“Probabilistic identification of soil stratigraphy using CPT data”

Additional Thesis (10 ECTs)

MSc Guido de Zeeuw
5188083

December 10th, 2021

Assessors
Dr. Divya Varkey
Dr. ir. Bram van den Eijnden
Prof. Dr. Michael Hicks

MSc programme
Geo-Engineering section
Department of Geoscience & Engineering

Table of Contents

Abstract **3**

List of Symbols **4**

Chapter 1 – Introduction **5**

1.1 General Background *5*
1.1.1 Cone Penetration Testing 5
1.1.2 Limitations of Deterministic Approach 6
1.1.3 Bayesian Approach 6

1.2 Scope of This Study *7*

Chapter 2 – Theoretical Framework **8**

2.1 Brief Overview *8*

2.2 Calculating the most probable soil layer thicknesses for model class MN *9*
2.2.1 The Bayesian Approach: short overview 9
2.2.2 Description of Equations 9
2.2.3 Input Parameters 10
2.2.4 Objective Function 11
2.2.5 Output 12

2.3 Obtaining the Most Probable Model Class *12*
2.3.1 Description of equations 12
2.3.2 Output 13

2.4 Pseudocode and procedure description *13*

Chapter 3 – Code Implementation **15**

3.1 Code requirements *15*
3.1.1 Input parameters 15
3.1.2 package installments 16

3.2 Detailed code description: Read CPT Data *16*
3.2.1 The .gef file 16
3.2.2 Obtain matrix from .gef file 17
3.2.3 Generalize matrix 17

3.3 Detailed code description: Calculate probability per datapoint *18*

3.4 Detailed code description: Obtain the most probable thicknesses per model class *19*
3.4.1 Brief overview 19
3.4.2 Initial guesses 21
3.4.3 Optimizing 23

3.5 Detailed code description: Obtain the most probable model class *24*

Chapter 4 – Case Study **25**

4.1 The most probable soil layer thicknesses *25*

4.2 The most probable model class *26*

4.3 Discussion *26*
4.3.1 Comparison with deterministic approach 26
4.3.1 Limitations 27
4.3.2 Optimization 29

Chapter 5 – Conclusion **30**

References **31**

Appendices: Python code **33**

Appendix A: Main.py *33*

Appendix B: Read_cpt.py *35*

Appendix C: Robertson.py *40*

Appendix D: Posterior.py *44*

Abstract

The deterministic approach for interpreting CPT soil profiles poses the serious limitation of not taking data uncertainty into account. Therefore, a Bayesian model was developed by Wang et al. (2013) that, for a given CPT profile, determines the most probable number of soil layers and most probable soil layer thicknesses by simulating and comparing multiple ‘model classes’ with different complexities. In this study, this proposed model is implemented into the Python coding environment after which the functionality is verified by conducting a case study on a $23\ m$ CPT profile from the Groningen area (NE Netherlands). For the given CPT profile, the model distinguishes 6 separate soil layers from which the position and thickness are in agreement with the deterministic analysis and the available borehole data. However, the case study suggests that the model fails to correctly identify the most probable soil types for CPT measurements within the vicinity of the edges of the Robertson chart. This is most-likely related to a “cut-off”-effect of the joint Gaussian distribution describing the uncertainty of a single datapoint. A subsequent study on the integration of the statistical parameters within the model is therefore required. Additionally, the code includes several optimizing strategies, but remains time consuming for very complex model classes. Further optimization is suggested to achieve greater model precision and efficiency.

List of Symbols

Symbol	Unit	Description
General		
q_c	$[MPa]$	Cone resistance
q_t	$[MPa]$	Corrected cone resistance
f_s	$[MPa]$	Sleeve friction
σ_v	$[MPa]$	Total effective stress
$\sigma_v{}'$	$[MPa]$	Effective stress
nQ_t	$[-]$	Normalized cone resistance
nF_r	$[-]$	Normalized friction ratio
Bayesian Framework		
M_N	$[-]$	Model class
N_{max}	$[-]$	Number of model classes considered
ξ_{ni}	$[-]$	Set of $\ln(nQ_t)$ and $ln(nF_r)$ data for the ith datapoint in the nth soil layer
$\underline{\xi}_n$	$[-]$	Set of $\ln(nQ_t)$ and $ln(nF_r)$ data for nth soil layer
$\underline{\xi} = \left[\underline{\xi}_1, \underline{\xi}_2, \dots, \underline{\xi}_N\right]$	$[-]$	Set of $\ln(nQ_t)$ and $ln(nF_r)$ data for a given CPT profile
$P(\underline{\xi}_n \mid N)$	$[-]$	Probability of $\underline{\xi}_n$ given the model class N = the probability that all data points in the nth layer belong to the same soil type.
$P_{ST_J}(\underline{\xi}_n \mid N)$	$[-]$	Probability that all data points in the nth soil layer belong to the same soil type J
$P_{ST_J}(\underline{\xi}_{n,i} \mid N)$	$[-]$	Probability that datapoint i in the nth soil layer belongs to soil type J
σ_{nF_r}	$[-]$	Standard deviation of ln (nF_r)
σ_{nQ_t}	$[-]$	Standard deviation of ln (nF_r)
$\underline{\Omega}_N = [\sigma_1, \sigma_2, \dots, \sigma_N]$	$[-]$	Set of model parameters where $\sigma_n = [\sigma_{nF_r}, \sigma_{nQ_t}]$ for the nth soil layer
Posterior distribution		
$P(\underline{\Omega}_N \mid \underline{\xi}, N)$	$[-]$	Posterior distribution
K_N	$[-]$	Normalizing constant
$P(\xi \mid \underline{\Omega}_N, N)$	$[-]$	Likelihood function, probability of observing the site observation data given$\underline{\Omega}_N$ and N
$P(\underline{\Omega}_N \mid N)$	$[-]$	Prior distribution, which reflects knowledge about $\underline{\Omega}_N$ in the absence of data.
h_N^*	$[m]$	Most probable soil layer thickness for the nth soil layer
Conditional probability		
$P(M_N \mid \underline{\xi})$	$[-]$	Conditional probability, probability of model class M_N given a set of CPT data
$P(\underline{\xi} \mid M_N)$	$[-]$	Conditional probability, probability of $\underline{\xi}$ given the model class M_N
$P(\underline{\xi} \mid M_N, \underline{h}_N^*)$	$[-]$	Conditional probability, probability of $\underline{\xi}$ given the model class M_N and most probable soil layer thicknesses $\underline{h}_N^*$
H_t	$[-]$	Total thickness of the soil profile

Chapter 1 – Introduction

1.1 General Background

A geotechnical study mostly involves simplifying the ground structure and generalizing soil properties to an extent that ground models can be applied to provide results with the least amount of data required. One of the fastest methods to do so is a deterministic approach which allows for such simplification adopting a single value for the input parameters (Lacasse & Nadim, 1998). Although praised for its efficiency, this method is often limited when dealing with problems that contain parameter heterogeneities. In geotechnical studies this is a major downside, as soil properties (e.g. layer thickness, grain size, unit weight, permeability, etc.) are characterized by large spatial variations. Studies in the Netherlands tried to generalize such variations between different projects but conclude that they are often site-specific, requiring information on the local subsurface (Berbee & Fennis, 2019). Therefore, a stochastic approach is introduced to include parameter heterogeneity based on a mean, standard deviation and spatial variability (Fenton, 1999; Hicks, 2005). In a Random Finite Element Model (RFEM), this approach leads to a unique random field that is different between each realization. By adopting many realizations, one may obtain the most conservative outcome (e.g. the realization with the least favorable soil conditions; Luo & Bathurst, 2019).

1.1.1 Cone Penetration Testing

The dominant source of data needed for a stochastic approach, or admittedly for all geotechnical site characterization, is data obtained from the widely used cone penetration tests (CPT's). The application of CPT's has various advantages, being a fast and economical method with a well-developed theoretical background (Robertson, 2010); empirical relations have been developed to correlate CPT data to soil properties (e.g. unit weight, grain size distributions, strength parameters and hydraulic conductivities; Robertson & Cabal, 2010; Tillman et al., 2007). Figure 1.1 illustrates how CPT data (normalized cone tip resistance nQ_t and normalized friction ratio nF_r) relates to different soil behavior types (SBT). Both axes are normalized following Eq. [1] and [2] as Robertson (2009) showed that this provides a more reliable identification of SBT compared to non-normalized charts:

$$nQ_t = \frac{q_t - \sigma_v}{\sigma_v'} \qquad [1]$$

$$nF_r = \frac{f_s}{q_t - \sigma_v} \cdot 100\% \qquad [2]$$

Where q_t is the corrected cone resistance (in MPa) and f_s is the sleeve friction (in MPa). σ_v and σ_v' are the total and effective stresses.

Using this graph, each datapoint on a continuous CPT profile (general distance between measurements is 2 cm) is then classified as one of the nine possible soil types in the Robertson chart; i.e. the area in which a single measurement is located. For a given CPT profile, this allows for a deterministic soil strata classification where different datapoints with similar data values are grouped together, based on engineering judgement (Fig. 1.2).

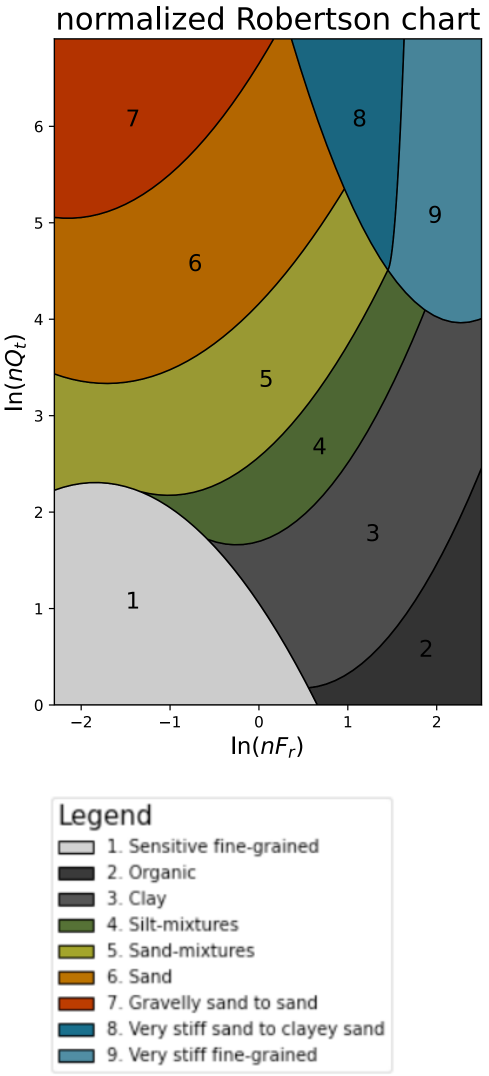


***Figure 1.1**: Robertson Chart (Robertson, 2010)*

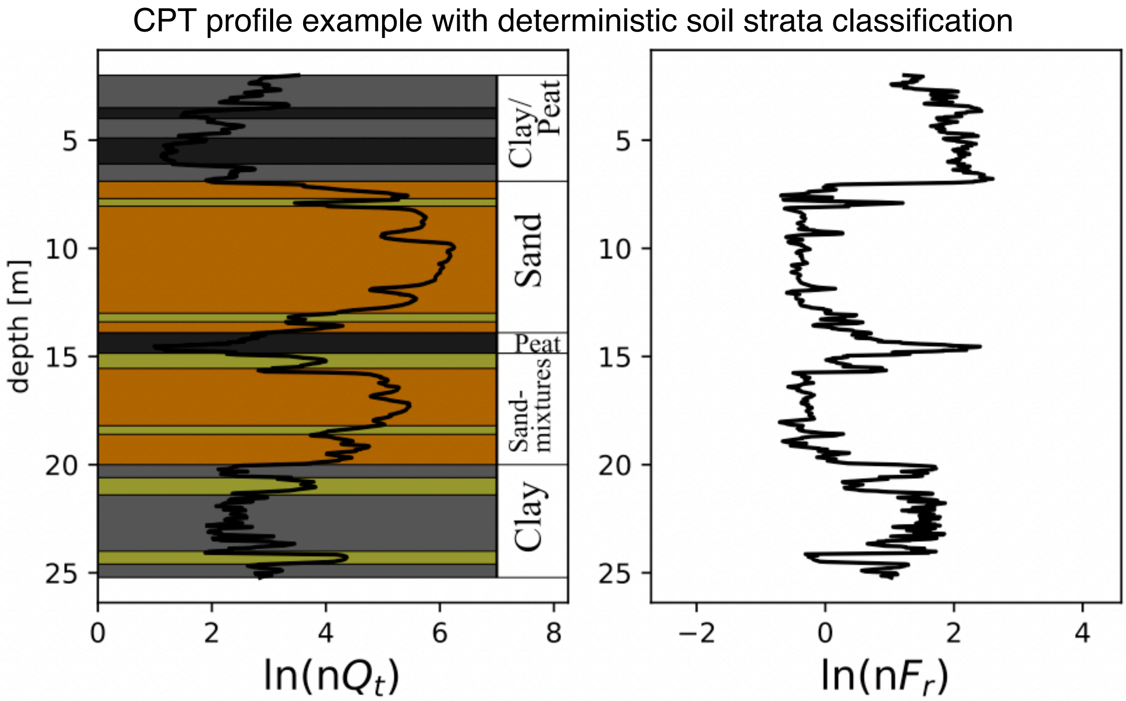


Figure 1.2: *CPT profile (DINOloket sample: CPT000000097976) with a deterministic soil classification indicating the presence of 20 different layers. Colors represent similar soil types as in Figure 1.1.*

1.1.2 Limitations of Deterministic Approach

While including engineering judgement in the analysis might be considered an advantage, it also poses a serious limitation. Different engineers might have different engineering judgement, being the result of experience, which may lead to different identifications of soil strata for similar CPT profiles. Whereas the deterministic approach in Figure 1.2 suggests the presence of 20 different soil layers, a differently experienced engineer might suggest less, or even more, subdivisions. In addition to this, classifying this many soil layers is often not practical for a geotechnical study.

Several authors ascribed this problem to the inexplicit use of data uncertainty within the CPT measurements (Fenton 1999; Wang et al. 2010). Consider, for example, a single measurement point of a CPT profile. In many studies (e.g. Vessia et al., 2020; Kurup & Griffin, 2006; Reale et al., 2018), such measurements are interpreted as deterministic, meaning that the SBT at the depth of that datapoint is fully based on its measured q_t and f_s. However, instead of this measurement being considered deterministic, one can assume that it holds some uncertainty. To illustrate, adopting a uniform distribution (for simplicity), such uncertainty would lead to Figure 1.3, where the measured datapoint may describe different soil types each with its own probability ($P_{soiltype,4} = 0.04$, $P_{soiltype,5} = 0.48$, $P_{soiltype,6} = 0.48$). Including this knowledge would then possibly lead to different soil strata classifications.

1.1.3 Bayesian Approach

The problem posed in the previous section is discussed in detail by Wang et al. (2013). They suggest a Bayesian approach to interpret CPT profiles including these underlying uncertainties. By including prior knowledge, a method is developed to calculate the most probable number of soil layers and soil layer thicknesses. The prior knowledge included in the example by Wang et al. (2013) is limited to specifying a maximum possible number of soil layers and a prior distribution.

Wang et al. (2013) state that the proposed method is consistent with the available borehole data in their region of interest (NGES sit of Texas) but note that the approach requires more testing at other sites with different case studies. In several publications that followed (e.g. Cao & Wang, 2013; Wang et al., 2014; Cao et al., 2019) they further tested the approach, adding friction angles to the model and doing an optimization analysis. For the latter, they found that the measurement interval used for the model calculation is of great importance (Wang et al., 2014).

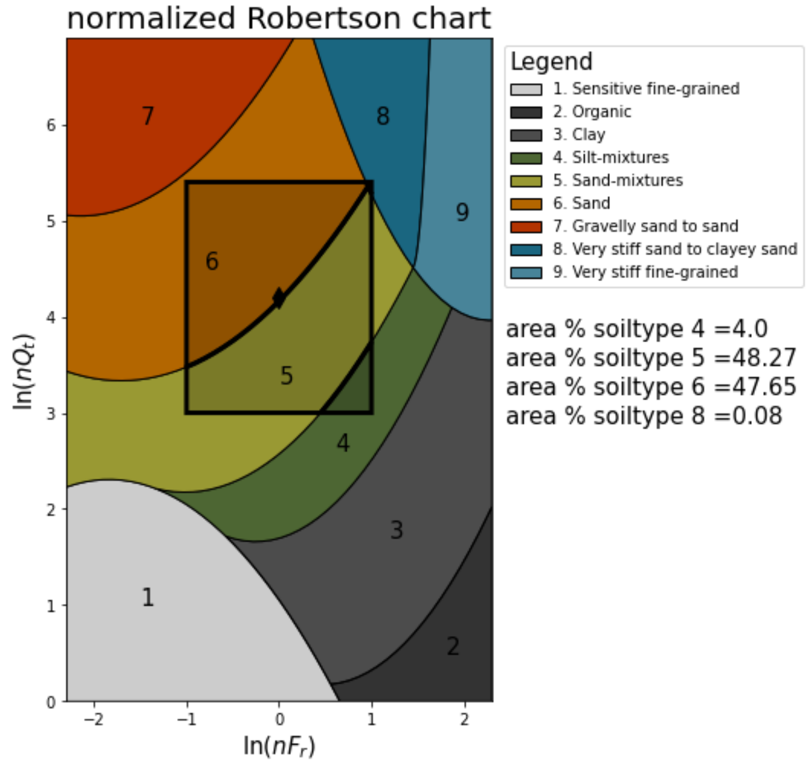


Figure 1.3*: illustrative example of how the uncertainty of a CPT measurement relates to the interpretation of that datapoint*

1.2 Scope of This Study

In this report, the basic model proposed by Wang et al. (2013) is implemented into code software. This study functions as a detailed walkthrough of the code, discussing its theoretical background and the code implementation. The functionality of the code is illustrated via a case study, where it is used to classify soil layers for a 25 m deep soil profile situated to the northeast of the Groningen village (NE Netherlands; Fig. 1.2). This profile is of interest due to its large variations of soil types and number of soil layers, based on a deterministic analysis (Section 1.1.1 Cone Penetration Testing); in total, 20 layers can be distinguished, varying between clays, peats and sands, with large thickness variations from $< 0.5\,m$ to $> 5\,m$. Note, that by doing this case study, this report automatically functions as a revision of the model proposed by Wang et al. (2013).

This report may function as a starting point for researchers that wish to learn about the Bayesian approach as a mean to interpret CPT profiles and to those who want to further improve such a model without the inconvenience of implementing the basic code themselves.

Chapter 2 – Theoretical Framework

2.1 Brief Overview

With the Bayesian approach, a CPT profile is interpreted by calculating the most probable number of soil layers and the most probable layer thicknesses for that particular number of soil layers, based on the uncertainties of CPT measurements. To do so, the approach is subdivided into two parts.

(1) Calculate most probable soil layer thicknesses: First, consider N_{max} different 'model classes' ($\overline{M}$), where N_{max} is the maximum possible number of soil layers considered. For each model class, a different total number of soil layers is taken as such that $\overline{M} = [M_1, M_2, \dots, M_N, M_{N_{max}}]$, where M_N is the Nth model class; coincidingly, N also equals the total number of soil layers considered within that model class. Then, for each model class (each considering a different total number of soil layers) the most probable soil layer thicknesses ($\overline{h_N^*}$) are calculated, $\overline{h_N^*} = [h_1^*, h_2^*, \dots, h_n^*, h_N^*]$, where h_n^* is the most probable thickness of the nth layer of that model class. An example is given in Figure 2.1, where the most probable layer thicknesses are determined for 5 different model classes; note, that Figure 2.1 functions only as an illustrative example for which no real calculations are made. The example indicates that the model becomes more advanced for higher model classes, as more layers are considered.

(2) Select most probable model class: Secondly, all model classes are compared by calculating a conditional probability, $P(M_n|\underline{\xi})$, with $\underline{\xi}$ being a set of $\ln(nF_r)$ and $\ln(nQ_t)$ data of a given CPT profile. In descriptive terms, $P(M_n|\underline{\xi})$ is the probability of model class N given this set of CPT data. The most probable model class (M_N^*) is then selected as the one with the highest value of $P(M_n|\underline{\xi})$, thus obtaining the most probable soil stratigraphy. For the example in Figure 2.1, this would lead to (e.g.) M_4 being the most probable model class, thus resulting in soil strata classification given by that model class.

The next 2 chapters provide a more detailed description on the equations needed for both parts. In 2.4 Pseudocode and procedure description, a pseudocode and an overview of the implementation steps for the approach is given.

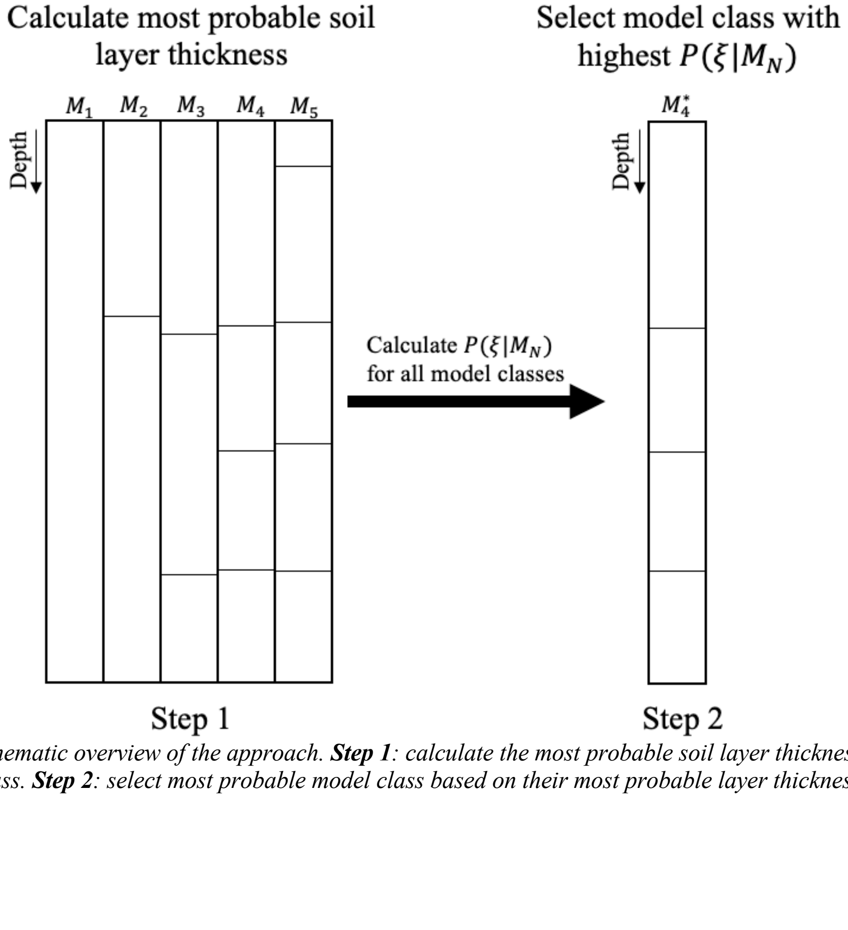

Calculate most probable soil layer thickness
Select model class with highest $P(\xi|M_N)$
M_1 M_2 M_3 M_4 M_5
M_4^*
Depth
Depth
Calculate $P(\xi|M_N)$ for all model classes
Step 1
Step 2

***Figure 2.1**: schematic overview of the approach. **Step 1**: calculate the most probable soil layer thicknesses per model class. **Step 2**: select most probable model class based on their most probable layer thickness.*

2.2 Calculating the most probable soil layer thicknesses for model class M_N

2.2.1 The Bayesian Approach: short overview

In the Bayesian approach, prior knowledge (Θ_P) and the observed data ($\underline{X}$) are used to describe a posterior distribution ($P(\Theta|\underline{X})$; Wang et al., 2016):

$$P(\Theta|\underline{X}) = \frac{P(\underline{X}|\Theta)P(\Theta)}{P(\underline{X})} \qquad [3]$$

Where:

1. $P(\underline{X}|\Theta)$ is the likelihood function, that is, the probability of observing the measured data $\underline{X}$ given the model parameters Θ.
2. $P(\Theta)$ is the prior distribution, that is, the prior knowledge on Θ in the absence of data.
3. $P(\underline{X})$ is a normalizing constant to assure that the sum of all possibilities for Eq. [3] is equal to 1.

Eq. [3] suggests that by multiplying the prior knowledge with newly observed information (the likelihood function), an updated distribution (posterior) is obtained taking this new information into account (Fig. 2.2). This is a large benefit of the Bayesian approach, as new observation data can be used to further improve the model. In the next chapter, Eq. [3] is rewritten for a set of $\ln(nF_r)$ and $\ln(nQ_t)$ data given the underlying uncertainty (σ_{Fr} and σ_{Qt}).

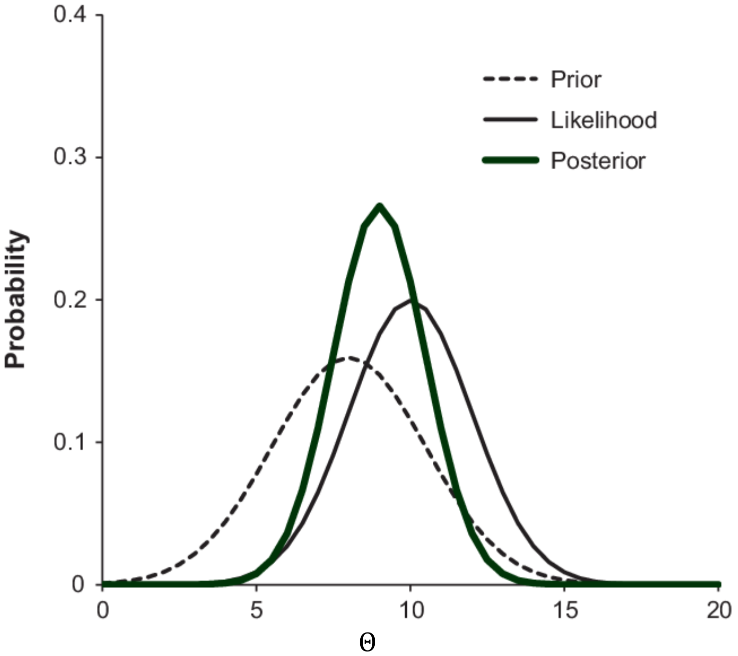


***Figure 2.2**: schematic illustration of the relation between the prior, likelihood and posterior distribution.*

2.2.2 Description of Equations

Rather than writing down the equations for all model classes, let us consider one model class, M_N, with N number of soil layers. According to Wang et al. (2013), the most probable soil layer thicknesses for M_N are calculated by maximizing a joint posterior distribution. Similar to Eq. [3], this is written as:

$$P\left(\underline{\Omega}_N\middle|\underline{\xi}, N\right) = K_N P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right) P(\underline{\Omega}_N|N) \qquad [4]$$

Where $K_N \left(= \frac{1}{P\left(\underline{\xi}\middle|N\right)}\right)$ is a normalizing constant, $P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)$ is the likelihood function and $P(\underline{\Omega}_N|N)$ is the prior distribution of a set model parameters $\Omega_{\mathrm{N}} = \left[\underline{\sigma_1}, \underline{\sigma_2}, \ldots, \underline{\sigma_n}, \underline{\sigma_N}\right]$. The latter is calculated with Eq. [5]:

$$P(\underline{\Omega}_N|N) = \prod_{n=1}^{N} P(\underline{\sigma_n}|N) \qquad [5]$$

Where $P(\underline{\sigma_n}|N)$ is the prior distribution of the model parameters $\underline{\sigma_n} = [\sigma_{Fr,n}, \sigma_{Qt,n}]$, which considers the uncertainty of Fr and Qt for the nth layer. This term is further expressed by Wang et al. (2013) as, Eq. [6]:

$$P\left(\underline{\sigma_n}\middle|N\right) = \left\{\frac{1}{\sigma_{Fr,max} - \sigma_{Fr,min}} \frac{1}{\sigma_{Qt,max} - \sigma_{Qt,min}}\right\} \qquad [6]$$

$\sigma_{Fr,max}$, $\sigma_{Fr,min}$, $\sigma_{Qt,max}$ and $\sigma_{Qt,min}$ are the upper and lower bounds of $\sigma_{Fr,n}$ and $\sigma_{Qt,n}$, leading to $P\left(\underline{\sigma_n}\middle|N\right) = \left\{\left(\frac{1}{\sigma_{Fr}}\right)\left(\frac{1}{\sigma_{Qt}}\right)\right\}$; to the interested reader, a full derivation can be found in Wang et al. (2013, pp. 770-771).

The posterior distribution (Eq. [4]) is a function of the soil layer thicknesses where each soil thickness configuration results in a different value for the posterior distribution. Therefore, *the solution lies in finding the soil thickness configuration leading to the maximum value of the posterior distribution.* Note, however, that both K_N and $P(\underline{\Omega}_N|N)$ in Eq. [4] are constant for each model class. As the approach merely requires maximizing the posterior, it holds that maximizing for $P\left(\underline{\xi}\middle|\underline{\Omega}, N\right)$, the likelihood function, yields to the same most probable layer thicknesses for a given model class.

The likelihood function is calculated with Eq. [7] (Wang et al., 2013):

$$P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right) = \prod_{n=1}^{N} P(\underline{\xi_n}|\underline{\sigma_n}, N) \qquad [7]$$

Where $P\left(\underline{\xi_n}\middle|\underline{\sigma_n}, N\right)$ is calculated for each layer in the model class. $P\left(\underline{\xi_n}\middle|\underline{\sigma_n}, N\right)$ equals to the sum of the probabilities that all datapoints in the nth layer belong to either of the 9 soil types specified on the Robertson Chart, Eq. [8]:

$$P\left(\underline{\xi_n}\middle|\underline{\sigma_n}, N\right) = \sum_{J=1}^{9} P_{ST_J}\left(\underline{\xi_n}\middle|N\right) \qquad n = 1,2,\ldots,N \qquad [8]$$

The probability that all datapoints in the nth layer belong to soil type J is given by Eq. [9]:

$$P_{ST_J}\left(\underline{\xi_n}\middle|N\right) = \prod_{i=1}^{k_n} P_{ST_J}\left(\underline{\xi_{n,i}}\middle|N\right) \qquad J = 1,2,\ldots,9 \qquad [9]$$

Where k_n is the total number of CPT datapoints in the nth soil layer and $P_{ST_J}\left(\underline{\xi_{n,i}}\middle|N\right)$ is the probability that the ith CPT datapoint in the nth soil layer belongs to soil type J. **From this, it becomes evident that the likelihood function, Eq. [7], describes the probability that each of the layers in the model class belongs to a single soil type, given a soil thickness configuration. By maximizing this probability, one obtains the most optimal thickness configuration for that model class.**

2.2.3 Input Parameters

The likelihood function requires two sets of input data: (1) a random set of soil layer thicknesses for which the probabilities are calculated and (2) the probability that datapoint i belongs to soil type J. As implied in the introduction, this probability is described by a probability density function (PDF) around the mean, that is, the measured F_r and Q_t values for the CPT datapoint i. The example given in the introduction adopts a uniform distribution, whereas the actual PDF is more likely to resemble a joint Gaussian distribution based on standard deviations for both F_r and Q_t (Fig. 2.3). The probability that datapoint i belongs to soil type J is then calculated by integrating the Gaussian curves over the areas on the Robertson chart describing the different soil types.

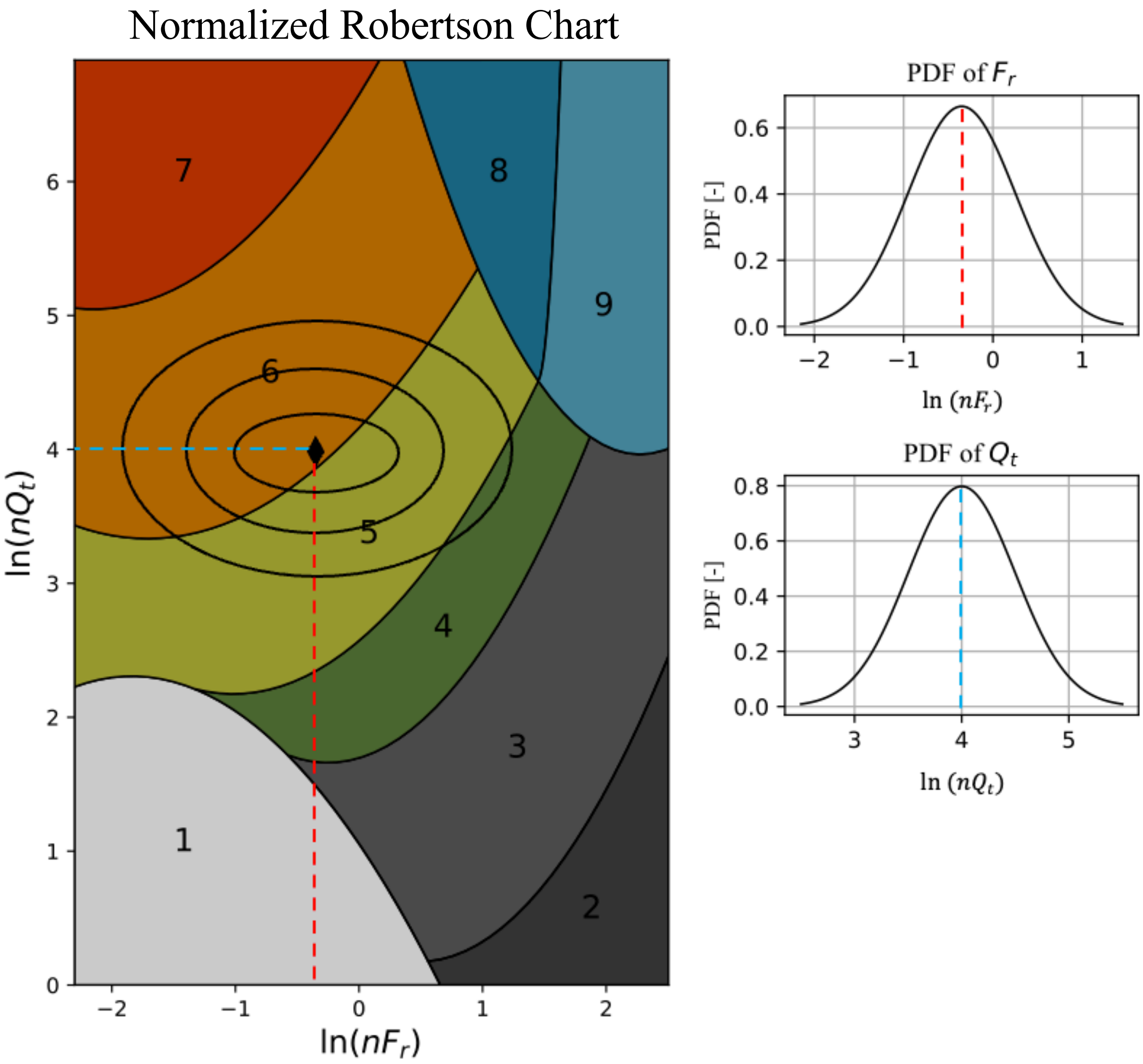


***Figure 2.3**: Example of a joint Gaussian distribution for datapoint i, for* $\sigma_{F_r} = 0.6, \sigma_{Q_t} = 0.5$.

2.2.4 Objective Function

The most probable thickness boundaries are found by maximizing the posterior PDF in Eq. [3] or the likelihood function in Eq. [6]. This method describes the asymptotic technique that involves approximating the posterior PDF as a Gaussian PDF, with a mean equal to the most probable value of the posterior PDF (Bleistein and Handelsman, 1986). Rather than maximizing the likelihood function, $P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)$, it is more convenient to define an objective function, Eq. [10], and to minimize for that:

$$f_{obj} = -\ln\left(P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)\right) \qquad [10]$$

This is done as the approach deals with very small probabilities, sometimes too small to grasp by computer software. For this, imagine a $15\ m$ thick soil layer with $\left(\frac{15}{0.02} =\right) 750$ datapoints, all having a probability of 0.2 that they belong to soil type J. Eq. [10], for that soil type, then leads to a probability of 0.2^{750} which, by a computer, would be approximated as 0.

2.2.5 Output

By finding the soil layer thickness configuration that leads to the lowest value of f_{obj}, the most probable soil layer thickness configuration for a model class is found. In 2.1 Brief Overview this is illustrated by Figure 2.1, where the most probable soil layer thicknesses are calculated for 5 different model classes.

2.3 Obtaining the Most Probable Model Class

2.3.1 Description of equations

Once the most probable soil layer thicknesses are acquired, a conditional probability is defined for each model class, Eq. [11] (Wang et al., 2013):

$$P\left(M_N\middle|\underline{\xi}\right)=\frac{P\left(\underline{\xi}\middle|M_N\right)P(M_N)}{P\left(\underline{\xi}\right)}\qquad N=1,2,\dots,N_{max}\qquad [11]$$

Where $P\left(\underline{\xi}\middle|M_N\right)$ is the conditional probability of $\underline{\xi}$ given the model class M_N, $P(M_N)$ reflects the prior knowledge on the number of soil layers and $P\left(\underline{\xi}\right)$ is the PDF of $\underline{\xi}$. **The most probable model class (M_N^*) is selected as the one with the highest value of $P(M_n|\underline{\xi})$.**

If there is no prior knowledge on the total number of soil layers, $P(M_N)$ can be assumed to be equal for all model classes, approximated by $\frac{1}{N_{max}}$. This, in combination with $P\left(\underline{\xi}\right)$ being independent of the model class, suggests that maximizing the conditional probability ($P\left(M_N\middle|\underline{\xi}\right)$) only requires maximizing for $P\left(\underline{\xi}\middle|M_N\right)$. This term is further approximated by Eq. [12]:

$$P\left(\underline{\xi}\middle|M_N\right)\approx P\left(\underline{\xi}\middle|M_N,h_N^*\right)P\left(\underline{h}_N^*\right)\qquad N=1,2,\dots,N_{max}\qquad [12]$$

Here, $P\left(\underline{\xi}\middle|M_N,\underline{h}_N^*\right)$ is the conditional probability of $\underline{\xi}$ given the model class (M_N) and its most probable soil layer thicknesses h_N^*. This term is calculated with:

$$P\left(\underline{\xi}\middle|M_N,\underline{h}_k^*\right)\approx\int P\left(\underline{\xi}\middle|\underline{\Omega}_N,M_N\right)P(\underline{\Omega}_N|M_N)d\underline{\Omega}_N\qquad [13]$$

Where $P\left(\underline{\xi}\middle|\underline{\Omega}_N,M_N\right)$ and $P(\underline{\Omega}_N|M_N)$ are the likelihood function and prior distribution of model class N (in Eq. [4]), respectively. As $\underline{\Omega}_N$ in constant and fixed throughout the analysis, this is further simplified by removing the integral from Eq. [13], leading to

$$P\left(\underline{\xi}\middle|M_N,\underline{h}_N^*\right)\approx P\left(\underline{\xi}\middle|\underline{\Omega}_N,M_N\right)P(\underline{\Omega}_N|M_N)\qquad [14]$$

Note, that both terms on the right side are calculated as described in 2.2.2 Description of Equations (Eq. [5] and [7])

$P\left(\underline{h}_N^*\right)$ in Eq. [12] is the occurrence probability of $\underline{h}_N^*$. Since the sum of all soil layer thicknesses must equal the total thickness (H_t) of the soil profile, there are N-1 independent layer thicknesses for a model class M_N. Under the assumption of a uniform distribution for the layer thicknesses with a constant probability density of $1/H_t$, $P\left(\underline{h}_N^*\right)$ is approximated as (Wang et al., 2013):

$$P\left(\underline{h}_N^*\right)\approx\frac{1}{H_t^{N-1}}\qquad [15]$$

For higher model classes, this term becomes smaller, acting as a penalty against overparameterization. Finally, combining Eq. [14] and [15] leads to the maximizing function:

$$P\left(\underline{\xi}\middle|M_N\right)=\frac{1}{H_t^{N-1}}P\left(\underline{\xi}\middle|\underline{\Omega}_N,M_N\right)P(\underline{\Omega}_N|M_N)\qquad [16]$$

2.3.2 Output

Eq. [16] is computed for each model class, thus leading to N_{max} model classes each having a unique value of $P\left(\underline{\xi}\middle|M_N\right)$. Eq. [16] describes the probability of observing the set of data from the CPT profile, given the model class M_N. Thus, the model class resulting in a maximum value of $P\left(\underline{\xi}\middle|M_N\right)$ is selected as the most probable model class. In 2.1 Brief Overview this is illustrated by Figure 2.1 where from 5 model classes (e.g.) model class 4 (M_4) is selected as the most probable model class, having the largest value for $P\left(\underline{\xi}\middle|M_N\right)$.

2.4 Pseudocode and procedure description

Figure 2.4 provides a pseudocode, describing the general procedure of the approach discussed in the previous chapters. Below is a detailed description of each step including the associated equations.

1. Obtain a set of CPT data and convert them to $\ln(nQ_t)$ and $\ln(nF_r)$ using Eq. [1] and [2]. For this set of data, specify a maximum possible number of soil layers (N_{max}), thus resulting in N_{max} model classes.
2. Define an objective function, $f_{obj} = -\ln\left(P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)\right)$, for model class M_N adopting a prior distribution for the model parameters $\Omega_\mathrm{N} = \left[\underline{\sigma}_1, \underline{\sigma}_2, \dots, \underline{\sigma}_n, \underline{\sigma}_N\right]$. For this, use Eq. [7] and [10].
    a. For each CPT datapoint in the nth soil layer, calculate the probability of that datapoint belonging to each of the 9 soil types of the Robertson chart.
    b. Calculate the probability that all datapoints in the nth soil layer belong to the same soil type, Eq. [9].
    c. Calculate the likelihood function $P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)$, Eq. [7], being the direct input of the objective function.
3. Minimize the objective function for the M_N model class to obtain $\overline{h_N^*}$. This is done by either (1) evaluating every possible soil layer thickness configuration for that model class and select the one resulting in the lowest value of f_{obj} or (2) using the python package 'scipy.optimize' (see 3.4.3 Optimizing).
4. Calculate the conditional probability, $P\left(\underline{\xi}\middle|M_N\right)$, using Eq. [16].
5. Repeat step 2-4 N_{max} times, evaluating the conditional probability for each model class.
6. Select the model class with the highest value of $P\left(\underline{\xi}\middle|M_N\right)$, this is taken as the most probable model class M_N^*.
7. For this model class, select $\overline{h_N^*}$ and determine the most probable soil types.

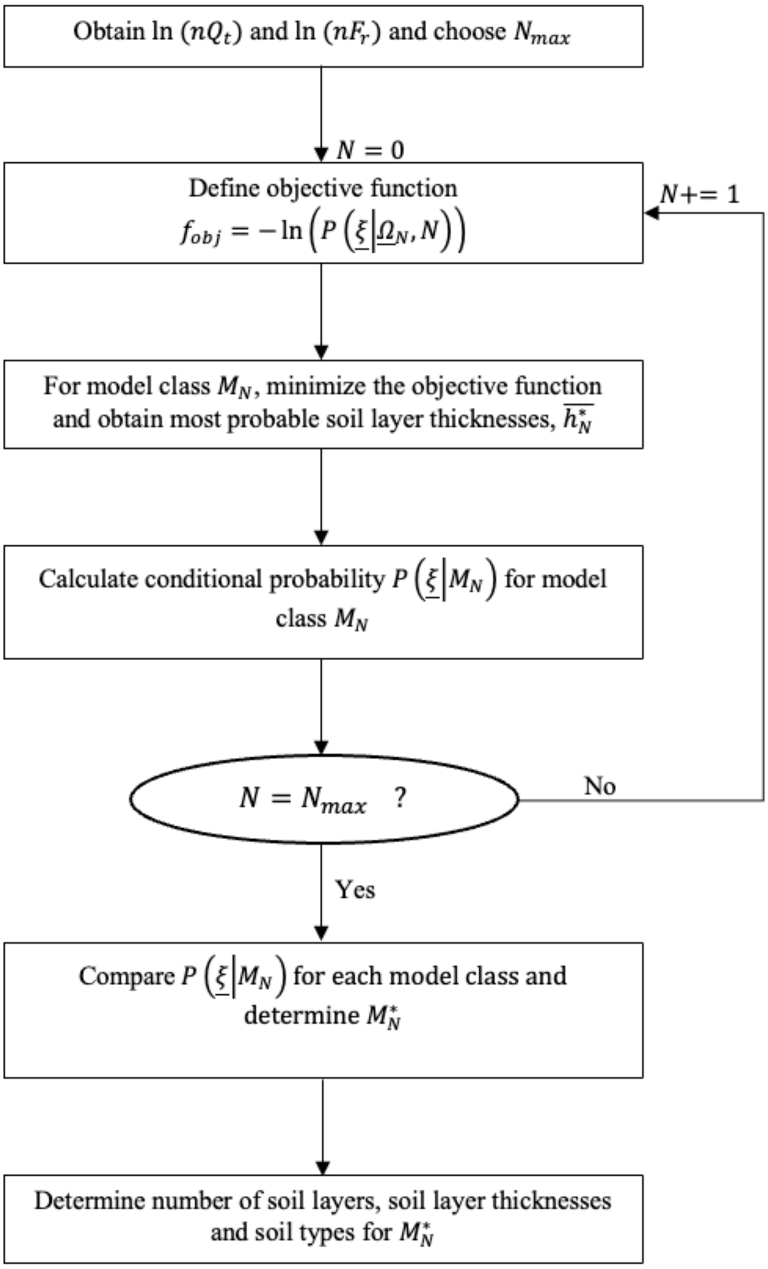

Obtain $\ln(nQ_t)$ and $\ln(nF_r)$ and choose N_{max}
$N = 0$
Define objective function
$f_{obj} = -\ln\left(P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)\right)$
$N+= 1$
For model class M_N, minimize the objective function and obtain most probable soil layer thicknesses, $\overline{h_N^*}$
Calculate conditional probability $P\left(\underline{\xi}\middle|M_N\right)$ for model class M_N
$N = N_{max}$?
No
Yes
Compare $P\left(\underline{\xi}\middle|M_N\right)$ for each model class and determine M_N^*
Determine number of soil layers, soil layer thicknesses and soil types for M_N^*

Figure 2.4*: Pseudocode of the proposed approach*

Chapter 3 – Code Implementation

The method discussed in Chapter 2 – Theoretical Framework is implemented into Python code. The steps are elaborated on as such, that readers who are more comfortable with other programming software (e.g. MATLAB) could easily convert the method to a different coding environment. The full python script is provided in the Appendix. The code goes through the 7 steps described in 2.4 Pseudocode and procedure description subdivided into 4 parts:

1. Read the CPT data (`.gef` file)
2. Calculate the probability that datapoint i belongs to soil type $1, 2, \dots, 9$.
3. Obtain the most probable soil layer thicknesses for model class $1, 2, \dots, M_N, M_{N_{max}}$.
4. Obtain the most probable model class M_N^*.

The code includes 4 different python script files (`.py`):

1. `main.py`: the main file which follows the general workflow described above. The other files (2-4) are imported as packages and the individual functions in these files are called upon when necessary (Appendix A).
2. `read_cpt.py`: file with functions to read and store CPT data in a matrix (Appendix B).
3. `robertson.py`: file with functions to specify the soil areas on the Robertson chart and to calculate the probability that datapoint i belongs to soil type $1, 2, \dots, 9$ (Appendix C).
4. `posterior.py`: file with functions to calculate the most probable soil layer thicknesses per model class and to obtain the most probable model class M_N^* (Appendix D).

3.1 Code requirements

3.1.1 Input parameters

In `main.py`, the input parameters are specified. These include:

- *cpt:* a .gef file of the CPT that needs to be analyzed (see 3.2.1 The .gef file)
- *N max:* the maximum number of soil layers to consider. This value equals the number of model classes for which the model will calculate.
- *min thickness:* minimum thickness of a soil layer to be considered a separate layer.
- *std Fr/std Qt:* the standard deviation of the friction ratio and cone resistance, respectively
- *implement constraints:* determines if the *boundary constraint* (see below) should be considered. Set to “yes” or “no”.
- *boundary constraint:* if needed, specify a matrix with shape = (number of constraints, 3). For each constraint, specify the array [top of depth interval, bottom of depth interval, minimum model class to consider constraint].
  Applications: this constraint is used to force the model to include a soil layer boundary within the specified domain, depending on the model class. **Example:** If the user wants to make sure that a soil boundary between 4-6 m depth is included when 3 or more soil layers are considered (=model class 3 or higher) specify the input as `np.array([4,6,3])`. More constraints can be included following the same structure, e.g.: `np.array([[4,6,3], [6,9,5]])`.
- *Probability iterations:* number of iterations to approximate the joint Gaussian distribution with a Monte Carlo analysis (3.3 Detailed code description: Calculate probability per datapoint). *Note:* a higher value results in a better approximation of the joint Gaussian curves. However, this will also lead to larger computation times.
- *Threshold model class:* last model class that does not need further optimization, see 3.4.3 Optimizing.
- *No.* guesses: the number of best soil thickness configurations that are selected to further optimize the model class, see 3.4.3 Optimizing.
- *Guesses iterations*: number of randomly generated soil layer thickness configurations from which X (=’no. guesses) best configurations are selected, see 3.4.2 Initial guesses and 3.4.3 Optimizing.

3.1.2 package installments

The Python code makes use of three useful packages: shapely, geopandas and descartes. As these packages are greatly dependent on each other, it is required that the user installs the correct versions. To avoid conflicts with interdependencies with pre-installed python packages, it is suggested to create a separate working environment for the code. This is best done within the anaconda environment which can be installed from: https://www.anaconda.com/products/individual.

STEP 1: SETTING UP A NEW ENVIRONMENT

1. Open the anaconda navigator
2. Type: `conda create -n geo_env` (this creates a new environment called “geo_env”)
3. Type: `conda activate geo_env` (this activates the new environment)

STEP 2: DOWNLOAD GEOPANDAS

1. Type: `conda config --env --add channels conda-forge`
2. Type: `conda config --env --set channel_priority strict`
3. Type: `conda install python=3 geopandas`

STEP 3: DOWNLOAD DESCARTES

1. Type: `conda install -c conda-forge Descartes`

Note, that to be able to run the code successfully, the environment needs to be active. The environment can always be activated in the anaconda navigator using `conda activate geo_env.`

3.2 Detailed code description: Read CPT Data

The code described in this part can be found in Appendix B.

3.2.1 The .gef file

CPT data is typically stored in a `.gef` file. A small section of the data in such a file is given below, where the columns are separated with a ‘;’.

```
2.000;0.640;1.997;137.0;3;-1;3;0.021;0.000;3.7;!
2.020;0.590;2.017;138.0;3;-2;3;0.021;-0.001;4.0;!
2.040;0.530;2.037;139.0;3;-2;3;0.021;-0.004;4.4;!
2.060;0.470;2.057;140.0;3;-2;4;0.020;-0.003;4.6;!
2.080;0.440;2.077;141.0;3;-1;3;0.017;-0.002;4.1;!
2.100;0.410;2.097;142.0;3;-2;4;0.015;0.003;3.8;!
```

The definition of each column value may vary between each .gef file but are always given at the top of the file. In this particular case, the following definitions are adopted for the different column values:

```
#COLUMNINFO= 1, m (meter), sondeertrajectlengte, 1
#COLUMNINFO= 2, MPa (megaPascal), conusweerstand, 2
#COLUMNINFO= 3, m (meter), diepte, 11
#COLUMNINFO= 4, s (seconde), verlopen tijd, 12
#COLUMNINFO= 5, ° (graden), helling oost-west, 10
#COLUMNINFO= 6, ° (graden), helling noord-zuid, 9
#COLUMNINFO= 7, ° (graden), hellingresultante, 8
#COLUMNINFO= 8, MPa (megaPascal), plaatselijke wrijving, 3
#COLUMNINFO= 9, MPa (megaPascal), waterspanning u2, 6
#COLUMNINFO= 10, % (procent; MPa/MPa), wrijvingsgetal, 4
```

Sometimes, the cone penetration test fails to capture data at a certain depth, leading to a ‘void’ in the data set. The .gef file therefore specifies the errors if a value is missing:

```
#COLUMNVOID= 1, 999.999
#COLUMNVOID= 2, 999.999
#COLUMNVOID= 3, 999.999
#COLUMNVOID= 4, 99999.9
#COLUMNVOID= 5, 99
#COLUMNVOID= 6, 99
#COLUMNVOID= 7, 99
#COLUMNVOID= 8, 9.999
#COLUMNVOID= 9, 99.999
#COLUMNVOID= 10, 999.9
```

3.2.2 Obtain matrix from .gef file

The `.gef` file is imported using the function ‘`read_cpt`’ in `read.py`. The different columns of the `.gef` file are stored in a matrix and, for each datapoint, used to obtain the following 12 different columns.

```
    #0 = depth
    #1 = cone resistance
    #2 = sleeve friction
    #3 = u2
    #4 = friction ratio
    #5 = I_sbt
    #6 = volumetric weigth soil
    #7 = total vertical stress
    #8 = effective vertical stress assuming ps is at z=0
    #9 = normalized cone stress
    #10 = normalized friction ratio
    #11 = corrected cone resistance
```

The first 5 columns (0-4) follow directly from the `.gef` file. The pore water pressure (`u2`), however, was often found to contain false values that, from an engineering perspective, make limited sense. Therefore, the assumption is made that the pore water pressure increases linearly with depth. Secondly, if a datapoint contains an error, a new value is taken via interpolation from the surrounding datapoints.

The values for the remaining columns (5-11) are based on empirical relations. I_{SBT} is related to the corrected cone resistance (q_t) and the atmospheric pressure (taken at 0.1 MPa):

$$I_{SBT} = \left(3.47 - \log\left(\frac{q_t}{P_{atm}}\right)\right)^2 \qquad [17]$$

The volumetric soil weight is based on an empirical relation given by Robertson (1990), Eq. [18]:

$$W_{soil} = \gamma_w\left(0.27\log(f_r) + 0.36\log\left(\frac{q_t}{P_{atm}}\right)\right) + 1.236 \qquad [18]$$

Where γ_w is the unit weight of water (in $\frac{kN}{m^3}$) and f_r is the friction ratio. From this follows the total vertical stress, being the cumulative sum of the soil weights of the overlying datapoints. Subsequently, the effective stress is calculated from the total stress (σ_v) and the pore water pressure (u):

$$\sigma_v' = \sigma_v - u \qquad [19]$$

Adopting Eq. [1] and [2] then leads to the normalized cone resistance and friction ratio.

3.2.3 Generalize matrix

Although the approach makes use the natural logarithm to deal with very small probabilities, it is found that this solution is not always sufficient, especially dealing with very large datasets (e.g. very large CPT profiles). Therefore, it is necessary to include a data generalization (function ‘`generalize`’ in `read_cpt.py`, Appendix B). Here, a very simple solution is implemented where the dataset is reduced by a factor directly related to the minimum possible thickness (H_{min}) specified by the user where:

$$F_{reduction} = \frac{H_{min}}{0.02} \qquad [20]$$

Where $0.02\ m$ is the distance between measurements used by CPT's. For the case that a minimum possible thickness of $10\ cm$ is adopted, this results in a reduction factor of 5. This means that the generalized dataset is 5 times smaller than the original dataset. The column values for each generalized segment are taken as the average of the values for which it the segment is generalized.

Note, that only the depth, normalized cone resistance and friction ratio are needed for the remaining part of the approach. The generalized matrix therefore contains 12 redefined columns:

```
    #0 = depth generalized
    #1 = log of nQt generalized
    #2 = log of nFr generalized
    #3-11 = for probabilities; for now empty
```

Column 4-12 are empty for now and will be used for data storage in the next part.

3.3 Detailed code description: Calculate probability per datapoint

In 2.2.3 Input Parameters, the probability that datapoint i belongs to each of the 9 soil types ($P_{ST_J}\left(\underline{\xi}_{n,i}\middle|N\right)$ in Eq. [8]) is described as the integration of a joint Gaussian distribution (of F_r and Q_t) over the different areas of the Robertson chart describing the different soil types. Rather than solving for many different integrations per datapoint, which is both complex and time demanding, a Monte Carlo (MC) analysis is conducted (function `MC_probability` in `robertson.py` Appendix C). The joint Gaussian PDF for datapoint i is approximated by generating many points on the Robertson chart, each point being randomly generated based on a mean (the measured F_r and Q_t of datapoint i) and a set of standard deviations (e.g. Fig. 3.1). From this, $P_{ST_J}\left(\underline{\xi}_{n,i}\middle|N\right)$ for datapoint i is approximated as:

$$P_{ST_J}\left(\underline{\xi}_{n,i}\middle|N\right) = \frac{count\ of\ MC\ realizations\ in\ area\ J}{total\ MC\ realizations} \qquad [21]$$

To count the number of points that fall within a certain soil type area J, the boundaries of the areas of the Robertson chart are approximated by eight quadratic functions (Fig. 3.2; table 3.1) and their intersecting points (table 3.2), based on Wang et al. (2013). Using the python package '`shapely`', these coordinates are then used to create polygons. This is done because the `shapely` package has a nice built-in function '`.within`' that evaluates if a point lies within a polygon. Evaluated this for each generated point of the MC analysis will result in the numerator of Eq. [21].

For every datapoint of the CPT profile, this leads to 9 different probabilities (probability of belonging to soil type 1, 2, 3,..., 9), which are stored in column 4-12 of the generalized matrix from 3.2.3 Generalize matrix. Note, that if the MC analysis is applied to many realizations (this number is specified by the user =*Probability iterations*), this may take a significant time to finish.

***Table 3.1**: best-fitted quadratic functions for the boundaries on the Robertson chart (Wang et al., 2013).*

I	$\ln(Q_t) = -0.3707\ln(F_r)^2 - 1.3625\ln(F_R) + 1.0549$
II	$\ln(Q_t) = -0.5586\ln(F_r)^2 - 0.5399\ln(F_R) + 0.3049$
III	$\ln(Q_t) = 0.5405\ln(F_r)^2 + 0.2739\ln(F_R) + 1.6959$
IV	$\ln(Q_t) = 0.3833\ln(F_r)^2 + 0.7805\ln(F_R) + 2.5718$
V	$\ln(Q_t) = 0.2827\ln(F_r)^2 + 0.967\ln(F_R) + 4.1612$
VI	$\ln(Q_t) = 0.3477\ln(F_r)^2 + 1.4933\ln(F_R) + 6.6507$
VII	$\ln(Q_t) = 0.8095\ln(F_r)^2 - 3.6795\ln(F_R) + 8.1444$
VIII	$\ln(Q_t) = 64.909\ln(F_r)^2 - 187.07\ln(F_R) + 139.2901$

***Table 3.2**: Coordinates of the intersection points of the Robertson chart (Wang et al., 2013).*

Intersection point	$\ln(F_r)$	$\ln(Q_t)$	Intersection point	$\ln(F_r)$	$\ln(Q_t)$
A	-2.3026	0	**L**	0.9622	5.3534
B	0.6569	0	**M**	-2.3026	3.4335
C	-2.3026	2.2268	**N**	0.3655	6.9078
D	2.3026	0	**O**	0.1658	6.9078
E	2.3026	2.0234	**P**	-2.3026	5.0557
F	0.5589	0.1776	**Q**	-2.3026	6.9078
G	2.3026	3.9639	**R**	2.3026	6.9078
H	1.8687	4.0953	**S**	1.6334	6.9078
J	1.4505	4.5104	**T**	-0.5773	1.7179
K	-1.3334	2.2126			

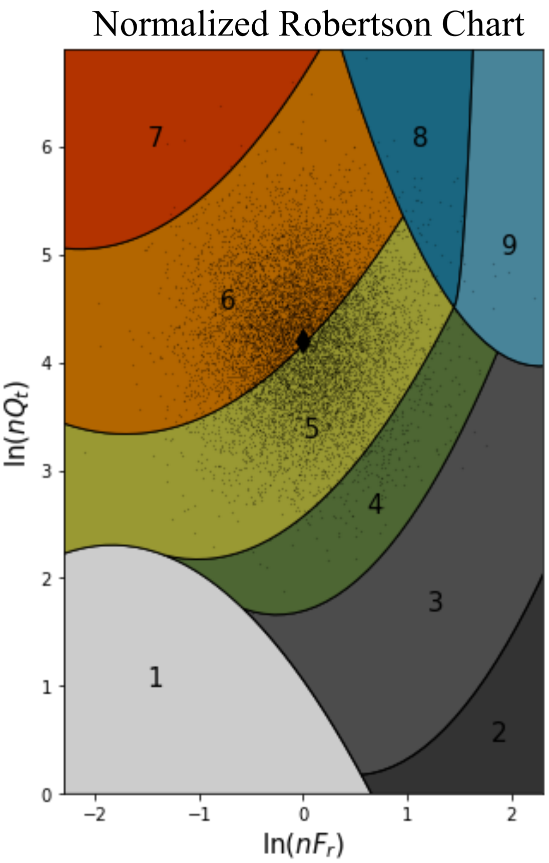


***Figure 3.1**: Monte Carlo analysis that approximates the joint Gaussian PDF for datapoint i.*

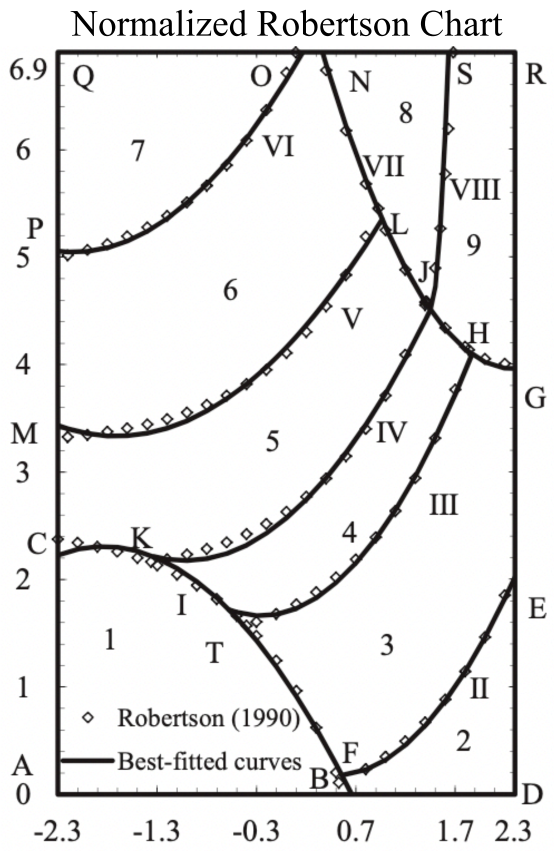


***Figure 3.2**: Overview of the quadratic functions to approximate the boundaries on the Robertson chart (Wang et al., 2013).*

3.4 Detailed code description: Obtain the most probable thicknesses per model class

3.4.1 Brief overview

Here, the code that implements the theory described in 2.2 Calculating the most probable soil layer thicknesses for model class M_N is discussed. The output following from 3.3 Detailed code description: Calculate probability per datapoint is used. That is, a matrix wherein for each datapoint the probabilities are listed that that datapoint belongs to one of the 9 soil types described by the Robertson chart. For each model class, an objective function (Eq. [9]) is constructed and minimized based on these probabilities. As the objective function yields a different solution for each different soil thickness configuration, a straight-forward approach would be to calculate the objective function for many thickness configurations. Then, the most probable thickness configuration is taken as the one configuration that leads to the lowest value for the objective function.

This approach is valid for model classes with a low number of soil layers, but lacks efficiency for model classes with a large number of soil layers. For model classes with a high number of soil layers, significantly more configurations must be evaluated to include all possibilities, thus requiring significantly longer computation times (Fig. 3.3). Therefore, for higher model classes, an optimization approach is suggested.

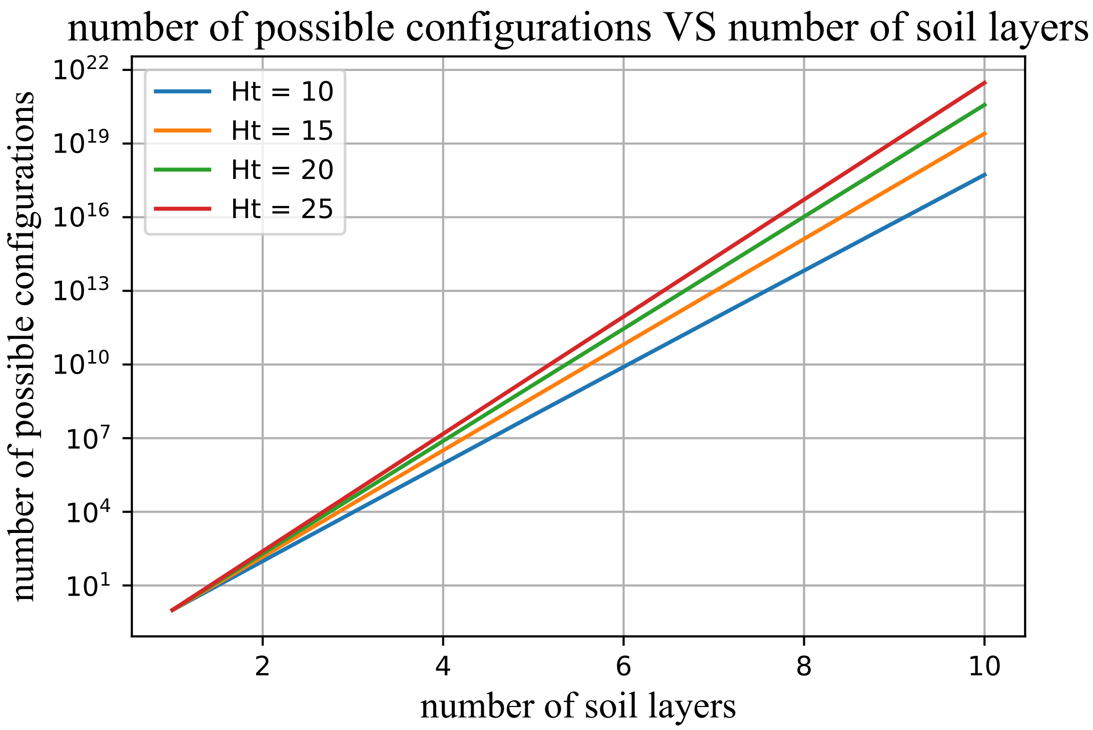


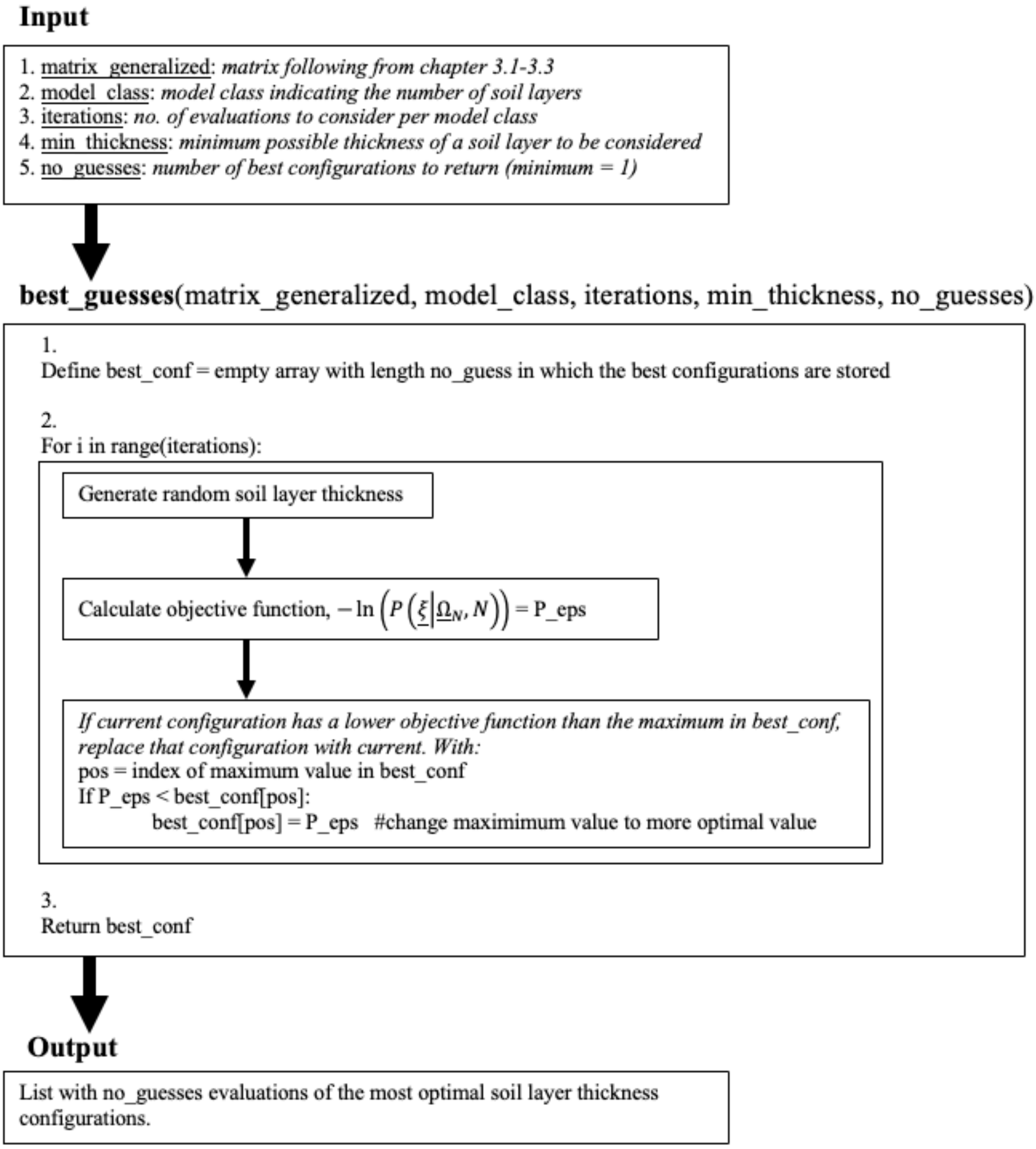
Figure 3.3*: Number of possible soil layer thickness configurations versus the number of soil layers considered. 4 different soil profile thicknesses are included. Note: y-scale is logarithmic.*

In `posterior.py` (Appendix D) this optimization is achieved with the two functions '`best_guesses`' and '`optimizer`'. '`best_guesses`' simulates the likelihood function for a large set of randomly generated layer thickness configurations. The function returns a list "best_conf" where X (number specified by the user) configurations are stored having the lowest values for the objective function. For model classes with a *low number* of soil layers, the objective function is then minimized by selecting the configuration within that list that contains the lowest value for the objective function.

For model classes with a *high number* of soil layers, each individual configuration in the list following from '`best_guesses`' is taken as the input parameter for the function '`optimizer`'. For a given layer thickness configuration this function further minimizes the objective function, using the `scipy.optimize` package. Then, similarly to the model classes containing a low number of soil layers, the configuration leading to the lowest value of the objective function is then taken as the configuration with the most probable soil layer thicknesses.

3.4.2 Initial guesses

In `posterior.py`, the function (`best_guesses`) is defined in which the objective function is evaluated for many different (randomized) soil layer thickness configuration. Below is a schematic pseudocode of the function.



Input

1. matrix_generalized: *matrix following from chapter 3.1-3.3*
2. model_class: *model class indicating the number of soil layers*
3. iterations: *no. of evaluations to consider per model class*
4. min_thickness: *minimum possible thickness of a soil layer to be considered*
5. no_guesses: *number of best configurations to return (minimum = 1)*

best_guesses(matrix_generalized, model_class, iterations, min_thickness, no_guesses)

```
1.
Define best_conf = empty array with length no_guess in which the best configurations are stored

2.
For i in range(iterations):
    Generate random soil layer thickness
    Calculate objective function, $-\ln\left(P\left(\underline{\xi}\middle|\underline{\Omega}_N, N\right)\right)$ = P_eps
    If current configuration has a lower objective function than the maximum in best_conf, replace that configuration with current. With:
    pos = index of maximum value in best_conf
    If P_eps < best_conf[pos]:
            best_conf[pos] = P_eps   #change maximimum value to more optimal value

3.
Return best_conf
```

Output

List with no_guesses evaluations of the most optimal soil layer thickness configurations.

1. Generating a random configuration of soil layer thicknesses:
First, an array of possible boundary positions is generated based on the boundaries of the CPT profile, based on a of $0.1\ m$ spacing. From this array, the positions of the boundaries are randomly generated with the `random.choice()` command (see below, line 8). By doing this iteratively for each boundary, the randomized value is deleted from the array with total possibilities to assure that the other boundaries are not assigned to the same positions. In the python script this is results in:

```
1.
2.     boundaries = np.zeros(model_class+1)  #empty list in which boundaries are stored
3.
4.     boundaries[0] = matrix_generalized[0,0] #first value = top of CPT profile
5.
6.     if model_class>1:
7.         for nn in range(1+len(L_constraints),model_class+1):
8.             boundaries[nn] = random.choice(boundary_possibilities)
9.
10.            #indices of value +- min_thickness in boundary_possibilities
11.            idx = np.array(np.where((boundary_possibilities>boundaries[nn]-min_thickness)
12.                     *(boundary_possibilities<boundaries[nn]+min_thickness))[0])
13.
14.            #delete indices so that next iteration only a correct boundary
15.            boundary_possibilities=np.delete(boundary_possibilities,idx)
```

Note, that the boundaries need only to be generated for model classes that have more than 1 layer. One might also note the variable `len(L_constraints)` in line 2. This term allows to include constraints in the random generation of layer boundaries. If no constraints are specified this value equals to 0. The constraint is a mean to force the position of a single (or more) soil boundary, having the following input layout:

```
1.  np.array([[start,stop,model class],   #constraint 1
2.            [start,stop,model class],   #constraint 2
3.            ...)
```

Where ‘start’ and ‘stop’ specify the range in which (at least) one of the boundaries should be positioned. The ‘model class’ parameter specifies when this constraint should be included, for example: the constraint [6,9,3] makes sure that (at least) one soil boundary is located between 6 and 9 m depth, only if 3 or more soil layers are considered.

If constraints are included in the analysis, the program will automatically provide a CPT profile showing the different constraints (e.g. Fig. 3.3).

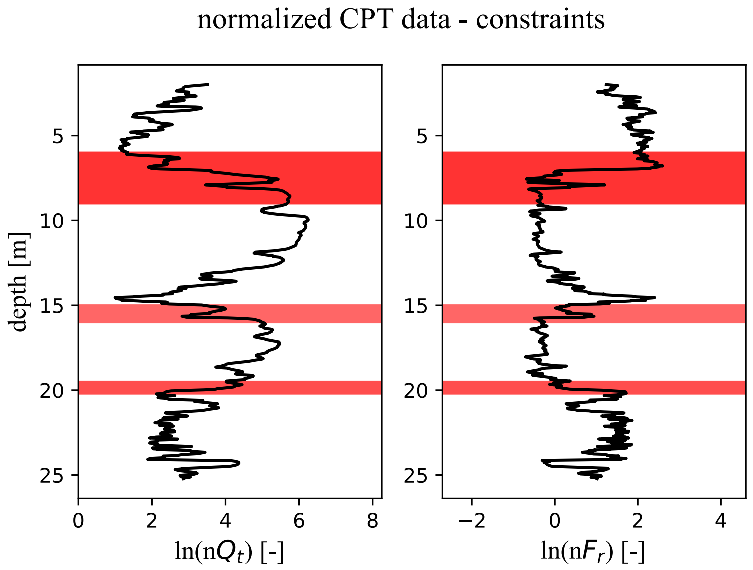


***Figure 3.3**: CPT profile including the constraints: [6, 9, 2], [15, 16, 4] and [19.5, 20.2, 3].*

2. Returning list with X best configurations

The likelihood function (or `P_Eps` in the python code) is calculated for many different soil layer thickness configurations. Each different soil layer thickness configuration will therefore lead to a unique `P_Eps` value. From all the different layer thickness configurations for which the objective function is calculated, a selection of configurations is returned giving the minimal value of the objective function; the size of this selection is defined by `no_guesses`. The function output is a matrix, `best_conf`, for which the number of rows is equal to `no_guesses`. The number of columns is dependent on the model class where in the first column (=`best_conf[:,0]`) the objective function values are stored. In the remaining columns, the layer thicknesses are stored. For model class 4, `best_conf` will (e.g.) look like this:

$$\begin{bmatrix} [200 & 4 & 5 & 3 & 6] \\ [198 & 2 & 6 & 5 & 5] \\ [201 & 6 & 5 & 1 & 6] \\ [\ldots & \ldots & \ldots & \ldots & \ldots] \end{bmatrix}$$

Where 200, 198 and 201 are the values for the objective function. The other numbers represent soil layer thicknesses.

3.4.3 Optimizing

For low model classes, the matrix that follows from 3.4.2 Initial guesses is not further optimized. The most probable thickness configuration is taken as the one giving the lowest value for the likelihood function. For the example above, this would lead to the second configuration.

For higher model classes, each configuration that follows from 3.4.2 Initial guesses is further optimized, for reasons stated in 3.4.1 Brief overview. This is done within the function `optimizer` using the `scipy.optimize` package.

For this study, the basin-hopping algorithm was found to be most efficient. The algorithm can deal with functions that have many minima, which is the case for this study. Figure 3.4 schematically explains the workings of the basin-hopping algorithm. Starting from an initial function-value, the algorithm tries to change the input parameters as such that a lower function-value is found. The algorithm does this step-wise. Each step, either (1) a less favorable condition is found and thus neglected or (2) a more favorable condition is found which then will function as the starting point for the next iteration. Iterating many times will then lead to a minimum value.

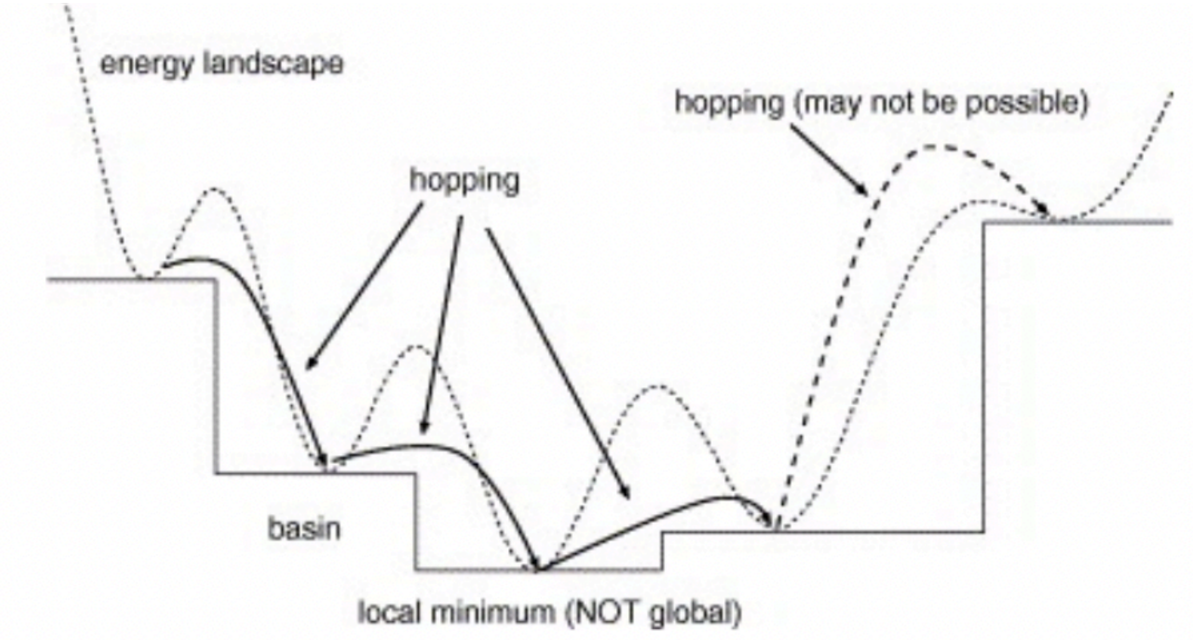


Figure 3.4: *Basin-hopping example. Source: https://ars.els-cdn.com/content/image/1-s2.0-S0009261404016082-gr1.jpg*

Note, however, that for a single optimization there is no way of ensuring that this is a *local* minimum rather than a *global* minimum value. This is illustrated in Figure 3.5 where obtaining a local or global minimum is fully dependent on the starting point (or ‘initial guess’) of the optimization. This relation between local and global minima is the reason for the part described in 3.4.2 Initial guesses. By optimizing for multiple different starting points (i.e. the most optimal different initial guesses), it is more likely to obtain the global minimum; thus, comparing all optimized configurations and selecting the one leading to the minimum value of the objective function.

The basin-hopping algorithm allows to include boundary conditions and constraints, which can be very helpful. For example, below is the list of the current applied constraints and boundary conditions:

- *Constraint 1:* the sum of all soil layer thicknesses must equal the total thickness of the CPT profile.
- *Constraint 2:* soil layer thicknesses cannot be less than the minimal thickness specified by the user
- *Boundary condition 1:* the thickness of a soil layer cannot exceed the total thickness of a CPT profile (if not specified it was found that, even with constraint 2, constraint 1 was met by having negative thicknesses).

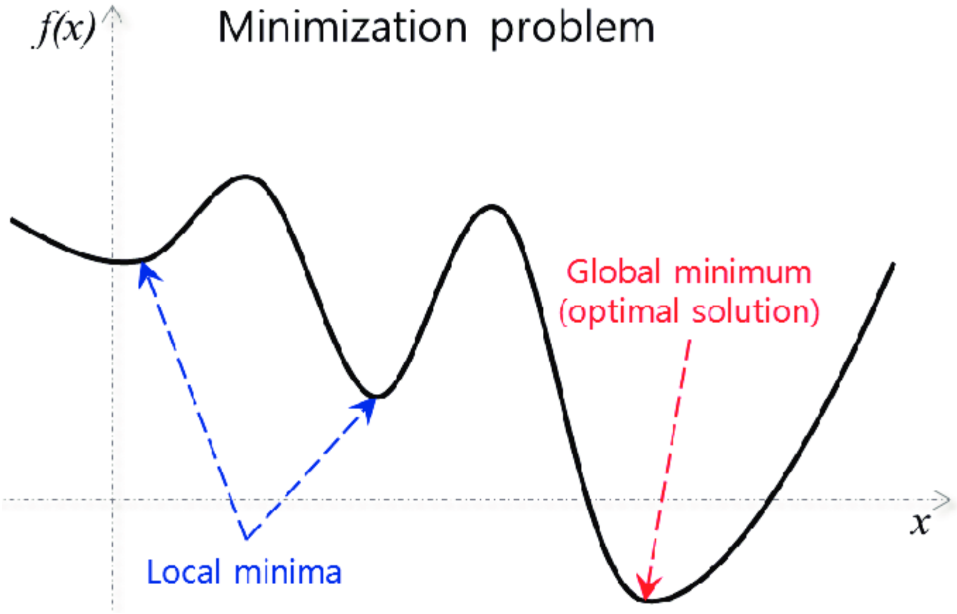


***Figure 3.5**: difference between local and global minimum. Source:* https://www.researchgate.net/publication/322270023/figure/fig3/AS:667627483562008@1536186204657/Example-of-local-and-global-solutions-in-an-optimization-problem.png

3.5 Detailed code description: Obtain the most probable model class

When the most probable soil layer thicknesses are calculated for all model classes (3.4 Detailed code description: Obtain the most probable thicknesses per model class), the simulation is completed by calculating the most probable model class. Compared to the previous section, this is done in a few coding lines. For each model class, the minimum value of the objective function is multiplied with the prior distribution (Eq. [5]). This product is then divided by H_t^{N-1} according to Eq. [15], to obtain the conditional probability. The most probable model class is then taken as the one leading to the highest value of the conditional probability. Lastly, the most probable soil types are found. This is achieved by calculating $P\left(\underline{\xi}_n \middle| \underline{\sigma}_n, N\right)$ (Eq. [8], the probability that all datapoints in the nth layer belong the either of the 9 soil types specified on the Robertson chart) for each soil layer and selecting the soil type for that layer resulting in the highest value for $P\left(\underline{\xi}_n \middle| \underline{\sigma}_n, N\right)$.

Chapter 4 – Case Study

The model approach described in chapter 2 and 3 is applied to an existing CPT profile in the proximity of the Groningen Village (Fig. 1.2). As mentioned in 1.2 Scope of This Study, this CPT profile describes a stratigraphy in which 20 different layers are separated, following a deterministic classification. Based on these subdivisions, 5 regions can be distinguished: (1) clays between 2-7 m, (2) sands between 7-14 m, (3) peat between 14-15 m, (4) sands between 15-20 m and (5) clays at depths >20 m.

4.1 The most probable soil layer thicknesses

The most probable soil layer thicknesses are calculated following the approach described in Chapter 2.2 (2.2 Calculating the most probable soil layer thicknesses for model class M_N). The results are obtained adopting the input parameter values in Table 4.1.

(1) Boundaries that are found by lower model classes (e.g. $M_2 - M_5$) are considered soil boundaries as well in the higher model classes (e.g. $M_6 - M_9$); for example, the soil boundary at ca. 7.4 m depth is considered in all model classes $M_2 - M_9$. This leads to a pattern where for each higher model class, a new soil boundary is added while the other soil boundaries are similar to the ones found by the previous model classes. One exception to this is the boundary at ca. 19.9 m depth found by M_8, which is absent in M_9. This deviation can be ascribed to the relatively low number of iterations ('guesses iterations' in Table 4.1) considered. This value is significantly lower than the total number of possibilities of soil layer thicknesses for model 8; for a CPT profile thickness of $23.22\ m$ and a minimum layer thickness of $0.1\ m$, a total of $6 * 10^{18}$ soil layer configurations are possible. Although the code involves an optimization of the best configurations, deviations are likely to occur at the higher model classes. An obvious solution would be to increase the number of iterations, but that leads to significantly more computation times.

(2) The most probable layer boundaries that follow for each model class complement well with the CPT data. For example, in Fig. 4.1 the soil boundaries based on M_6 are added to the CPT profile. They suggest that the model correctly identifies different soil layers depending on the model class.

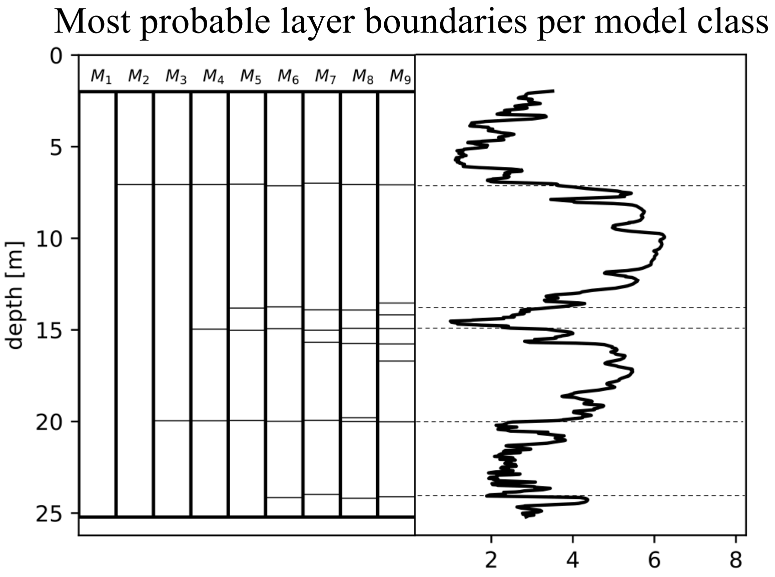


Figure 4.1: Results of the Bayesian approach calculating the most probable layer thickness configurations per model class.

Table 4.1: Input values used in the case study

Parameter	Input
N_{min}	1
N_{max}	9
$Thickness_{min}\ (in\ m)$	0.1
σ_{Fr}	1
σ_{Qt}	1.2
Implement constraints	"no"
Probability iterations	200.000
Number of guesses	20
Guesses iterations	400.000
Threshold model class	4

4.2 The most probable model class

Following the method described in chapter 2.3 (2.3 Obtaining the Most Probable Model Class), the results for obtaining the most probable model class are summarized in table 4.2. As defined, the model class with the maximum value of $\ln\,[P\xi|M_N]$ is considered the most probable model class. For model classes $M_1 - M_6$, the value of $\ln\,[P\xi|M_N]$ increases from -470.9 to -208.8, after which the value decreases to -214.3 between model class $M_7 - M_8$. It becomes evident that the most probable number of soil layers is 6. Subsequently, the most probable soil classification types are calculated for this model class leading to the following stratigraphy: (1) clay from $2.3 - 7\ m$, (2) sand from $7 - 14\ m$, (3) clay from $14 - 15\ m$, (4) sand from $15 - 20\ m$, (5) clay from $20 - 24.5\ m$ and (6) sand mixtures from $24.5 - 25\ m$.

***Table 4.2**: results of the Bayesian class selection*

		Most probable thickness, h_N^* (m)								
Model Class M_N	ln[P(ξ|M_N)]	h_1^*	h_2^*	h_3^*	h_4^*	h_5^*	h_6^*	h_7^*	h_8^*	h_9^*
M_1	-470.9	23.08	-	-	-	-	-	-	-	-
M_2	-349.8	5.06	18.02	-	-	-	-	-	-	-
M_3	-241.9	5.06	12.9	5.12	-	-	-	-	-	-
M_4	-245.2	5.06	7.9	5.0	5.12	-	-	-	-	-
M_5	-211.5	5.05	6.77	1.21	4.92	5.14	-	-	-	-
M_6	-208.8	5.14	6.61	1.19	5.05	4.17	0.92	-	-	-
M_7	-210.8	5.0	6.91	1.11	0.66	4.25	4.04	1.1	-	-
M_8	-211.2	5.06	6.86	1.0	0.83	4.06	0.21	4.18	0.89	-
M_9	-214.3	5.09	6.45	0.64	0.75	0.84	0.93	3.32	4.09	0.97

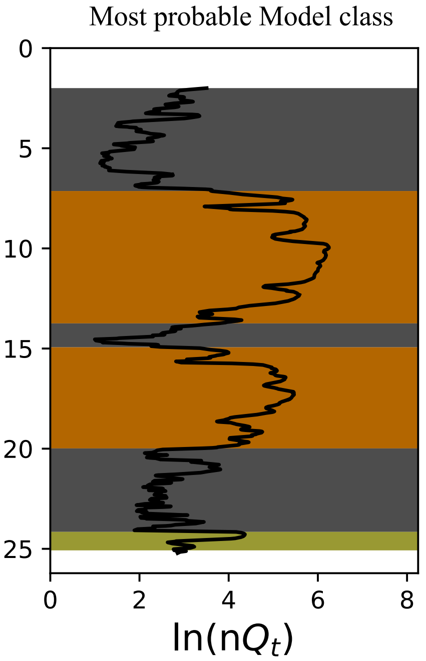


***Figure 4.2**: soil types for the most probable number of soil layers and soil*

4.3 Discussion

Figure 4.3 summarizes the results of the Bayesian approach and compares it to nearby available borehole data and the deterministic analysis. The comparison between the original CPT data and Generalized CPT data is also included (see 3.2.3 Generalize matrix). This comparison indicates that although the total number of datapoints is reduced, no critical information is lost. Note, that the results are acquired by setting the minimal thickness to $0.1\ m$. As the generalization is directly related to this value, adopting a higher minimal thickness might result in the loss of critical information.

4.3.1 Comparison with deterministic approach

The study results are generally in good agreement with those acquired from the deterministic analysis (Fig. 43). The general trend of the soil layers based on the deterministic analysis are well captured by the model, separating the sand layers from the clay and/or peat layers. At ca. $12\ m$ depth, however, the model indicates the presence of clay, whereas the deterministic analysis suggests a peat layer instead, also being slightly thinner ($1.2\ m$ compared to $0.95\ m$). One explanation for this difference is that the deterministic analysis is based on engineering judgement. The judgement for this layer is heavily influenced by the 'spike' of the CPT profile. Based on this 'spike' alone, this layer would be classified as peat. Considering the other measurements in that layer as well, however, which is done by the model, leads to a more considerate result, being a clay layer. In addition to this, it must be noted that there is also no consensus on the presence of a peat layer by the available borehole data around this depth. Around this depth, the soil type is marked as peat by $BH2$ and as clay by $BH1$.

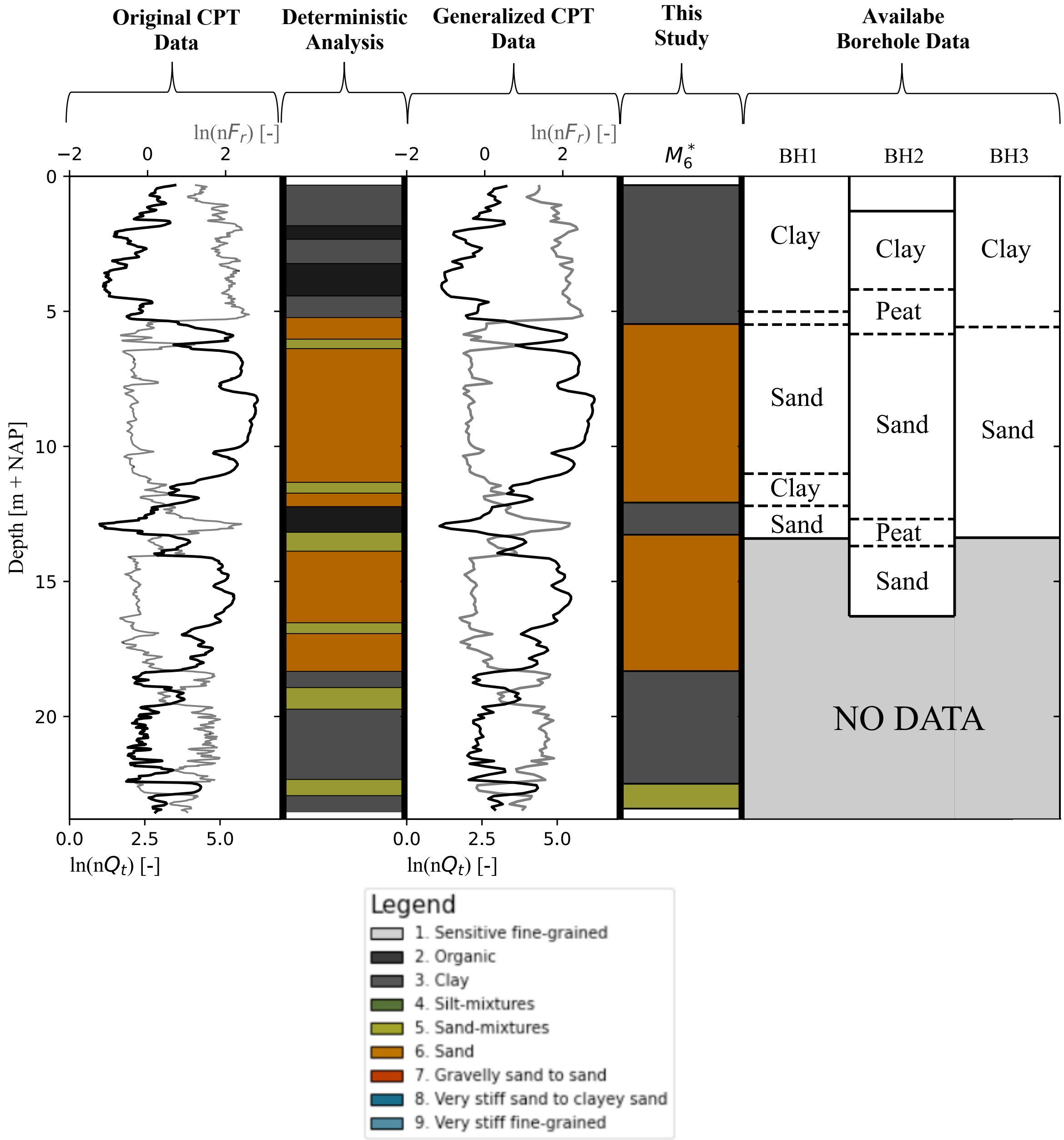


***Figure 4.3**: Model results and comparisons to other methods.*

4.3.1 Limitations

The model does not indicate the presence of a peat layer within the top $5\,m$ of the soil profile, whereas this layer is clearly present based on the deterministic analysis. BH_1 and BH_2 also distinguish a peat layer overlying the sand layer around $5\,m$ depth (Fig. 4.3). The absence of this layer following the model approach might be related to the relatively low numbers of soil layers that resulted from the model. However, for the higher model classes ($M_7 - M_9$) this layer is also absent. This might be related to the problem posed in Figure 4.4, illustrating the example of a joint Gaussian distribution around a single datapoint, similar too Figure 2.2. However, rather than adopting a datapoint in the center of the Robertson chart, a measurement close to the edges of the Robertson chart is taken. The same Gaussian distribution as in Figure 2.2 then shows a “cut-off” at the edges of the Robertson chart, leading to a biased distribution. As a result of this cut-off, the chance of that datapoint belonging to soil type 3 increases compared to the change of it belonging to soil type 2. The effect of this cut-off is not implemented in the current model, possibly explaining the absence of peat layers in the model results. This problem is not mentioned by Wang et al. (2013), suggesting that they (1) implemented a simple solution that did not need explanation or (2) did not take this problem into account. It must be noted, however, that the deterministic analysis of CPT profile analyzed by Wang et al. (2013) does not distinguish peat layers. Therefore, it is also likely that Wang et al. (2013) did not face this problem at all, thus not requiring a solution to this problem. However, the results following from case study conducted here suggest that this “cut-off” might affect the results, mainly for CPT measurements that fall within the vicinity of the edges of the Robertson chart. A subsequent study on the impact of this cut-off and its solutions is therefore suggested. A simple solution would be to account the datapoints (for the Monte Carlo analysis in 3.3 Detailed code description: Calculate probability per datapoint) that fall outside the Robertson chart to the nearest soil type. This introduces the assumption that large $\ln{(nF_r)}$ and $\ln{(nQ_t)}$ values have an equal chance of occurrence compared to low values of $\ln{(nF_r)}$ and $\ln{(nQ_t)}$. This, of course, does not hold as the Robertson chart is based empirical data, excluding measurement values that are unlikely to be found (e.g. $\ln{(nF_r)}$ values > 2.5). A different solution could be to alter the statistical parameters (σ_{F_r} and σ_{Q_t}) of the joint Gaussian curve depending on the position of the datapoint within the Robertson chart. This, however, introduces several complications including the effect of boundary conditions and the prior distribution ($P(\underline{\Omega}_N|N)$, Eq. [5]) no longer being uniform for every layer. Lastly, a solution is proposed to base the statistical parameters (σ_{F_r} and σ_{Q_t}) on the non-logarithmic values for nF_r and nQ_t, resulting in an asymmetric shape on the joint Gaussian distribution on the logarithmic Robertson chart. This, however, requires an evaluation of the parameters values of σ_{Fr} and σ_{Qt} that will lead to the most-optimal results.

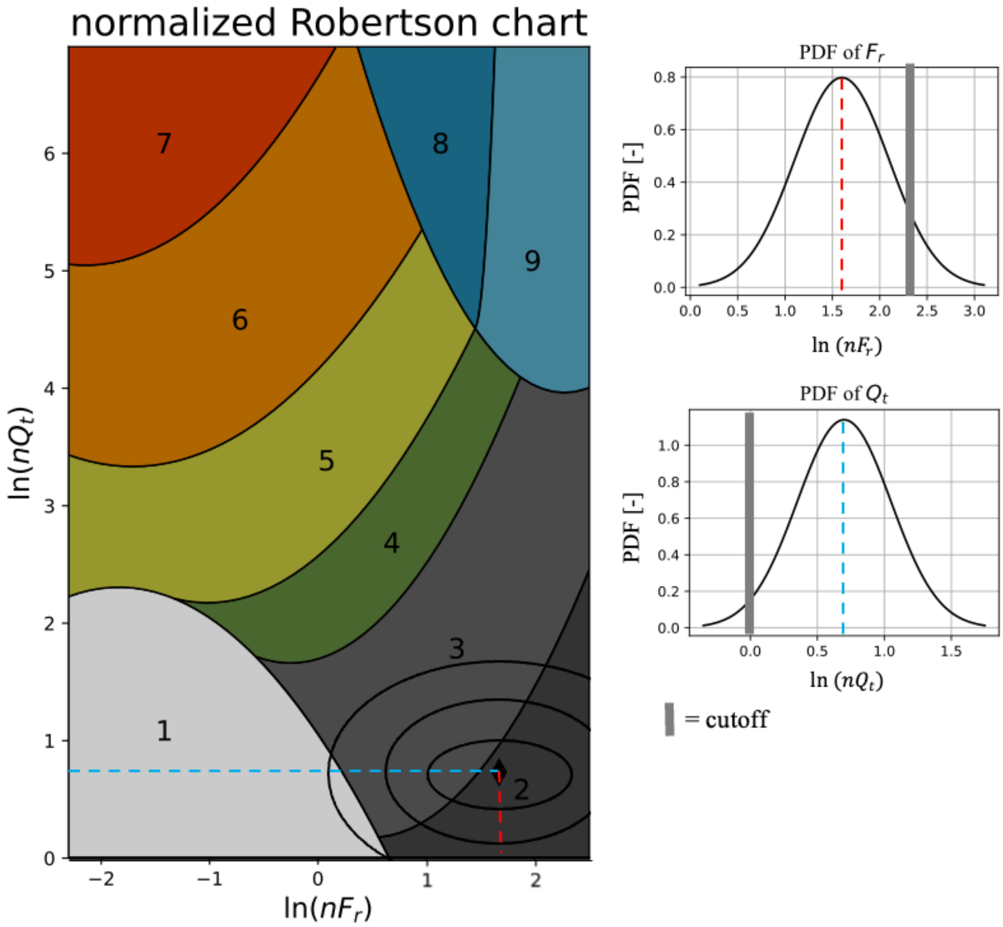


***Figure 4.4**: Illustrative example of the proposed problem when calculating the probability per datapoint. The Gaussian distribution is cut off at the edges of the Robertson Chart.*

4.3.2 Optimization

Running the code can be quite time consuming, depending on the number of model classes used within the simulation and the number of soil layers considered. Chapter 3.4.3 Optimizing proposes an optimizing module to increase the model efficiency and accuracy specifically for high model classes. Without this optimizing module, to obtain similar results, the computation time was found to be in the order of 15 hours. Including this module significantly enhanced the processes, lowering this computation time to roughly 2 hours[1].

As described, the input for the optimizing function is a list of the most optimal 'initial guesses' based a Monte Carlo analysis in which soil layer thickness configurations are randomly generated. However, the current version of the code does not include a revised selection of these optimal initial guesses. The code only selects the X most optimal initial guesses without taking similarities into account. This means that optimizing for multiple initial guesses with large (or even identical) similarities is possible. For example, table 4.2 shows the 20 most optimal configurations (=initial guesses) for model class 7 based on 400.000 randomly generated soil layer thickness configurations. At the top of the table, several soil layer thickness configurations are grouped together based on their similarities. By obtaining similar initial guesses, the total number of initial guesses is effectively reduced; in this case, the model optimizes for 16 different soil thickness configurations, rather than 20. This, of course, affects the model accuracy as the input variable '`no_guesses`' (3.1.1 Input parameters) is an 'apparent' quantity. Future studies should therefore consider if a 20% loss is acceptable or not. If not, one could either (1) increase the value for '`no_guesses`' to account for the loss in accuracy or (2) implement an additional module to ensure that no similarities between initial guesses are allowed.

Although the implemented optimizing module leads to great improvements regarding efficiency, additional optimization is needed to further improve the model. For example:

- Depend the number of optimal configurations (= initial guesses) on the number of soil layers for a model class; e.g. increase `no_guesses` for higher model classes to allow for a wider range of initial guesses.
- If the user has previous knowledge on the area regarding the number of soil layers, the model could be improved by specifying a minimal number of soil layers to consider. By doing this, the model efficiency increases as irrelevant model classes are disregarded.
- The `scipy.optimize` package has a wide range of possibilities to optimize the solver (e.g. specify tolerance levels, step sizes, etc.). A study on the sensitivity and computing performance related to these parameters could lead to significant improvements.

Table 4.2: *20 most optimal initial guesses for model class 7. Several configurations are grouped together based on their similarities.*

'initial guess'	h_1	h_2	h_3	h_4	h_5	h_6	h_7
1	5.063	4.9	1.8	1.3	5	4	1.017
2	5.063	4.6	2.2	1.4	4.7	4.2	0.917
3	4.963	0.9	5.9	1.2	5.1	4	1.017
4	5.063	1.1	5.7	1.3	4.7	4.1	1.117
5	5.063	3.4	3.4	1.2	4.8	4.1	1.117
6	4.963	3.2	3.6	1.2	5	4.3	0.817
7	5.163	5.8	1	1.1	0.5	4.3	5.217
8	5.163	6.1	1	0.8	0.8	4.1	5.117
9	4.963	7	1	0.6	4.4	3.7	1.417
10	4.963	6.8	2	4.3	4	0.5	0.517
11	5.063	6.6	1.3	1.8	3.2	4.1	1.017
12	1.563	3.5	6.6	1.5	4.7	4.2	1.017
13	5.063	5.9	1	1	4.8	0.2	5.117
14	5.063	6.8	1.3	2.9	1.8	4.2	1.017
15	5.063	6.1	0.7	1.1	4.9	4.3	0.917
16	4.663	0.3	6.9	1.1	4.9	4.2	1.017
17	4.463	0.6	6.7	1.3	5	4	1.017
18	5.063	5.9	1.1	1	4.9	1.8	3.317
19	5.063	6.7	0.6	0.7	4.8	4.3	0.917
20	4.963	0.2	5.9	1.2	0.7	5	5.117

[1] On a Mac Mini (M1) with 8 gb ram.

Chapter 5 – Conclusion

The Bayesian approach proposed by Wang et al. (2013) to interpret CPT soil profiles taking the underlying uncertainty of the data measurements into account, is implemented into the Python coding environment. First, the code defines a set of model classes for a given CPT profile, each model class increasing in complexity; that is, the total number of soil layers considered. Secondly, for each model class, the most probable soil layer thicknesses are determined after which all model classes are compared to obtain the most probable number of soil layers. This results in a CPT profile interpretation in which the most-likely soil layers (soil type, thickness and position) are distinguished. The results following from a case study on a $23\ m$ CPT profile are promising. For the given CPT profile, the model identifies 6 soil layers which agree with the results following the deterministic approach and observational data from nearby located boreholes. However, the soil types of CPT measurements within the vicinity of the edges of the Robertson chart are incorrectly identified as a result of a "cut-off"-effect of the joint Gaussian distribution describing the uncertainty of a single datapoint. This is most likely related to the integration of the statistical parameters within the model, which should be examined in a subsequent study.

Additionally, a simple optimization module is integrated within the model to improve the efficiency for higher model classes. However, this module still holds several simplifications (e.g. allowing multiple 'initial guesses' to be similar leading to an 'apparent' accuracy) as well as being insufficient for higher model classes. It is therefore suggested to implement additional optimization modules that allow for more accurate results for higher model classes while maintaining reasonable computation times.

References

Arel, E. (2012). *Predictin the spatial distribution of soil profile in Adapazari/Turkey by artificial neural networks using CPT data.* Computers & Geosciences, vol. 43, pp. 90-100.

Berbee, B. M., Fennis, F. (2019). *Rapportage – Handreiking grondonderzoek voor Piping.* POV Piping, dijkinnovatie van binnenuit.

Bleistein, N., Handelsman, R. (1986). *Asymptotic expansions of integrals.* Dover, New York.

Cao, Z., Wang, Y. *Bayesian Approach for Probabilistic Site Characterization Using Cone Penetration Tests.* Journal of Geotechnical and Geoenvironmental Engineering, vol. 139, pp. 267-276.

Cao, Z., Zheng, S., Li, D., Phoon, K. (2019). *Bayesian identification of soil stratigraphy based on soil behaviour type index.* Canadian Geotechnical Journal, vol. 56, pp. 570-586.

Fenton, G. A. (1999). *Random Field Modelling of CPT Data.* ASCE Journal of Geotechnical and Geoenvironmental Engineering, vol. 125, pp. 486-498.

Hicks, M. A. (2005). *Risk and Variability in Geotechnical Engineering.* Géotechnique, vol. 55, pp. 1-2.

Kurup, P. U., Griffin, E. P. (2006). *Prediction of Soil Composition from CPT Data Using General Regression Neural Network.* Journal of Computing in Civil Engineering, vol. 20, issue 20.

Lacasse, S., Nadim, F. (1998). *Risk and Reliability in Geotechnical Engineering.* Proceedings: Fourth International Conference on Case Histories in Geotechnical Engineering.

Luo, N., Bathurst, R. J. (2018). *Probabilistic analysis of reinforced slopes using RFEM and considering spatial variability of frictional soil properties due to compaction.* Georisk: Assessment and Management of Risk of Engineered Systems and Geohazards, vol. 12, pp. 87-108.

Reale, C., Gavin, K., Libric, L., Juric-Kacunic, D. (2018). *Automatic classification of fine-grained soils using CPT measurements and Artificial Neural Networks.*

Robertson, P. K. (1990). *Soil classification using the cone penetration test.* Canadian Geotechnical Journal, vol. 27, pp. 151-158.

Robertson, P. K. (2009). *Interpretation of cone penetration tests – a unified approach.* Canadian Geotechnical Journal, vol. 46, pp. 1337-1355.

Robertson, P. K. (2010). *Soil behaviour type from the CPT: an update.* 2nd International Symposium on Cone Penetration Testing, CA, USA.

Robertson, P. K., Cabal, K. L. (2010). *Estimating soil unit weight from CPT.* 2nd International Symposium on Cone Penetration Testing, CA, USA.

Tillman, A., Englert, A., Nyary, Z., Fejes, I., Vanderborght, J., Vereecken, H. (2007). *Characterization of subsoil heterogeneity, estimation of grain size distribution and hydraulic conductivity at the Krauthausen test site using Cone Penetration Test.* Journal of Contaminant Hydrology, vol. 7, pp. 57-75.

Vessia, G., Di Curzio, D., Castrignanò, A. (2020). *Modeling 3D soil lithotypes variability through geostatistical data fusion of CPT parameters.* Science of The Total Environment, vol. 698.
Wang, Y., Au, S. K., Cao, Z. (2010). *Bayesian approach for probabilistic characterization of sand friction angles.* Engineering Geology, vol. 144, pp. 354-363.

Wang, Y., Huang, K., Cao, Z. (2013). *Probabilistic identification of underground soil stratification using cone penetration tests.* Canadian Geotechnical Journal, vol. 50, pp. 766-776.

Wang, Y., Huang, K., Cao, Z. (2014). *Bayasian identification of soil strata in London clay.* Géotechnique, vol. 64, pp. 239-246.

Appendices: Python code

Appendix A: Main.py

```
import pandas as pd
import numpy as np
import robertson as Frob
import read_cpt as Fread
import posterior as Fpost
import os
import matplotlib.pyplot as plt

"""
#####################################
#------------1. Input------------#
#####################################
"""

params = {
    "cpt":              "CPT000000097976_IMBRO_A.gef",   #cpt gef file
    "N min":            1,                         #minimum soil layers to consider
    "N max":            9,                         #maximum soil layers to consider
    "min thickness":    0.1,                       #minimum layer thickness
    "std Fr":           1,                         #standard deviation of friction ratio
    "std Qt":           1.2,                       #standard deviation of cone resistance
    "Table number":     1,                         #if desired to use constraints: "yes".
    "implement constraints": "no",                 #if desired to use constraints: "yes".
    "boundary constraint": np.array([[6,9,2],      #specify constraints
                                     [19.5,20.2,3],
                                     [15,16,4]]),
    "probability iterations": 200000, #number of iterations for probability per datapoint
    "treshold model class": 4,       #largest model class that does not need optimization
    "no. guesses": 20,   #number of initial guesses used for optimization per model class
    "guesses iterations": 400000     #number of iterations of which no. guesses are stored
    }

"""
#####################################
#-----------2a. read cpt---------#
#--------2b. generalize cpt--------#
#####################################
"""
#2a.
matrix = Fread.read_cpt(params)                    #read cpt file and store in matrix
Fread.plot_boundary_constraints(matrix,params)     #plot figure with boundary constraints

#2b.
matrix_generalized = Fread.generalize(matrix,params)   #generalize matrix

Frob.plot_Robertson(matrix_generalized)  #plot generalized data on Robertson chart
print("Obtain matrix: SUCCES")

"""
#####################################
#--------3 get probability per data point---------#
#####################################
"""

Polygons = Frob.plot_Robertson(matrix_generalized)    #get polygons from robertson chart

#check if the file with probability per datapoint exists, if not: make one
if os.path.isfile('prob_generalized2.csv') == False:
    #get probability per measurement
    matrix_generalized = Frob.MC_probability(matrix_generalized,Polygons,params)
    np.savetxt('prob_generalized2.csv', matrix_generalized,delimiter=";",
                      header='depth;nQt;nFr;P_1;P_2;P_3;P_4;P_5;P_6;P_7;P_8;P_9')

```

```
if os.path.isfile('prob_generalized2.csv') == True:
    matrix_generalized = np.array(pd.read_csv('prob_generalized2.csv', sep=';'))

print("get probability per datapoint: SUCCES")

"""
####################################
#---------4. calculate most probable N and layer thicknesses-----------#
####################################
"""
#calculate likelihood function and conditional probability per model class
final_result,summary= Fpost.calculator(matrix_generalized,params)

"""
####################################
#---------5. plot results-----------#
####################################
"""
Fread.plot_summary(matrix,summary,final_result,params)

```

Appendix B: Read_cpt.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math

def find_columns(cpt):
    """
    Function to read gef file and find which columns contain what values
    """
    data = open(cpt,'r')

    for row in data:
        if row[0]=='#':
            if row[-11:-1] == 'diepte, 11':
                lengthcol = int(row[13])-1
            elif row[-18:-1] == 'conusweerstand, 2':
                conuscol = int(row[13])-1
            elif row[-25:-1] == 'plaatselijke wrijving, 3':
                sleevecol = int(row[13])-1
            elif row[-20:-1] == 'waterspanning u2, 6':
                u2col = int(row[13])-1
        else:
            break
    columns = [lengthcol, conuscol, sleevecol, u2col]
    columns = np.sort(columns)
    return columns


def read_cpt(params):
    """
    Function to read cpt data and store them in a matrix
    """

    cpt = params["cpt"]
    no_of_columns = 12 #if more columns are included, add those here
    #0 = length
    #1 = conus
    #2 = sleeve
    #3 = u2
    #4 = fr
    #5 = I_sbt
    #6 = volumetric weigth soil
    #7 = total vertical stress
    #8 = effective vertical stress assuming ps is at z=0
    #9 = normalized cone stress
    #10 = normalized friction ratio
    #11 = corrected cone resistance

    """ Error values """
    lengtherror = 999.999
    conuserror = 999.999
    sleeveerror = 9.999
    u2error = 99.999

    pa = 0.1        #atmospheric pressure
    yw = 10         #unit weight of water
    alpha = 0.8     #coefficient to correct for pore pressures

    """ read columns from gef file """
    columns = find_columns(cpt) #read the columns from gef file
    df = pd.read_csv(cpt, header=None, delimiter=';',comment='#', usecols=columns)
    matrix = np.zeros((len(df),no_of_columns)) #create matrix in which data is stored

    #0: depth
    matrix[:,0] = np.array(df.iloc[:,1])

    #1: conus resistance
    matrix[:,1] = np.array(df.iloc[:,0])

    #2: sleeve friction
    matrix[:,2] = np.array(df.iloc[:,2])

    #3: u2
    matrix[:,3] = (matrix[:,0] * 0.01) -0.5*0.01
    matrix[:,3] = matrix[:,3]*(matrix[:,3]>0)

    #check if first value has error and if so, give it the closest 'real' value
    if matrix[0,0] == lengtherror:
        matrix[0,0]=matrix[np.where(matrix[:,0]!=lengtherror)[0][0],0]
    if matrix[0,1] == conuserror:
        matrix[0,1]=matrix[np.where(matrix[:,1]!=conuserror)[0][0],1]
    if matrix[0,2] == sleeveerror:
        matrix[0,2]=matrix[np.where(matrix[:,2]!=sleeveerror)[0][0],2]
    if matrix[0,3] == u2error:
        matrix[0,3]=matrix[np.where(matrix[:,3]!=u2error)[0][0],3]

    #if errorvalue is found, change it to the value above
    for point in range(len(matrix)):
        if matrix[point,0] == lengtherror:
            matrix[point,0] = matrix[point-1,0]
        if matrix[point,1] == conuserror:
            matrix[point,1] = matrix[point-1,1]
        if matrix[point,2] == sleeveerror:
            matrix[point,2] = matrix[point-1,2]
        if matrix[point,3] == u2error:
            matrix[point,3] = matrix[point-1,3]

    """ calculate other columns """

    #11: corrected cone resistance
    matrix[:,11] =  matrix[:,1]

    #4: fr
    matrix[:,4]= (matrix[:,2]/matrix[:,11])*100

    #5: Isbt
    matrix[:,5] = ((3.47-np.log10(matrix[:,11]/pa))**2)

    #6: volumetric soil weight kN/m3 (Robertson, 1990)
    matrix[:,6] = (yw*(0.27*np.log10(matrix[:,4])+
                    0.36*np.log10(matrix[:,11]/pa)+1.236))

    for ii in range(len(matrix)):
        if matrix[ii,6] == np.inf or matrix[ii,6] == -np.inf:
            matrix[ii,6] = matrix[ii-1,6]

    #7: total vertical stress
    matrix[0,7] = matrix[0,6]*matrix[0,0]/1000

    for ii in range(len(matrix)-1):
        matrix[ii+1,7] = matrix[ii,7]+(((matrix[ii+1,0]-
                             matrix[ii,0])*matrix[ii+1,6])/1000)

        if math.isnan(matrix[ii+1,7])==True:
            matrix[ii+1,7] = matrix[ii,7]

    #8: effective vertical stress
    matrix[:,8] = matrix[:,7]-matrix[:,3]

    #9: normalized cone resistance
    matrix[:,9] = (matrix[:,11]-matrix[:,7])/matrix[:,8]

    #10: normalized friction ratio
    matrix[:,10] = matrix[:,2]/(matrix[:,11]-matrix[:,7])*100

    return matrix
```

```


def generalize(matrix, params):
    """
    Function to generalized the cpt matrix by a ratio based on the min_thickness.
    e.g. if min_thickness = 0.02 (= the general spacing between 2 cpt measurements), no
    generalization will be done
    e.g. if min_thickness = 0.1, every 5 points (=0.1/0.02) an average value will be
    taken for nQt and nFr.
    in addition 9 new (empty) columns are added in which the probability of MC are be
    stored.
    """

    min_thickness = params["min thickness"]
    no_of_columns = 12
    #0 = depth generalized
    #1 = log of nQt generalized
    #2 = log of nFr generalized
    #3-11 = for probabilities; for now empty

    generalize_ratio = min_thickness/0.02
    length_generalized = math.floor(round(len(matrix)/generalize_ratio,1)) #new length
    matrix_generalized = np.zeros((length_generalized,no_of_columns)) #Create matrix

    #0 = depth generalized
    matrix_generalized[:,0]=  np.mean(matrix[0:int(length_generalized*
                                       generalize_ratio),0].reshape(-
                                       1,int(generalize_ratio)),axis=1)

    #1 = log of nQt generalized
    matrix_generalized[:,1]= np.log(np.mean(matrix[0:int(length_generalized*
                                       generalize_ratio),9].reshape(-
                                       1,int(generalize_ratio)),axis=1))

    #2 = log of nFr generalized
    matrix_generalized[:,2] = np.log(np.mean(matrix[0:int(length_generalized*
                                       generalize_ratio),10].reshape(-
                                       1,int(generalize_ratio)),axis=1))

    return matrix_generalized

def plot_boundary_constraints(matrix,params):
    """
    Function that plots normalized data and boundary constraints. Function also checks
    if boundary conditions are valid
    """
    Tfont ={'fontname':'Times New Roman'}

    if params["implement constraints"] == "yes":

        boundary_constraint = params["boundary constraint"]

    if params ["implement constraints"] =="no":
        boundary_constraint=[]

    N_max = params["N max"]

    #check if boundary constraints are valid
    if len(boundary_constraint) >0:
        if max(boundary_constraint[:,-1]) >=N_max:
            raise ValueError("constraint given for boundary that exceeds N_max")

        for ii in range(1,N_max):
            if np.count_nonzero(boundary_constraint == ii) >ii:
                raise ValueError("too many boundary constraints given for N=",ii)

    fig, (ax1,ax2) = plt.subplots(1,2)
    fig.suptitle('normalized CPT data - constraints',**Tfont, fontsize=15)
    ax1.plot(np.log(matrix[:,9]),matrix[:,0], color='k')
    ax1.set_xlabel(r'ln(n$Q_t$) [-]', **Tfont, fontsize=13)
    ax1.set_ylabel('depth [m]', **Tfont, fontsize=13)
    ax1.set_xlim(0,1.5*max(np.log(matrix[:,9])))
    ax1.set_ylim(0,matrix[-1,0]+1)

    ax1.invert_yaxis()
    ax1.set_xlim(0,max(np.log(matrix[:,9])+2))
    ax2.plot(np.log(matrix[:,10]),matrix[:,0],color='k')
    ax2.set_ylim(0,matrix[-1,0]+1)
    ax2.set_xlabel(r'ln(n$F_r$) [-]', **Tfont, fontsize=13)
    ax2.invert_yaxis()
    ax2.set_xlim(min(np.log(matrix[:,10])-2),max(np.log(matrix[:,10])+2))
    for ii in range(len(boundary_constraint)):
        ax1.fill_between(np.arange(0,12),boundary_constraint[ii,1],
          boundary_constraint[ii,0], edgecolor='red', facecolor='red',alpha=1-
          boundary_constraint[ii,2]/N_max)
        ax2.fill_between(np.arange(-10,10),boundary_constraint[ii,1],
          boundary_constraint[ii,0], edgecolor='red', facecolor='red',alpha=1-
          boundary_constraint[ii,2]/N_max)

    plt.savefig('constraints.pdf')
    plt.show()

def collor_fill(layer_type):
    if layer_type == 1:
        collor_matrix = [0,0,0,0.2]
    if layer_type ==2:
        collor_matrix = [0,0,0,0.8]
    if layer_type ==3:
        collor_matrix = [0,0,0,0.7]
    if layer_type == 4:
        collor_matrix = [0.3,0.4,0.2,1]
    if layer_type == 5:
        collor_matrix = [0.6,0.6,0.2,1]
    if layer_type == 6:
        collor_matrix = [0.7,0.4,0,1]
    if layer_type == 7:
        collor_matrix = [0.7,0.2,0,1]
    if layer_type == 8:
        collor_matrix = [0.1,0.4,0.5,1]
    if layer_type == 9:
        collor_matrix = [0.1,0.4,0.5,0.8]
    return collor_matrix


def plot_summary(matrix,summary,final_result,params):
    “”” function plotting the results of the approach “””
    Tfont ={'fontname':'Times New Roman'}

    #plot every model class!

    fig, (ax1,ax2) = plt.subplots(1,2)
    fig.suptitle('normalized CPT data', **Tfont, fontsize=20)

    #initial settings

    ax2.plot(np.log(matrix[:,9]),matrix[:,0], color='k')
    ax2.set_xlabel(r'ln(n$Q_t$)', fontsize=15)
    ax1.set_ylabel('depth [m]')
    ax2.set_xlim(0,1.5*max(np.log(matrix[:,9])))
    ax2.set_ylim(0,matrix[-1,0]+1)
    ax2.invert_yaxis()
    ax2.set_xlim(0,max(np.log(matrix[:,9])+2))
    start = matrix[0,0]

    ax1.set_xlim(0,1)
    ax1.set_ylim(0,matrix[-1,0]+1)
    ax1.invert_yaxis()
    ax1.hlines(matrix[-1,0],0,1,color='k')
    ax1.hlines(start,0,1,color='k')
    ax1.set_xticks([])

    for i in range(len(summary)):
        bounds = summary[i,1:i+2]
        ax1.vlines((1/len(summary))*i,start,matrix[-1,0],color='k')
        ax1.text((1/len(summary))*i+(1/len(summary))/2 -0.02,1.5,f'$M_{i+1}$',
                    fontsize=7.5, **Tfont)
        for j in range(len(bounds)):
            ax1.hlines(sum(bounds[:j])+matrix[0,0],(1/len(summary))*i,
                    (1/len(summary))*i+(1/len(summary)),color='k',linewidth=0.5)

    for i in range(len(final_result)):
        collor_matrix = collor_fill(final_result[i,-1])
        if i == 0:
            ax2.fill_between(np.arange(0,10),start, start+final_result[0,0],
                    facecolor=collor_matrix)
        else:
            print(start+final_result[i-1,0], start+sum(final_result[:i+1,0]))
            ax2.fill_between(np.arange(0,10),start+sum(final_result[:i,0]),
                    start+sum(final_result[:i+1,0]), facecolor=collor_matrix)

    plt.savefig('summary_depth.pdf')
    plt.show()

    #plot table with most probable thicknesses

    fig, ax = plt.subplots(figsize = (10,5))
    #domain
    ax.set_xlim(0,4+len(summary)*0.8+0.8)
    ax.set_ylim(-0.2-0.12*len(summary),1)
    #lines
    ax.hlines(0,0,1011.5, color='k',linewidth=1.5)
    ax.hlines(0.6,0,11.5, color='k',linewidth=1.5)
    ax.hlines(0.3,4,11.5, color='k', linewidth=1.5)
    #text
    ax.text(0.1,0.17,f'Model', **Tfont, fontsize=15)
    ax.text(0.1,0.07,f'Class $M_N$', **Tfont, fontsize=15)
    ax.text(1.7,0.07,r'ln[P(${\xi}$|$M_N$)]', **Tfont, fontsize=15)
    ax.text(4,0.5,'Most probable',**Tfont, fontsize=15)
    ax.text(4,0.37,r'thickness, $h^*_N$ (m)', **Tfont, fontsize=15)

    ax.axes.xaxis.set_visible(False)
    ax.axes.yaxis.set_visible(False)

    for i in range(len(summary)):
        ax.text(4+0.8*i,0.07,f"$h^*_{i+1}$", **Tfont, fontsize=15)
        ax.text(0.1,-0.1-i*0.12,f"$M_{i+1}$", **Tfont, fontsize=15)
        ax.text(1.7,-0.1-i*0.12,f"{round(summary[i,-1],1)}", **Tfont, fontsize=15)
        for j in range(len(summary[i])-2):
            value = round(summary[i,j+1],2)
            if value != 0:
                ax.text(4+j*0.8,-0.1-i*0.12,f'{value}', **Tfont, fontsize=15)
            else:
                ax.text(4+j*0.8,-0.1-i*0.12,f'  -', **Tfont, fontsize=15)

   plt.savefig('summary_table.pdf')

```

Appendix C: Robertson.py

```python
from shapely import geometry as geom
import numpy as np
import matplotlib.pyplot as plt
import descartes as des
import random

def plot_Robertson(matrix_generalized):
    """
    Get polygons
    """
    largest_nFr = max((matrix_generalized[:,2]))
    #intersection points
    A = [-2.3026,0]
    B = [0.6569,0]
    C = [-2.3026,2.2268]
    D = [2.3026,0]
    E = [2.3026,2.0234]
    F = [0.5589,0.1776]
    G = [2.3026,3.9639]
    H = [1.8687, 4.0953]
    J = [1.4505,4.5104]
    K = [-1.3334,2.2126]
    L = [0.9622,5.3534]
    M = [-2.3026,3.4335]
    N = [0.3655,6.9078]
    O = [0.1658,6.9078]
    P = [-2.3026,5.0557]
    Q = [-2.3026,6.9078]
    R = [2.3026, 6.9078]
    S = [1.6334,6.9078]
    T = [-0.5773,1.7179]
    #if Fr values lie outside of given coordinates, extrapolate chart.
    if largest_nFr>2.3026:
        D = [largest_nFr,0]
        E = [largest_nFr,0.5586*largest_nFr**2+-0.5399*largest_nFr+0.3049]
        G = [largest_nFr,0.8095*largest_nFr**2+-3.6795*largest_nFr+8.1444]
        R = [largest_nFr,6.9078]
    intersect = np.array([A,B,C,D,E,
                          F,G,H,J,K,
                          L,M,N,O,P,
                          Q,R,S,T])

    #x-range of functions
    I_range =np.arange(C[0],B[0],0.1)
    I_range1 = np.arange(C[0],K[0],0.1)
    I_range2 = np.arange(K[0],T[0],0.1)
    I_range3 = np.arange(T[0],F[0],0.1)
    I_range4 = np.arange(F[0],B[0],0.1)
    II_range =np.arange(F[0],E[0],0.1)
    III_range =np.arange(T[0],H[0],0.1)
    IV_range =np.arange(K[0],J[0],0.1)
    V_range =np.arange(M[0],L[0],0.1)
    VI_range =np.arange(P[0],O[0],0.1)
    VII_range =np.arange(N[0],G[0],0.1)
    VII_range1 = np.arange(N[0],L[0],0.1)
    VII_range2 = np.arange(L[0],J[0],0.1)
    VII_range3 = np.arange(J[0],H[0],0.1)
    VII_range4 = np.arange(H[0],G[0],0.1)
    VIII_range =np.arange(J[0],S[0],0.01)

    #y-values based on x-range
    I = -0.3707*I_range**2+-1.3625*I_range+1.0549
    I1 = -0.3707*I_range1**2+-1.3625*I_range1+1.0549
    I2 =-0.3707*I_range2**2+-1.3625*I_range2+1.0549
    I3 = -0.3707*I_range3**2+-1.3625*I_range3+1.0549
    I4 =-0.3707*I_range4**2+-1.3625*I_range4+1.0549
    II = 0.5586*II_range**2+-0.5399*II_range+0.3049
    III = 0.5405*III_range**2+0.2739*III_range+1.6959
    IV = 0.3833*IV_range**2+0.7805*IV_range+2.5718
    V = 0.2827*V_range**2+0.967*V_range+4.1612
    VI = 0.3477*VI_range**2+1.4933*VI_range+6.6507
    VII = 0.8095*VII_range**2+-3.6795*VII_range+8.1444
    VII1 = 0.8095*VII_range1**2+-3.6795*VII_range1+8.1444
    VII2 = 0.8095*VII_range2**2+-3.6795*VII_range2+8.1444
    VII3 = 0.8095*VII_range3**2+-3.6795*VII_range3+8.1444
    VII4 = 0.8095*VII_range4**2+-3.6795*VII_range4+8.1444
    VIII = 64.909*VIII_range**2+-187.07*VIII_range+139.2901


    #make lists for polygons
    #P1
    P1_X=np.concatenate((I_range, [B[0],A[0]]))
    P1_Y=np.concatenate((I,[B[1],A[1]]))
    P1_mat = np.column_stack((P1_X,P1_Y))

    #P2
    P2_X=np.concatenate((II_range,[E[0],D[0],B[0]],np.flip(I_range4)))
    P2_Y=np.concatenate((II,[E[1],D[1],B[1]],np.flip(I4)))
    P2_mat = np.column_stack((P2_X,P2_Y))

    #P3
    P3_X = np.concatenate((III_range,VII_range4,[G[0],E[0]],
                      np.flip(II_range),np.flip(I_range3[1:])))
    P3_Y = np.concatenate((III,VII4,[G[1],E[1]],
                      np.flip(II),np.flip(I3[1:])))
    P3_mat = np.column_stack((P3_X,P3_Y))

    #P4
    P4_X = np.concatenate((IV_range,VII_range3,[H[0]],
                      np.flip(III_range),[T[0]],np.flip(I_range2)))
    P4_Y = np.concatenate((IV,VII3,[H[1]],np.flip(III),[T[1]],np.flip(I2)))
    P4_mat = np.column_stack((P4_X,P4_Y))

    #P5
    P5_X = np.concatenate((V_range,VII_range2,[J[0]],
                      np.flip(IV_range),[K[0]],np.flip(I_range1)))
    P5_Y = np.concatenate((V,VII2,[J[1]],np.flip(IV),[K[1]],np.flip(I1)))
    P5_mat = np.column_stack((P5_X,P5_Y))

    #P6
    P6_X = np.concatenate((VI_range,[O[0]],VII_range1,[L[0]],np.flip(V_range)))
    P6_Y = np.concatenate((VI,[O[1]],VII1,[L[1]],np.flip(V)))
    P6_mat = np.column_stack((P6_X,P6_Y))

    #P7
    P7_X = np.concatenate(([Q[0],O[0]],np.flip(VI_range)))
    P7_Y = np.concatenate(([Q[1],O[1]],np.flip(VI)))
    P7_mat = np.column_stack((P7_X,P7_Y))

    #P8
    P8_X = np.concatenate(([S[0]],np.flip(VIII_range),
                      np.flip(VII_range2),np.flip(VII_range1)))
    P8_Y = np.concatenate(([S[1]],np.flip(VIII),np.flip(VII2),np.flip(VII1)))
    P8_mat = np.column_stack((P8_X,P8_Y))

    #P9
    P9_X = np.concatenate(([S[0],R[0],G[0]],np.flip(VII_range4),
                      np.flip(VII_range3),VIII_range[1:]))
    P9_Y = np.concatenate(([S[1],R[1],G[1]],np.flip(VII4),np.flip(VII3),VIII[1:]))
    P9_mat = np.column_stack((P9_X,P9_Y))

    """
    Plot Polygons
    """
    fig = plt.figure(figsize=(5,8))
    ax = fig.add_subplot(111)
```

```

  #add polygons to plot
  ax.add_patch(des.PolygonPatch(geom.Polygon(P1_mat),
          facecolor=[0,0,0,0.2], label='1. Sensitive fine-grained'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P2_mat),
          facecolor=[0,0,0,0.8], label = '2. Organic'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P3_mat),
          facecolor=[0,0,0,0.7], label= '3. Clay'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P4_mat),
          facecolor=[0.3,0.4,0.2,1], label='4. Silt-mixtures'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P5_mat),
          facecolor=[0.6,0.6,0.2,1], label='5. Sand-mixtures'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P6_mat),
          facecolor=[0.7,0.4,0,1], label='6. Sand'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P7_mat),
          facecolor=[0.7,0.2,0,1], label='7. Gravelly sand to sand'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P8_mat),
          facecolor=[0.1,0.4,0.5,1], label='8. Very stiff sand to clayey sand'))
  ax.add_patch(des.PolygonPatch(geom.Polygon(P9_mat),
          facecolor=[0.1,0.4,0.5,0.8], label='9. Very stiff fine-grained'))

  #plot specifics
  ax.set_xlim(A[0],D[0])
  ax.set_ylim(A[1],Q[1])
  #ax.set_xlim(-2.3026,2.3026)
  leg = ax.legend(bbox_to_anchor=(1.01,1),title='Legend', title_fontsize=15)
  leg._legend_box.align = "left"
  ax.text(-1.5,1,'1',fontsize=15)
  ax.text(1.8,0.5,'2',fontsize=15)
  ax.text(1.2,1.7,'3',fontsize=15)
  ax.text(0.6,2.6,'4',fontsize=15)
  ax.text(0,3.3,'5',fontsize=15)
  ax.text(-0.8,4.5,'6',fontsize=15)
  ax.text(-1.5,6,'7',fontsize=15)
  ax.text(1.05,6,'8',fontsize=15)
  ax.text(1.9,5,'9',fontsize=15)
  ax.set_title('normalized Robertson chart', fontsize=20)
  ax.set_xlabel(r'ln$(nF_r)$', fontsize=15)
  ax.set_ylabel(r'ln$(nQ_t)$', fontsize=15)

  Polygons = np.array([geom.Polygon(P1_mat),geom.Polygon(P2_mat),geom.Polygon(P3_mat),
                         geom.Polygon(P4_mat),geom.Polygon(P5_mat),geom.Polygon(P6_mat),
                         geom.Polygon(P7_mat),geom.Polygon(P8_mat),
                        geom.Polygon(P9_mat)])

    ax.plot(matrix_generalized[:,2], matrix_generalized[:,1],
                              marker='o',ls='',color='k',markersize=2)
    return Polygons

def MC_probability(matrix_generalized, Polygons, params):
    “””function that approximates joint gaussian distribution via a monte carlo
    analysis and calculates the probabilities for each datapoint
    “””

    std_Fr = params["std Fr"]
    std_Qt = params["std Qt"]
    iterations = params["probability iterations"]
    for jj in range(len(matrix_generalized)):
        data = (matrix_generalized[jj,2],matrix_generalized[jj,1])

        std_Fr = 1
        std_Qt = 1.2
        print(jj)
        count = np.array([0,0,0,0,0,0,0,0,0])
        for ii in range(0,iterations):

            Fr =random.gauss(data[0],std_Fr)
            Qt =random.gauss(data[1],std_Qt)
            point = geom.Point(Fr,Qt)
            if point.within(Polygons[0]) == True:
                count[0] += 1
            if point.within(Polygons[1]) == True:
                count[1] += 1
            if point.within(Polygons[2]) == True:
                count[2] += 1
            if point.within(Polygons[3]) == True:
                count[3] += 1
            if point.within(Polygons[4]) == True:
                count[4] += 1
            if point.within(Polygons[5]) == True:
                count[5] += 1
            if point.within(Polygons[6]) == True:
                count[6] += 1
            if point.within(Polygons[7]) == True:
                count[7] += 1
            if point.within(Polygons[8]) == True:
                count[8] += 1
        matrix_generalized[jj,3:] = count/np.sum(count)

    return matrix_generalized

```

Appendix D: Posterior.py

```python
import numpy as np
import random
import scipy.optimize as optimize
import math
import warnings

#a warning is raised when -inf values are found. The code uses a workaround. For clarity,
#this warning is excluded
warnings.filterwarnings("ignore", category=RuntimeWarning)

def find_nearest(array,value):
    """
    function to calculate index in array closest to the value given
    """
    idx=[]
    for ii in value:
        idx.append(np.abs(array-ii).argmin())
    return idx


def thickness_to_boundaries(thickness,matrix_generalized):
    """
    Function to get the indices of the boundaries (convert array with thicknesses
    information to array with depth (boundary) information)
    """
    boundaries = np.zeros(len(thickness)+1)

    boundaries[0] = matrix_generalized[0,0]   #first boundary = top of soil profile
    for ii in range(len(boundaries)-1):
        boundaries[ii+1] = matrix_generalized[0,0] + sum(thickness[:ii+1])

    """ 2.1 get indices in data matrix"""
    soil_boundaries = np.arange(0,len(thickness)+1)
    indices = find_nearest(matrix_generalized[:,0],boundaries[soil_boundaries])
    indices=np.array(indices) #index positions of boundaries in data matrix.

    return indices

def Eqs_Wang(matrix_generalized,indices):
    """
    function that follows Eq. 2 and 1 given by Wang et al. (2013). calculates, for each
    layer, the chance that that layer belongs fully to 1 single soil type
    """
    P_ST_J=np.zeros((len(indices)-1,9))    #matrix with results following eq. 2
    P_eps_n=np.zeros(len(indices)-1).     #matrix with results following from eq. 1

    """ 2.2 eq. 2 & 1 Wang et al. (2013) """

    #the subdivision between upper and lower bound has no coding benefit other than being
    #more clear to the reader.
    upper_bound = indices[0:-1] #upper bounds of soil layers
    lower_bound = indices[1:] #lower bounds of soil layers

    for nn in range(0,len(indices)-1): #iterates for each soil layer in the model class
        #for soil type 1, Eq. 2
        P_ST_J[nn,0]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],3]))

        #for soil type 2, Eq. 2
        P_ST_J[nn,1]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],4]))

        #for soil type 3, Eq. 2
        P_ST_J[nn,2]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],5]))

        #for soil type 4, Eq. 2
        P_ST_J[nn,3]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],6]))

        #for soil type 5, Eq. 2
        P_ST_J[nn,4]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],7]))

        #for soil type 6, Eq. 2
        P_ST_J[nn,5]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],8]))

        #for soil type 7, Eq. 2
        P_ST_J[nn,6]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],9]))

        #for soil type 8, Eq. 2
        P_ST_J[nn,7]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],10]))

        #for soil type 9, Eq. 2
        P_ST_J[nn,8]= np.sum(
                      np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],11]))

        #Eq. 1
        P_eps_n[nn] = np.sum(np.exp(P_ST_J[nn]))

    return P_eps_n

def soil_type(matrix_generalized, indices):
    """
    calculates most probable soiltype (1-9) per layer based on the given boundaries
    """
    P_ST_J=np.zeros((len(indices)-1,9))   #matrix with results following from eq. 2

    soiltype_final = np.zeros((len(indices)-1))

    """ 2.2 eq. 2 & 1 Wang et al. (2013) """

    #the subdivision between upper and lower bound has no coding benefit other than
    #being more clear to the reader.
    upper_bound = indices[0:-1] #upper bounds of soil layers
    lower_bound = indices[1:] #lower bounds of soil layers

    for nn in range(0,len(indices)-1): #iterates for each soil layer in the model class
        #for soil type 1, Eq. 2
        P_ST_J[nn,0]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],3]))

        #for soil type 2, Eq. 2
        P_ST_J[nn,1]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],4]))

        #for soil type 3, Eq. 2
        P_ST_J[nn,2]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],5]))

        #for soil type 4, Eq. 2
        P_ST_J[nn,3]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],6]))

        #for soil type 5, Eq. 2
        P_ST_J[nn,4]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],7]))

        #for soil type 6, Eq. 2
        P_ST_J[nn,5]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],8]))

        #for soil type 7, Eq. 2
        P_ST_J[nn,6]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],9]))

        #for soil type 8, Eq. 2
        P_ST_J[nn,7]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],10]))

        #for soil type 9, Eq. 2
        P_ST_J[nn,8]= np.sum(
                    np.log(matrix_generalized[upper_bound[nn]:lower_bound[nn],11]))

        soiltype_final[nn] = np.argmax(P_ST_J[nn,:]) +1
    return soiltype_final

def objective_function(thickness,matrix_generalized):
    """
    calculates objective function = -ln(likelihood function)
    """
    #get indices of the layer boundaries in the matrix
    indices = thickness_to_boundaries(thickness,matrix_generalized)
    """ 2.2 eq. 2 & 1 Wang et al. (2013) """
    P_eps_n = Eqs_Wang(matrix_generalized,indices)
    """ 2.3 eq. 6 Wang et al. (2013) """
    P_eps = np.sum(np.log(P_eps_n))
    return -P_eps

def best_guesses(matrix_generalized,model_class,params):
    """
    Function that calculates a number (=no_guesses) of best guesses. that is, for layer
    thickness configurations that give the lowest objective functions
    """
    min_thickness= params["min thickness"]
    boundary_constraint = params["boundary constraint"]
    no_guesses = params["no. guesses"]
    iterations = params["guesses iterations"]


    #1st column = posterior value, others = thicknesses
    best_conf = np.zeros((no_guesses,model_class+1))
    for i in range(iterations):
        #empty list in which boundaries will be stored -->
        #[start_depth,boundary_1,boundary_2,...,end_depth]
        boundaries = np.zeros(model_class+1)

        """ 0. make array of possible boundaries, spacing of 0.1 """
        boundary_possibilities = np.arange(round(matrix_generalized[0,0],1)+0.1*
                                       (round(matrix_generalized[0,0],1)
                                       <matrix_generalized[0,0])+min_thickness,
                                        round(matrix_generalized[-1,0],1)-
                                       0.1*(round(matrix_generalized[-1,0],1)
                                       >matrix_generalized[-1,0])+0.1,0.1)

        """ 1. get thickness distribution """
        #1.1 first boundary is start of profile (depth at which cpt starts)
        boundaries[0] = matrix_generalized[0,0]

        #1.2 generate random values within constraint
        L_constraints=np.array([])
        for constraint in boundary_constraint:
            if model_class >= constraint[-1]:
                random_constraint = round(np.random.uniform(constraint[0],
                                       constraint[1]),1)
                L_constraints=np.append(L_constraints,random_constraint)

                #indices of value +- min_thickness in boundary_possibilities
                idx = np.array(np.where((boundary_possibilities
                             >random_constraint-min_thickness)
                             *(boundary_possibilities
                             <random_constraint+min_thickness))[0])

                #delete indices so that next iteration only a correct boundary
                boundary_possibilities=np.delete(boundary_possibilities,idx)

        #1.3 add constraints to boundaries
        boundaries[1:len(L_constraints)+1] = L_constraints

        #1.4 give random values to other boundaries (but not for first modelclass, as
        #then there is only 1 layer)
        if model_class>1:
            for nn in range(1+len(L_constraints),model_class+1):
                boundaries[nn] = random.choice(boundary_possibilities)

                #indices of value +- min_thickness in boundary_possibilities
                idx = np.array(np.where((boundary_possibilities
                              >boundaries[nn]-min_thickness)
                              *(boundary_possibilities
                              <boundaries[nn]+min_thickness))[0])

                #delete indices so that next iteration only a correct boundary
                boundary_possibilities=np.delete(boundary_possibilities,idx)

        #1.5 last boundary will be at final depth of CPT
        boundaries[model_class] = matrix_generalized[-1,0]

        boundaries=np.sort(boundaries)      #sort boundary list.
        thickness =np.diff(boundaries)

        """ 2. get likelyhood function """
        P_eps = objective_function(thickness,matrix_generalized)

        """ 3. store best ones in matrix """
        if i < no_guesses:              #first fill guess matrix (which is now all 0's))
            best_conf[i,0] = P_eps
            best_conf[i,1:]=thickness

        if i >= no_guesses: #update guess matrix if a more preferred value is found
            pos = np.where(best_conf[:,0] == max(best_conf[:,0]))[0][0]
            if best_conf[pos,0] > P_eps:
                best_conf[pos,0] = P_eps
                best_conf[pos,1:] = thickness

    return best_conf


def optimizer(matrix_generalized,model_class, params,best_conf):
    """
    Function that further optimizes a layer thickness configuration based on the basin-
    hopping approach
    """
    min_thickness = params["min thickness"]      #minimal thickness to consider
    thickness_tot = round(matrix_generalized[-1,0]-matrix_generalized[0,0],2)
    cons = [] #list summarizing the constraints that will be used by the optimizer

    #constraint 1 = sum of thicknesses must be equal to total thickness
    cons.append({'type':'eq','fun': lambda thickness: np.sum(thickness)-thickness_tot})

    #constraint 2 = thickness must be larger or equal to the minimal thickness
    for con in range(model_class):
        cons.append({'type':'ineq','fun': lambda thickness: thickness[con]-
                                              min_thickness})

    bounds = [] #list summarizing the bounds of the thickness

    #bound 1= thickness per layer must be minimal of 0.2 and maximum of total thickness
    for bound in range(model_class):
        bounds.append([0.2,thickness_tot])

    minimizer_kwargs = dict(method='SLSQP',args=(matrix_generalized), constraints=cons,
                              bounds=bounds, tol=0.01, options=dict(maxiter=200))

    #an array where the minimal value of the objective function will be stored. start
    #with infinity
    min_objective = np.array([np.inf])
    thicknessL = np.zeros((1,model_class))

    for opt in range(len(best_conf)): #optimize for all best configurations
        initial_guess = best_conf[opt,1:] #specify initial guess for optimizer

        counter = 0 #start a counter
        while counter == 0: #as long as counter = 0, keep trying to optimize
            res_loop = optimize.basinhopping(objective_function,initial_guess,
                    minimizer_kwargs=minimizer_kwargs,niter=500, niter_success=100)

            #if optimization is found, add counter so optimizing for that configuration
            #will stop
            if res_loop.lowest_optimization_result.success == True:
                counter += 1

                #if the new optimized value is lower than the one stored in
                #"objective", change old value to this value
                if res_loop.fun < min_objective[0]:
                    #store current optimal objective function
                    min_objective[0] = res_loop.fun

                    #store thicknesses for that optimal value
                    thicknessL[:] = np.array(res_loop.x)[:]

            #if optimization failed, try again
            if res_loop.lowest_optimization_result.success == False:
                print('False')

    #return minimized objective function and corresponding thicknesses
    return min_objective, thicknessL

def calculator(matrix_generalized,params):
    """
    main function solving for the different subparts
    """

    N_min = params["N min"]
    N_max = params["N max"]
    treshold = params["treshold model class"]
    std_Fr = params["std Fr"]           #standard deviation of friction ratio
    std_Qt = params["std Qt"]           #standard deviation of cone resistance
    k = np.arange(N_min,N_max+1)

    """1a. if a probability = 0, this will cause problems --> change all 0's to a small
    Number
    """
    thickness_tot = matrix_generalized[-1,0]-matrix_generalized[0,0]
    zero_position = np.argwhere(matrix_generalized[:,3:]==0)
    for index in zero_position:
        matrix_generalized[index[0],index[1]+3]=0.00001

    #list with best layer thickness configuration per model class
    all_optimized = np.zeros((int(N_max),int(N_max)+1))
    K_N_L = np.zeros((1,N_max))

    """2. minimize objective function (=-ln(likelihood function)) per model class"""
    """2a. Model class 1: having 1 layer, so only 1 solution"""
    N1 = objective_function([thickness_tot],matrix_generalized)
    all_optimized[0,0:2]=[N1,thickness_tot]
    print("Caclulation for Model Class 1: SUCCES")

    """2b. calculate for other model classes"""
    for N in range(N_min+1,N_max+1):

        """first, get best number (=no_guesses) of best guesses"""
        best_conf= best_guesses(matrix_generalized,N,params)

        """if N is lower than specified number of layers (=threshold), no optimization
        is needed--> take lowest of best guesses"""
        if N<=treshold:
            min_index = np.argmin(best_conf[:,0])

            all_optimized[N-1,:N+1]=best_conf[min_index]

        """if N is larger than specified number of layers (=threshold), optimization is
        needed."""

        if N>treshold:
            #optimizes for each guess and returns most optimal optimized value
            objective, thicknessL = optimizer(matrix_generalized,N,params,best_conf)
            all_optimized[N-1,0]=objective
            all_optimized[N-1,1:N+1]=thicknessL
        print(f"Calculation for Model Class: {N}: SUCCES")

    """PART 3: calculate most probable model class"""
    likelihood = -all_optimized[:,0]
    prior_distribution = np.log((1/std_Fr * 1/std_Qt)**k)

    #calculate conditional probability (Eq. 12, Wang et al. (2013))
    cond_prob = likelihood+prior_distribution
    cond_prob = np.exp(cond_prob)
    cond_prob = np.log(cond_prob/(thickness_tot**(k-1)))

    #maximize posterior function and select layer and thicknesses
    max_index = np.argmax(cond_prob)
    final_class = all_optimized[max_index,:max_index+2]

    """PART 4: get soil types for that model class"""
    #get soil types for final class
    thickness_final = final_class[1:]
    indices = thickness_to_boundaries(thickness_final,matrix_generalized)
    soiltype_final = soil_type(matrix_generalized,indices)


    """PART 5: save results"""
    #get summary
    cond_prob = np.reshape(cond_prob,(len(cond_prob),1))
    summary = np.hstack((all_optimized,cond_prob))

    #get final configuration
    final_result = np.column_stack([thickness_final, soiltype_final])

    return final_result, summary

```