        i = 0
        while input2[i] != "/":
            value_hor = value_hor + input2[i]
            i += 1
            #last i-value is "/"
        for j in range(i+1, len(input2)):
            value_ver = value_ver + input2[j]
        inclination_FB = float(value_hor)/float(value_ver)
        break
    except:
        print("Please enter a valid inclination for the friction base excavation slopes.")
else:
    friction_base = False

##### Drainage System #####
print("Do you want to include a drainage system? (Y/N)")
input4 = input()
print("")
if input4 == "Y" or input4 == "y":
    drainage = True
    while True:
        print("How thick should the drainage layer be? [m] (e.g. 0.5)")
        try:
            thickness_DL = float(input())
            print("")
            if thickness_DL >= 2 or thickness_DL < 0:
                raise ValueError
            break
        except:
            print("Please enter a valid value for the thickness of the drainage layer.")
    if thickness_DL == 0:
        zeroDL = True
    else:
        zeroDL = False

while True:
    print("What spacing should the drains have? [m] (e.g. 2.0)")
    try:
        spacing_DR = float(input())
        print("")
        break
    except:
        ("Please enter a valid value for the drain spacing.")

while True:
    print("How long should the drains be? [m] (e.g. 7.5)")
    try:
        length_DR = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for drain length.")
else:
    drainage = False

##### Overburden #####
print("Do you want to include an overburden? (Y/N)")
input5 = input()
print("")
if input5 == "Y" or input5 == "y":
    overburden = True
    while True:
        print("How thick should the overburden be - orthogonal to the top of the earth fill? [m] (e.g 2.0) ")
        try:
            thickness_OB = float(input())
            print("")
            break
        except:

```

```

        print("Please enter a valid value for overburden thickness.")
while True:
    print("How long does the construction of the overburden take? [days]")
    try:
        construction_OB = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for the overburden construction time.")
while True:
    print("How long until the deconstruction of the overburden? [days]")
    try:
        duration_OB = float(input())
        print("")
        break
    except:
        print("Please enter a valid value for the overburden consolidation time.")
else:
    overburden = False

##### Construction and consolidation times #####
while True:
    print("How long does the construction of each 2m-layer take? [days]")
    try:
        constructionTime = float(input())
        print("")
        break
    except:
        print("Please enter a valid time for the construction time of the 2m-layers.")
while True:
    print("How long is the consolidation time until the next 2m-layer is constructed? [days]")
    try:
        consolidationTime = float(input())
        print("")
        break
    except:
        print("Please enter a valid time for the consolidation time of the 2m-layers.")

##### START LOOP #####

for i in range(0, len(stations)):
    QPstation = stations[i]

##### Save Project #####

g_i.save(scriptPath + "\\ProjectFiles\\QP_" + str(int(QPstation)))

##### Delete everything but materials #####
if i != 0:
    g_i.gotostages()
    for i in range(1, len(g_i.Phases)):
        g_i.delete(g_i.Phases[i])
    g_i.gotostructures()
    for i in range(0, len(g_i.Polygons)):
        g_i.delete(g_i.Polygons[i])
    for i in range(0, len(g_i.Points)):
        g_i.delete(g_i.Points[i])
    for i in range(0, len(g_i.Lines)):
        g_i.delete(g_i.Lines[i])

#Lists to store QP-data in
embX = []
embY = []
embX_0 = []
embY_0 = []
embankmentX = []
embankmentY = []
embListX = []
embListY = []
topsoilX = []
topsoilY = []
oneLayerX = []
oneLayerY = []
layeringX = []
layeringY = []
subsoilX = []
subsoilY = []
helpListX = []
helpListY = []

#Iterate through QP-file data
#Create Point lists for Embankment, Topsoil and Subsoil layers

for i in range(0, len(QPF)):
    if str(QPF[i][0:2]) == "66" and float(QPF[i][7:17]) == QPstation and QPF[i][21:24] == " 3 ":
        j = i+1
        while QPF[j][0:3] != "66 ":
            embX.append(float(QPF[j][:10]))
            embY_0.append(float(QPF[j][11:22]))
            j += 1
        i0 = embX.index(0.0)
        heightCenterPoint = embY_0[i0]

```

```

for k in range (len(embY_0)):
    embY.append(embY_0[k]-embY_0[i0])

if str(QPF[i][0:2]) == "66" and float(QPF[i][7:17]) == QPstation and QPF[i][20:24] == " 21 ":
    j = i+1
    while QPF[j][0:3] != "66 ":
        topsoilX.append(float(QPF[j][:10]))
        topsoilY.append(float(QPF[j][11:22])-embY_0[i0])
        j += 1

if str(QPF[i][0:2]) == "66" and float(QPF[i][7:17]) == QPstation and QPF[i][20:24] == " 42 ":
    j = i+1
    while QPF[j][0:3] != "66 ":
        try:
            subsoilX.append(float(QPF[j][:10]))
            subsoilY.append(float(QPF[j][11:22])-embY_0[i0])
        except:
            print("Empty line in QP-file detected")

        j += 1
    if j == len(QPF):
        break

if str(QPF[i][0:2]) == "66" and float(QPF[i][7:17]) == QPstation and QPF[i][20:24] == " 51 ":
    j = i+1
    while QPF[j][0:3] != "66 ":
        embankmentX.append(float(QPF[j][:10]))
        embankmentY.append(float(QPF[j][11:22]) - embY_0[i0])
        j += 1
    if j == len(QPF):
        break

#Modify Embankment Polygon
topIndex = embankmentY.index(max(embankmentY))

i = 0
while topIndex - i > 0:
    j = 0
    while topsoilX[j] < embankmentX[topIndex-i]:
        j += 1
    compValue = topsoilY[j-1] + (topsoilY[j]-topsoilY[j-1])/(topsoilX[j]-topsoilX[j-1])\
    *(embankmentX[topIndex-i]-topsoilX[j-1])
    helpListX.append(embankmentX[topIndex-i])
    helpListY.append(embankmentY[topIndex-i])
    if abs(compValue-embankmentY[topIndex-i]) < 0.001:
        break
    i += 1

i = 1
helpListX2 = []
helpListY2 = []
while topIndex + i < len(embankmentY):
    k = 0
    while topsoilX[k] < embankmentX[topIndex+i]:
        k += 1
    compValue = topsoilY[k-1] + (topsoilY[k]-topsoilY[k-1])/(topsoilX[k]-topsoilX[k-1])\
    *(embankmentX[topIndex+i]-topsoilX[k-1])
    helpListX2.append(embankmentX[topIndex+i])
    helpListY2.append(embankmentY[topIndex+i])
    if abs(compValue-embankmentY[topIndex+i]) < 0.001:
        break
    i += 1

for i in range (0, len(helpListX)):
    embListX.append(helpListX[i])
    embListY.append(helpListY[i])

for i in range (j, k):
    embListX.append(topsoilX[i])
    embListY.append(topsoilY[i])

m = len(helpListX2)-1
while m >= 0:
    embListX.append(helpListX2[m])
    embListY.append(helpListY2[m])
    m -= 1

#Combine all subsoil layers in one array called: "layeringX/layeringY"
i = 0
while i < len(subsoilX):
    oneLayerX = []
    oneLayerY = []
    j = i
    while (j < len(subsoilX)-1) and subsoilX[j] != 100.0:
        oneLayerX.append(subsoilX[j])
        oneLayerY.append(subsoilY[j])
        j += 1
    if subsoilX[j] == 100.0:
        oneLayerX.append(subsoilX[j])
        oneLayerY.append(subsoilY[j])

    i = j + 1
    layeringX.append(oneLayerX)
    layeringY.append(oneLayerY)

```

```

##### Get Topsoil Polygon #####
topPolygonX = []
topPolygonY = []

p1x = getPoint(topsoilX, topsoilY, -50)[0]
p1y = getPoint(topsoilX, topsoilY, -50)[1]
p2x = getPoint(layeringX[0], layeringY[0], -50)[0]
p2y = getPoint(layeringX[0], layeringY[0], -50)[1]

p3x = getPoint(layeringX[0], layeringY[0], 50)[0]
p3y = getPoint(layeringX[0], layeringY[0], 50)[1]
p4x = getPoint(topsoilX, topsoilY, 50)[0]
p4y = getPoint(topsoilX, topsoilY, 50)[1]

topPolygonX.append(p1x)
topPolygonY.append(p1y)
topPolygonX.append(p2x)
topPolygonY.append(p2y)

for i in range(0, len(layeringX[0])):
    if layeringX[0][i] > -50.0 and layeringX[0][i] < 50.0:
        topPolygonX.append(layeringX[0][i])
        topPolygonY.append(layeringY[0][i])

topPolygonX.append(p3x)
topPolygonY.append(p3y)
topPolygonX.append(p4x)
topPolygonY.append(p4y)

topsoilX.reverse()
topsoilY.reverse()

for i in range(0, len(topsoilX)):
    if topsoilX[i] > -50.0 and topsoilX[i] < 50.0:
        topPolygonX.append(topsoilX[i])
        topPolygonY.append(topsoilY[i])

#Get bottom value for x=0
yMin = getPoint(layeringX[-1], layeringY[-1], 0)[1]

##### Get subsoil Polygons #####
subsoilPolListX = []
subsoilPolListY = []

for i in range(0, len(layeringX)):
    if i + 1 >= len(layeringX):
        break
    subsoilPolX = []
    subsoilPolY = []

    subsoilPolX.append(getPoint(layeringX[i], layeringY[i], -50)[0])
    subsoilPolY.append(getPoint(layeringX[i], layeringY[i], -50)[1])
    subsoilPolX.append(getPoint(layeringX[i+1], layeringY[i+1], -50)[0])
    subsoilPolY.append(getPoint(layeringX[i+1], layeringY[i+1], -50)[1])

    for j in range(0, len(layeringX[i+1])):
        if layeringX[i+1][j] > -50.0 and layeringX[i+1][j] < 50.0:
            subsoilPolX.append(layeringX[i+1][j])
            subsoilPolY.append(layeringY[i+1][j])

    subsoilPolX.append(getPoint(layeringX[i+1], layeringY[i+1], 50)[0])
    subsoilPolY.append(getPoint(layeringX[i+1], layeringY[i+1], 50)[1])

    subsoilPolX.append(getPoint(layeringX[i], layeringY[i], 50)[0])
    subsoilPolY.append(getPoint(layeringX[i], layeringY[i], 50)[1])

    layeringX[i].reverse()
    layeringY[i].reverse()

    for j in range(0, len(layeringX[i])):
        if layeringX[i][j] > -50.0 and layeringX[i][j] < 50.0:
            subsoilPolX.append(layeringX[i][j])
            subsoilPolY.append(layeringY[i][j])

    subsoilPolListX.append(subsoilPolX)
    subsoilPolListY.append(subsoilPolY)

##### Get complete polygon array #####
polygonListX = []
polygonListY = []

polygonListX.append(embListX)
polygonListY.append(embListY)
polygonListX.append(topPolygonX)
polygonListY.append(topPolygonY)

for i in range(0, len(subsoilPolListX)):
    polygonListX.append(subsoilPolListX[i])
    polygonListY.append(subsoilPolListY[i])

```

```
##### Modify Polygons #####
#delete "Stich-strecken"

for i in range (0, len(polygonListX)):
    for j in range (1, len(polygonListX[i])):
        v1 = polygonListX[i][j-1]
        v2 = polygonListX[i][j]
        v3 = polygonListX[i][j+1]
        w1 = polygonListY[i][j-1]
        w2 = polygonListY[i][j]
        w3 = polygonListY[i][j+1]

        #delete middle point if x coords are three times the same
        if v1 == v2 and v1 == v3:
            del polygonListX[i][j]
            del polygonListY[i][j]

        #delete middle point if y coords are three times the same
        if w1 == w2 and w1 == w3:
            del polygonListX[i][j]
            del polygonListY[i][j]

        if j == len(polygonListX[i])-2:
            break

#delete identical points
#for i in range (0, len(polygonListX)):
for j in range (1, len(polygonListX[0])-1):
    v1 = polygonListX[0][j-1]
    v2 = polygonListX[0][j]
    w1 = polygonListY[0][j-1]
    w2 = polygonListY[0][j]

    #delete points if they are identical
    if abs(v1 - v2) < 0.00001 or abs(w1 - w2) < 0.00001:
        del polygonListX[0][j]
        del polygonListY[0][j]
    if j == len(polygonListX[0])-1:
        break

#Modify points maximum and minimum x-coords of embankment
#lower them by a milimeter in order to fix a sometimes occuring numerical problem
iMax = polygonListX[0].index(max(polygonListX[0]))
iMin = polygonListX[0].index(min(polygonListX[0]))
polygonListY[0][iMax] = polygonListY[0][iMax] -0.001
polygonListY[0][iMin] = polygonListY[0][iMin] -0.001

#Make bottom a straight, horizontal line
yBottom = min(polygonListY[-1])
i50 = polygonListX[-1].index(50.0)
polygonListY[-1][1] = yBottom
polygonListY[-1][i50] = yBottom
del polygonListY[-1][2:i50]
del polygonListX[-1][2:i50]

#Get real height of last polygon
#For later use in material assignment
upperPoints = polygonListY[-1][3:]
upperPoints.append(polygonListY[-1][0])
minimum = min(upperPoints)
heightLastLayer = minimum - yBottom

##### Create Polygons in Plaxis #####

#Use function getPolygonCommand to create the embankment polygons
g_1.gotostructures()

for m in range (0, len(polygonListX)):
    exec(getPolygonCommand(polygonListX[m],polygonListY[m]))

##### Material Assignment #####
#Find boreholes closest to QPstation
for i in range (0, len(BHD)):
    for j in range (0, len(BHD[i])):
        BHD[i][j] = float (BHD[i][j])
        QP_borehole = (BHD[i][0] - QPstation)
        if (QP_borehole > 0):
            iRight = i
            iLeft = i-1
            BRight = BHD[i][0]
            BLeft = BHD[i-1][0]
            dL = QPstation - BLeft
            dR = BRight - QPstation
            dGes = dL+dR
            break

#Variables
maxY = []

#Polygon which has Point with abs. Coord (0,0) gets assigned no material so far
#Other Polygons get assigned material starting from top
#Get absolute Polygon coords for all polygons
for i in range (0 , len(g_i.Polygons)):
```

```

abs_CoordsY = []
for j in range (0, len(g_i.Polygons[i].Points)):
    x_abs = g_i.Polygons[i].x + g_i.Polygons[i].Points[j].x
    y_abs = g_i.Polygons[i].y + g_i.Polygons[i].Points[j].y
    abs_CoordsY.append(y_abs)

maxY.append(max(abs_CoordsY))

indexEmb = 0
newY = removeElement(maxY, indexEmb)
sortednewY = sorted(newY, reverse = True)

#For result points: get index of Topsoil
ITS = maxY.index(sortednewY[0])

#Get indeces of polygons below imbankment
indec = []
for i in range (0, len(sortednewY)):
    indec.append(maxY.index(sortednewY[i]))

#Start assignment from top-polygon
#Assign Material according to height
#Start values:
jLeft = 0
jRight = 0

for i in range (0, len(newY)):

    pIndex = maxY.index(sortednewY[i])
    h = layerHeight(pIndex)
    if i == len(newY) -1:
        h = heightLastLayer

    if (3+jLeft*2) < len(BHD[iLeft]) and abs(dR/dGes*BHD[iLeft][3 + jLeft*2] - h) < 0.001:
        matIndex = MaterialNumber.index(BHD[iLeft][2+jLeft*2])
        g_i.Soils[pIndex].Material = Materials[matIndex]
        jLeft += 1
    elif abs(dL/dGes*BHD[iRight][3 + jRight*2] - h) < 0.001:
        matIndex = MaterialNumber.index(BHD[iRight][2+jRight*2])
        g_i.Soils[pIndex].Material = Materials[matIndex]
        jRight += 1
    elif (abs(BHD[iLeft][3 + jLeft*2] + dL/dGes\
    *(BHD[iRight][3 + jRight*2]-BHD[iLeft][3 + jLeft*2]) - h) < 0.001):
        matIndex = MaterialNumber.index(BHD[iLeft][2 + jLeft*2])
        g_i.Soils[pIndex].Material = Materials[matIndex]
        jLeft += 1
        jRight += 1

##### Division of Embankment #####

#Start point is point of Embankment with x-min
absEMBCoordsX = []
absEMBCoordsY = []
PolPointsListX = []
PolPointsListY = []

#Get absolute embankment coordinates
#not necessary anymore (due to QP-file)
for j in range (0, len(g_i.Polygons[indexEmb].Points)):

    x_abs = g_i.Polygons[indexEmb].x + g_i.Polygons[indexEmb].Points[j].x
    y_abs = g_i.Polygons[indexEmb].y + g_i.Polygons[indexEmb].Points[j].y
    absEMBCoordsX.append(x_abs)
    absEMBCoordsY.append(y_abs)

#Reorder lists to start with highest embankment point
betterOrderX = []
betterOrderY = []

for i in range (0, len(absEMBCoordsX)):
    if abs(absEMBCoordsX[i] - polygonListX[0][polygonListY[0].index(max(polygonListY[0]))]) < 0.0001 \
    and abs(absEMBCoordsY[i] - max(polygonListY[0])) < 0.0001:
        i0 = i

for i in range (i0, len(absEMBCoordsX)):
    betterOrderX.append(absEMBCoordsX[i])
    betterOrderY.append(absEMBCoordsY[i])

for i in range (0, i0):
    betterOrderX.append(absEMBCoordsX[i])
    betterOrderY.append(absEMBCoordsY[i])

#Get indeces of x-min and x-max
iBL = betterOrderX.index(min(betterOrderX))
iBR = betterOrderX.index(max(betterOrderX))

#Get coordinates of subpolygons
hBottom = getPoint(embListX, embListY, 0)[1]

if drainage == True and zeroDL == False:
    hTarget = max(hBottom + thickness_DL, betterOrderY[iBL] +0.1, betterOrderY[iBR] + 0.1)
    hDrain = hTarget - hBottom
else:

```

```

    hTarget = hBottom + 2

#Note: thickness_DL is only taken if it is enough to cover all the area of the bottom embankment
#Otherwise the drainage is made 10cm higher than the points on the right and the left of the embankment

#in case Embankment is smaller ehan 2m:
if hTarget > max(embListY):
    PolPointsListX.append(embListX)
    PolPointsListY.append(embListY)

#Number of added Polygons
NoaP = 0

while hTarget < max(embListY):
    #Get start values
    iBL = betterOrderX.index(min(betterOrderX))
    iBR = betterOrderX.index(max(betterOrderX))

    PolygonPointsY = []
    PolygonPointsX = []

    i = 0
    #Get Coords until hTarget
    while betterOrderY[iBL-i] < hTarget:
        PolygonPointsX.append(betterOrderX[iBL-i])
        PolygonPointsY.append(betterOrderY[iBL-i])
        iLast = iBL-i
        i += 1

        if (iBL - i) == -1:
            break

    #Get Point on hTarget
    iNext = iLast - 1
    xHelp1 = (betterOrderX[iNext]-betterOrderX[iLast])\
    / (betterOrderY[iNext]-betterOrderY[iLast])*(hTarget-betterOrderY[iLast])
    PolygonPointsX.append(betterOrderX[iLast]+xHelp1)
    PolygonPointsY.append(hTarget)

    #Reverse list
    PolygonPointsX.reverse()
    PolygonPointsY.reverse()

    j= 1
    #Get Coords until hTarget for other part
    while betterOrderY[iBL+j] < hTarget:
        PolygonPointsX.append(betterOrderX[iBL+j])
        PolygonPointsY.append(betterOrderY[iBL+j])
        jLast = iBL+j
        j += 1

        #test: previous len(betterOrderY) +1 in next line
        if (iBL + j) == (len(betterOrderY)-1):
            break

    #Get Point on hTarget
    jNext = jLast + 1
    xHelp2 = (betterOrderX[jNext]-betterOrderX[jLast])\
    / (betterOrderY[jNext]-betterOrderY[jLast])*(hTarget-betterOrderY[jLast])
    PolygonPointsX.append(betterOrderX[jLast]+xHelp2)
    PolygonPointsY.append(hTarget)

    #Modify List of embankment coordinates for next section:
    #Replace last coordinates with coordinates at hTarget
    #Delete all points that are only part of lower polygon
    betterOrderX[iLast] = betterOrderX[iLast]+xHelp1
    betterOrderY[iLast] = hTarget
    betterOrderX[jLast] = betterOrderX[jLast]+xHelp2
    betterOrderY[jLast] = hTarget
    del betterOrderX[(iLast+1):(jLast)]
    del betterOrderY[(iLast+1):(jLast)]

    #Get new target height
    if drainage == True and NoaP == 0 and zeroDL == False:
        hTarget = hTarget + (2-hDrain)
    else:
        hTarget += 2

    NoaP += 1

    #Store PolygonPoints in array
    PolPointsListX.append(PolygonPointsX)
    PolPointsListY.append(PolygonPointsY)

    #Include top polygon if hTarget is greater than 0
    if hTarget > max(embListY):
        PolygonPointsX = []
        PolygonPointsY = []

        for k in range(0, (len(betterOrderX)-1)):
            PolygonPointsX.append(betterOrderX[k])
            PolygonPointsY.append(betterOrderY[k])

        PolPointsListX.append(PolygonPointsX)

```

```

PolPointsListY.append(PolygonPointsY)

##### Drainage #####
#Create array with coords for Drains
while True:
    drainsList = []
    if drainage == True and zeroDL == False:
        yTop_DR = hBottom + hDrain - 0.01
        yBottom_DR = yTop_DR - length_DR
        xLeft_DR = PolPointsListX[0][0]
        xRight_DR = PolPointsListX[0][-1]
        NumberOfDrains = floor((xRight_DR - xLeft_DR)/spacing_DR)
        xDR = xLeft_DR + ((xRight_DR - xLeft_DR)-(NumberOfDrains - 1) * spacing_DR)/2
        for i in range (0, NumberOfDrains):
            helpListDrains = []
            helpListDrains.append(xDR)
            helpListDrains.append(yTop_DR)
            helpListDrains.append(xDR)
            helpListDrains.append(yBottom_DR)
            drainsList.append(helpListDrains)
            xDR += spacing_DR

        yMin_DR = yBottom_DR
        yMin_Top_DR = yTop_DR

    elif drainage == True and zeroDL == True:
        xLeft_DR = min(PolPointsListX[0])
        xRight_DR = max(PolPointsListX[0])
        NumberOfDrains = floor((xRight_DR - xLeft_DR)/spacing_DR)
        xDR = xLeft_DR + ((xRight_DR - xLeft_DR)-(NumberOfDrains - 1) * spacing_DR)/2
        topsoilPointsX = PolPointsListX[0].index(xLeft_DR):PolPointsListX[0].index(xRight_DR)+1
        topsoilPointsY = PolPointsListY[0][PolPointsListX[0].index(xLeft_DR):PolPointsListX[0].index(xRight_DR)+1]

        for i in range (0, NumberOfDrains):
            helpListDrains = []
            yBottomList = []
            yTopList = []
            helpListDrains.append(xDR)

            yTop_DR = getPoint(topsoilPointsX, topsoilPointsY,xDR)[1]- 0.01
            helpListDrains.append(yTop_DR)
            helpListDrains.append(xDR)
            yBottom_DR = yTop_DR - length_DR
            helpListDrains.append(yBottom_DR)
            drainsList.append(helpListDrains)
            xDR += spacing_DR
            yBottomList.append(yBottom_DR)
            yTopList.append(yTop_DR)

        yMin_DR = min(yBottomList)
        yMin_Top_DR = min(yTopList)

    if drainage == True and yMin_DR < yBottom:
        maxLenght = yMin_Top_DR - yBottom
        print("Drain lenght has been selected too long - drains exceed bottom model boundary.")
        print("Please enter shorter drain lenght to proceed.")
        while True:
            print("How long should the drains be? [m] (max lenght:" +str(maxLenght)+ ")")
            try:
                length_DR = float(input())
                print("")
                break
            except:
                print("Please enter a valid value for drain lenght.")

    else:
        break

##### Setup Initial Phase #####

g_i.delete(g_i.Polygons[indexEmb])
g_i.gotostages()

#Activate all non-embankment polygons in initial phase
for i in range (0, len(g_i.Polygons)):
    BBox = getBoundingBox(g_i.Polygons[i])

    if BBox[0] == -50.0 and BBox[2] == 50.0:
        g_i.Soils[i].Active[g_i.Phases[0]] = True

g_i.gotostructures()

##### Soil replacement #####
if replace_soil == True:

    p2x = min(polygonListX[0])
    p2y = polygonListY[0][polygonListX[0].index(p2x)] - thickness_SR

```

```

p3x = max(polygonListX[0])
p3y = polygonListY[0][polygonListX[0].index(p3x)] - thickness_SR

p1x = p2x + p2y*inclination_SR
p1y = 0

p4x = p3x + p3y*-inclination_SR
p4y = 0

g_i.line(p1x, p1y, p2x, p2y)
g_i.line(p2x, p2y, p3x, p3y)
g_i.line(p3x, p3y, p4x, p4y)

#Boundaries where soil is replaced
xmin_SR = -49
xmax_SR = 49
ymin_SR = min(p2y-0.01, p3y-0.01)

##### Friction Bases #####
if friction_base == True:

#Left friction base
p2x = min(polygonListX[0]) - excess_lenght_FB
p2y = polygonListY[0][polygonListX[0].index(min(polygonListX[0]))] - thickness_FB

p3x = p2x + base_length_FB
p3y = p2y

p1x = p2x + p2y *inclination_FB
p1y = 0

if drainage == True and zeroDL == False:
    p4y = hBottom + hDrain -0.01
else:
    p4y = hBottom + 1.99

p4x = p3x + (p4y-p3y)*inclination_FB

g_i.line(p1x, p1y, p2x, p2y)
g_i.line(p2x, p2y, p3x, p3y)
g_i.line(p3x, p3y, p4x, p4y)

#Boundaries where soil is replaced
xmin_FB1 = p1x
xmax_FB1 = p4x
ymin_FB1 = p2y - 0.01

#Right friction base
p2x = max(polygonListX[0]) + excess_lenght_FB - base_length_FB
p2y = polygonListY[0][polygonListX[0].index(max(polygonListX[0]))] - thickness_FB

p3x = p2x + base_length_FB
p3y = p2y

if drainage == True and zeroDL == False:
    p1y = hBottom + hDrain -0.01
else:
    p1y = hBottom + 1.99

p1x = p2x - (p1y-p2y)*inclination_FB

p4x = p3x - p3y*inclination_FB
p4y = 0

g_i.line(p1x, p1y, p2x, p2y)
g_i.line(p2x, p2y, p3x, p3y)
g_i.line(p3x, p3y, p4x, p4y)
#Boundaries where soil is replaced
xmin_FB2 = p1x
xmax_FB2 = p4x
ymin_FB2 = p2y - 0.01

##### Overburden #####
if overburden == True:

#Find Top right and top left point of embankment
#Characterised through:
# a) more than 30degree inclincation to next point
# b) longer distance than 15cm to next point
for i in range (0, len(polygonListX[0])):
    deltax = abs(polygonListX[0][i]-polygonListX[0][i+1])
    deltax = abs(polygonListY[0][i]-polygonListY[0][i+1])
    distance = (deltax**2 + deltax**2)**(1/2)
    if deltax/deltax > 0.466 and distance > 0.15:
        topLeft = i
        break

for i in range (0, len(polygonListX[0])):
    j = len(polygonListX[0]) - 1 - i
    deltax = abs(polygonListX[0][j]-polygonListX[0][j-1])
    deltax = abs(polygonListY[0][j]-polygonListY[0][j-1])
    distance = (deltax**2 + deltax**2)**(1/2)
    if deltax/deltax > 0.5 and distance > 0.15:
        topRight = j
        break

```

```

alpha = atan((polygonListY[0][topRight]-polygonListY[0][topLeft])\
/(polygonListX[0][topRight]-polygonListX[0][topLeft]))

betaLeft = atan((polygonListY[0][topLeft]-polygonListY[0][topLeft+1])\
/(polygonListX[0][topLeft]-polygonListX[0][topLeft+1]))
hypLeft = thickness_OB/(sin(betaLeft+alpha))
ob1x= polygonListX[0][topLeft]+hypLeft*cos(betaLeft)
ob1y= polygonListY[0][topLeft]+hypLeft*sin(betaLeft)

betaRight = atan((polygonListY[0][topRight]-polygonListY[0][topRight-1])\
/(polygonListX[0][topRight-1]-polygonListX[0][topRight]))
hypRight = thickness_OB/(sin(betaRight-alpha))
ob2x= polygonListX[0][topRight]-hypRight*cos(betaRight)
ob2y= polygonListY[0][topRight]+hypRight*sin(betaRight)

obListX = []
obListY = []
obListX.append(ob2x)
obListY.append(ob2y)
obListX.append(ob1x)
obListY.append(ob1y)

i = topLeft
while i >= 0:
    obListX.append(polygonListX[0][i])
    obListY.append(polygonListY[0][i])
    i -= 1

j = len(polygonListX[0])-1
while j >= topRight:
    obListX.append(polygonListX[0][j])
    obListY.append(polygonListY[0][j])
    j -= 1

##### Create new embankment polygons #####
#Use function getPolygonCommand to create the embankment polygons
#Assign embankment material to them
g_i.gotostructures()
if drainage == True and zeroDL == False:
    exec(getPolygonCommand(PolPointsListX[0],PolPointsListY[0]))
    g_i.set(g_i.Polygons[-1].Name, "Drainage")
    g_i.Soils[-1].Material = Materials[-3]
    for m in range (1, (len(PolPointsListX))):
        exec(getPolygonCommand(PolPointsListX[m],PolPointsListY[m]))
        layerName = "Layer_" + str(m)
        g_i.set(g_i.Polygons[-1].Name, layerName)
        g_i.Soils[-1].Material = Materials[-1]
        NumberLayers = m
else:
    for m in range (0, (len(PolPointsListX))):
        exec(getPolygonCommand(PolPointsListX[m],PolPointsListY[m]))
        layerName = "Layer_" + str(m+1)
        g_i.set(g_i.Polygons[-1].Name, layerName)
        g_i.Soils[-1].Material = Materials[-1]
        NumberLayers = m+1

if overburden == True:
    exec(getPolygonCommand(obListX,obListY))
    g_i.set(g_i.Polygons[-1].Name, "Overburden")
    g_i.Soils[-1].Material = Materials[-1]

##### Find points on topsoil #####
#Get absolute coordinates
#NOTE:
#in Plaxis the Polygon Points are shown with the correct Coords
#however in the script the Polygon Points are given as "Helper Points"
#whose Coords are relative Coords form the Polygon Point
absPolygonCoordsX = []
absPolygonCoordsY = []
inOrderX = []
newOrderY = []
newOrderX = []

#Polygons[iTS] is the Topsoil
#make a list with absolute x-coords of polygon points
#make a list with absolute y-coords of polygon points

for i in range (0,len(g_i.Polygons[iTS].Points)):
    absPolygonCoordsX.append(g_i.Polygons[iTS].x + g_i.Polygons[iTS].Points[i].x)
    absPolygonCoordsY.append(g_i.Polygons[iTS].y + g_i.Polygons[iTS].Points[i].y)

indexXmax = absPolygonCoordsX.index(max(absPolygonCoordsX))
indexXmin = absPolygonCoordsX.index(min(absPolygonCoordsX))

if indexXmax < indexXmin:
    for j in range(indexXmax, (indexXmin+1)):
        newOrderY.append(absPolygonCoordsY[j])
        newOrderX.append(absPolygonCoordsX[j])
        #reorder list starting from bottom left point

elif indexXmin < indexXmax:
    for j in range (indexXmax, len(absPolygonCoordsX)):
        newOrderY.append(absPolygonCoordsY[j])
        newOrderX.append(absPolygonCoordsX[j])

```

```

    for j in range (0, (indexXmin+1)):
        newOrderY.append(absPolygonCoordsY[j])
        newOrderX.append(absPolygonCoordsX[j])

if newOrderX[0] == newOrderX[1]:
    del newOrderX[0]
    del newOrderY[0]

if newOrderX[-1] == newOrderX[-2]:
    del newOrderX[-1]
    del newOrderY[-1]

#Get surface Points
#get all points on the upper topsoil

#distance in which the displacements should be calculated
dx = 1

surfacePointsX = []
surfacePointsY = []
for i in range (0, (len(newOrderX)-1)):
    surfacePointsX.append(newOrderX[i])
    surfacePointsY.append(newOrderY[i])
    j = 1
    #while loop: looks at following point and adds a point every dx meters
    #following the upper contour of the topsoil until the next point is reached
    while ((newOrderX[i] - dx * j) > newOrderX[i+1]):
        surfacePointsX.append(newOrderX[i] - dx*j)
        surfacePointsY.append(newOrderY[i] + dx*j*(newOrderY[i+1]-newOrderY[i])\
        / (newOrderX[i] - newOrderX[i+1]-0.00001))
        j += 1
surfacePointsX.append(newOrderX[-1]) #adds last point
surfacePointsY.append(newOrderY[-1])

#in surfacePointsX and surfacePointsY are now stored all the points that lie on the topsoil

##### Set Phases #####

##### Activation List #####
#List in which order to activate embankment Polygons
g_i.gotostages()

#Model Soil replacement (if wanted)
if replace_soil == True:
    g_i.phase(g_i.Phases[-1])
    g_i.Phases[-1].Identification = "Soil Replacement"
    g_i.Phases[-1].TimeInterval = 2
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    for j in range ((0), len(g_i.Polygons)):
        BBox = getBoundingBox(g_i.Polygons[j])
        if drainage == True and zeroDL == False:
            if BBox[0] > xmin_SR and BBox[1] > ymin_SR \
            and BBox[2] < xmax_SR and BBox[3] < round(hBottom + hDrain,2):
                g_i.setmaterial(g_i.Soils[j], g_i.Phases[-1], Materials[-2])
            elif BBox[0] > xmin_SR and BBox[1] > ymin_SR \
            and BBox[2] < xmax_SR and BBox[3] < round(hBottom +2,2):
                g_i.setmaterial(g_i.Soils[j], g_i.Phases[-1], Materials[-2])

#Model friction bases (if wanted)
if friction_base == True:
    g_i.phase(g_i.Phases[-1])
    g_i.Phases[-1].Identification = "Friction Bases"
    g_i.Phases[-1].TimeInterval = 2
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    for j in range (0, len(g_i.Polygons)):
        BBox = getBoundingBox(g_i.Polygons[j])
        if BBox[0] > xmin_FB1 and BBox[1] > ymin_FB1 and BBox[2] < xmax_FB1:
            g_i.setmaterial(g_i.Soils[j], g_i.Phases[-1], Materials[-4])
        if BBox[0] > xmin_FB2 and BBox[1] > ymin_FB2 and BBox[2] < xmax_FB2:
            g_i.setmaterial(g_i.Soils[j], g_i.Phases[-1], Materials[-4])

#Add drainage (if wanted)
#set up the drains only there so that there is no problem with the bounding boxes of the friction bases
if drainage == True:

    g_i.phase(g_i.Phases[-1])

    g_i.Phases[-1].Identification = "Drainage Construction"
    g_i.Phases[-1].TimeInterval = 2
    g_i.Phases[-1].DeformCalcType = "Consolidation"

    if zeroDL == False:
        g_i.Soils[getIndex(hBottom+hDrain, min(PolPointsListX[0]), \
        max(PolPointsListX[0]))].Active[g_i.Phases[-1]] = True

    g_i.gotostages()

    for j in range (0, len(drainsList)):
        g_i.drain(drainsList[j][0],drainsList[j][1], drainsList[j][2], drainsList[j][3])

    g_i.gotostages()

#activate drains
for k in range (0, len(g_i.Drains)):
    g_i.Drains[k].Active[g_i.Phases[-1]] = True

```

```

#Add two phases for every additional embankment step and activate one embankment layer
heightEmb = 2.0

parentPhasesConsolidation = []
parentPhasesSafety = []

activationLayer = 1
while activationLayer < NumberLayers:
    g_i.phase(g_i.Phases[-1])
    parentPhasesSafety.append(str(g_i.Phases[-1].Name))
    phaseID = "Embankment " + str(heightEmb) + "m Construction"
    phaseID1 = "Embankment " + str(heightEmb) + "m Consolidation"
    g_i.Phases[-1].Identification = phaseID
    g_i.Phases[-1].TimeInterval = constructionTime
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    activateLayer = "g_i.activate(g_i.Layer_"+str(activationLayer)+"", g_i.Phases[-1])"
    exec(activateLayer)
    activationLayer += 1

    g_i.phase(g_i.Phases[-1])
    parentPhasesConsolidation.append(str(g_i.Phases[-1].Name))
    g_i.Phases[-1].Identification = phaseID1
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    g_i.Phases[-1].TimeInterval = consolidationTime
    heightEmb += 2

#Add phases for final layer
g_i.phase(g_i.Phases[-1])
nameOfFinalConstructionPhase = str(g_i.Phases[-1].Name)
g_i.Phases[-1].Identification = "Embankment Final Construction"
g_i.Phases[-1].DeformCalcType = "Consolidation"
g_i.Phases[-1].TimeInterval = constructionTime
activateLayer = "g_i.activate(g_i.Layer_"+str(NumberLayers)+"", g_i.Phases[-1])"
exec(activateLayer)

g_i.phase(g_i.Phases[-1])
g_i.Phases[-1].Identification = "Embankment Final Consolidation"
g_i.Phases[-1].DeformCalcType = "Consolidation"
g_i.Phases[-1].TimeInterval = consolidationTime

#Add phases for Overburden
if overburden == True:
    g_i.phase(g_i.Phases[-1])
    g_i.Phases[-1].Identification = "Overburden Construction"
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    g_i.Phases[-1].TimeInterval = construction_OB
    g_i.activate(g_i.Overburden, g_i.Phases[-1])

    g_i.phase(g_i.Phases[-1])
    g_i.Phases[-1].Identification = "Overburden Consolidation"
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    g_i.Phases[-1].TimeInterval = duration_OB

    g_i.phase(g_i.Phases[-1])
    g_i.Phases[-1].Identification = "Overburden Deconstruction"
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    g_i.Phases[-1].TimeInterval = construction_OB
    g_i.deactivate(g_i.Overburden, g_i.Phases[-1])

#Add U90-Phase
g_i.phase(g_i.Phases[-1])
g_i.Phases[-1].Identification = "U90"
g_i.Phases[-1].DeformCalcType = "Consolidation"
g_i.Phases[-1].Deform.LoadingType = "Degree of Consolidation"
g_i.Phases[-1].Deform.Loading.ConsolidDegree = 90

#Add min. excess Pore Water Pressure Phase
#g_i.phase(g_i.Phases[-1])
#g_i.Phases[-1].Identification = "Minimum Excess Pore Pressure"
#g_i.Phases[-1].DeformCalcType = "Consolidation"
#g_i.Phases[-1].Deform.LoadingType = "Minimum excess pore pressure"
#g_i.Phases[-1].Deform.Loading.PStop = 1

#Add Safety Phase
getPhase = "g_i.phase(g_i."+ nameOfFinalConstructionPhase +")"
exec(getPhase)
g_i.Phases[-1].Identification = "Safety"
g_i.Phases[-1].DeformCalcType = "Safety"
g_i.Phases[-1].Deform.LoadingType = "Incremental multipliers"
g_i.Phases[-1].Deform.Loading.Msf = 0.1

#Add two subphases for every sublayer
heightEmb = 2.0

for i in range(0, len(parentPhasesConsolidation)):
    getPhase = "g_i.phase(g_i."+ parentPhasesConsolidation[i] +")"
    exec(getPhase)
    phaseID = "Embankment " + str(heightEmb) + "m U95"
    phaseID1 = "Embankment " + str(heightEmb) + "m Safety"
    g_i.Phases[-1].Identification = phaseID
    g_i.Phases[-1].DeformCalcType = "Consolidation"
    g_i.Phases[-1].Deform.LoadingType = "Degree of Consolidation"

```

```

g_i.Phases[-1].Deform.Loading.ConsolidDegree = 95

getPhase = "g_i.phase(g_i." + parentPhasesSafety[i] + ")"
exec(getPhase)
g_i.Phases[-1].Identification = phaseID1

g_i.Phases[-1].Deform.CalcType = "Safety"
g_i.Phases[-1].Deform.LoadingType = "Incremental multipliers"
g_i.Phases[-1].Deform.Loading.Msf = 0.1
heightEmb += 2

##### Last Pause before Calculation #####
print("How fine should the mesh be? - Enter a value for 'Relative Element Size Factor'")
print("Values between 0.01 and 0.1 might lie in a reasonable range.")
while True:
    try:
        resfString = input()
        resf = float(resfString)
        print("")
        break
    except ValueError:
        print("Could not convert your input (" + resfString + ") \
to a valid value for 'Relative Element Size Factor'.")
        print("How fine should the mesh be? - Enter a value for 'Relative Element Size Factor'")

print("Please set a value for the 'Local Coarseness Factor' for the area around the embankment.")
print("Values between 0.03125 (very fine!) and 1.0 (no refinement) might lie in a reasonable range.")
while True:
    try:
        lcfString = input()
        lcf = float(lcfString)
        if lcf < 0.03125 or lcf > 8.0:
            raise ValueError
        print("")
        break
    except ValueError:
        print("Could not convert your input (" + lcfString + ") \
to a valid value for 'Local Coarseness Factor'.")
        print("Please specify a value for 'Local Coarseness Factor'")

print("Are all properties set correctly? - \
Please check all settings directly in Plaxis. Last pause before start of calculation.")
print("Continue with enter")
input()
print("")

##### Meshing #####

if True:
    g_i.gotostuctures()
    p1x = min(polygonListX[0]) -8
    p1y = 0
    p2x = p1x
    p2y = max(polygonListY[0][polygonListX[0].index(min(polygonListX[0]))] -8, min(polygonListY[-1])+1)

    if drainage == True:
        p2y = yMin_DR - (yMin_DR - min(polygonListY[-1]))*0.2

    p3x = max(polygonListX[0]) +8
    p3y = p2y
    p4x = p3x
    p4y = p1y
    g_i.line(p1x, p1y, p2x, p2y)
    g_i.line(p2x, p2y, p3x, p3y)
    g_i.line(p3x, p3y, p4x, p4y)
    g_i.gotomesh()

    for i in range(0, len(g_i.Polygons)):
        BBox = getBoundingBox(g_i.Polygons[i])
        #print(BBox)
        if BBox[0] > (p1x-0.01) and BBox[1] > (p2y-0.01) and BBox[2] < (p4x +0.01):
            g_i.Polygons[i].CoarsenessFactor = lcf

g_i.mesh(resf)

##### Start calculation #####
g_i.gotostages()
g_i.calculate()

##### Save Project #####

g_i.save()

##### Start loop for Result generation#####
#Displacement lists
ux = []
uy = []

g_i.view(g_i.Phases[-1])
s_o, g_o = new_server('localhost', Plaxis_Output_Port, password= Plaxis_Output_Password)

while True:

```

```

print("For which Phase do you want to generate a displacement-file? (enter e.g. 'Phase_12')")
print("Enter 'none' if finished")
phaseName = input()
print("")
if phaseName == "none" or phaseName == "None" or phaseName == "n" or phaseName == "N":
    break
else:
    #Check if entered Phase exists
    correctPhase = False
    for i in range (0, len(g_o.Phases)):
        if phaseName == g_i.Phases[i].Name:
            correctPhase = True

    if correctPhase == True:

        ux = []
        uy = []
        for m in range (0, (len(surfacePointsX))):

            exec(getSingleResultXCommand(phaseName, surfacePointsX[m], surfacePointsY[m]))
            exec(getSingleResultYCommand(phaseName, surfacePointsX[m], surfacePointsY[m]))

        displacedX = []
        displacedY = []

        for m in range (0, len(surfacePointsX)):
            displacedX.append(surfacePointsX[m] + ux[m])
            displacedY.append(surfacePointsY[m] + uy[m])

        exec("phaseID = str(g_o."+phaseName+".Identification)")

        exec("endTime = str(g_o."+phaseName+".Reached.Time)")
        endTime = str(round(float(endTime),1))

        z = heightCenterPoint

        #Delimiter
        delimiter = "#"

        with open (scriptPath + "\ProvIFiles\Pegging_Data_File.txt") as abs_file:
            for line in abs_file:
                row = line.split(delimiter)

                if len(row) > 1:
                    try:
                        axis_station = float(row[1].replace(",","."))
                        if axis_station == QPstation:

                            rw_axis = float(row[3].replace(",","."))
                            hw_axis = float(row[4].replace(",","."))

                    except:
                        pass
                    else:
                        break
            for line in abs_file:
                row = line.split(delimiter)

                if len(row) > 1:
                    try:
                        rightside_station = float(row[1].replace(",","."))
                        if rightside_station == QPstation:

                            rw_rightside = float(row[3].replace(",","."))
                            hw_rightside = float(row[4].replace(",","."))

                    except:
                        pass
                    else:
                        break
        abs_file.close

        deltaX = rw_rightside - rw_axis
        deltaY = hw_rightside - hw_axis

        theta = atan(abs(deltaY)/abs(deltaX))

        hwList = []
        rwList = []
        zList = []
        for i in range (0, len(displacedX)):

            if deltaX >= 0:
                if deltaY <= 0:
                    #Q1
                    hwList.append(hw_axis - displacedX[i]*sin(theta))
                    rwList.append(rw_axis + displacedX[i]*cos(theta))
                    zList.append(z + displacedY[i])
                if deltaY > 0:
                    #Q4:
                    hwList.append(hw_axis + displacedX[i]*sin(theta))
                    rwList.append(rw_axis + displacedX[i]*cos(theta))
                    zList.append(z + displacedY[i])
            if deltaX < 0:

```

```

        if deltaY <=0:
            #Q2
            hwList.append(hw_axis - displacedX[i]*sin(theta))
            rwList.append(rw_axis - displacedX[i]*cos(theta))
            zList.append(z + displacedY[i])
        if deltaY > 0:
            #Q3:
            hwList.append(hw_axis + displacedX[i]*sin(theta))
            rwList.append(rw_axis - displacedX[i]*cos(theta))
            zList.append(z + displacedY[i])

##### write to file #####

save_path = scriptPath + "\OutputFiles"
name = "QP"+str(int(QPstation))+"_"+phaseName
completeName = os.path.join(save_path, name+"_"+endTime + ".scr")
file1 = open(completeName, "w+")

#prepare script with leading commands
file1.write("_multiple\n")
file1.write("_point\n")

for i in range (0, len(hwList)):
    helpStringX = str(round(rwList[i],6))
    helpStringY = str(round(hwList[i],6))
    helpStringZ = str(round(zList[i],6))
    file1.write(helpStringX + "," + helpStringY + "," + helpStringZ + "\n")

file1.close()

##### Write to CSV-file #####
save_path = scriptPath + "\OutputFiles"
name = "QP"+str(int(QPstation))+"_"+phaseName + "_"+endTime + "days"
completeName = os.path.join(save_path, name+".txt")

file2 = open(completeName, "w+")
for i in range (0, len(hwList)):
    helpStringX = str(round(rwList[i],6))
    helpStringY = str(round(hwList[i],6))
    helpStringZ = str(round(zList[i],6))
    file2.write(str(i)+"#"+helpStringX + "#" + helpStringY + "#" + helpStringZ + "\n")

file2.close()

else:
    print("Please enter a valid Phase Name.")

```