UNIVERSITY OF BELGRADE

FACULTY OF CIVIL ENGINEERING

Miloš S. Marjanović

Analysis of Interaction Inside the Pile Group Subjected to Arbitrary Horizontal Loading

Doctoral Dissertation

Belgrade, 2020.

UNIVERZITET U BEOGRADU

GRAĐEVINSKI FAKULTET

Miloš S. Marjanović

Analiza interakcije šipova u grupi opterećenoj horizontalnim opterećenjem proizvoljnog pravca

Doktorska disertacija

Beograd, 2020.

Miloš S. Marjanović

Analysis of Interaction Inside the Pile Group Subjected to Arbitrary Horizontal Loading

Advisors:

Prof. Dr. Mirjana Vukićević
University of Belgrade, Faculty of Civil Engineering

Dr.-Ing. Diethard König (AkadOR)
Ruhr Universität Bochum, Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics

Committee:

Prof. Dr. Mirjana Vukićević
University of Belgrade, Faculty of Civil Engineering

Dr.-Ing. Diethard König
Ruhr Universität Bochum, Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics

Assist. Prof. Dr. Sanja Jocković
University of Belgrade, Faculty of Civil Engineering

Assist. Prof. Dr. Selimir Lelović
University of Belgrade, Faculty of Civil Engineering

Assoc. Prof. Dr. Petar Santrač
University of Novi Sad, Faculty of Civil Engineering Subotica

Belgrade, ____. ____. 2020.

Acknowledgments

This thesis is the result of the research carried mainly at the Chair of Geotechnical Engineering of University of Belgrade and partially at the Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics of Ruhr University Bochum (RUB) during 2015-2020.

First and foremost, I would like to thank to my thesis advisor, Prof. Dr. Mirjana Vukićević from University of Belgrade, for giving me the opportunity to start the research career in the field of Geotechnical Engineering, for her continuous support during the development of the thesis, valuable discussions and the scientific freedom that allowed me to grow as a researcher.

I express my sincerest gratitude to thesis coadvisor, Dr. Diethard König from RUB for introducing me into the field of horizontally loaded pile groups, for constant enthusiasm, motivation and important comments during my research. Beside his academic support, Dr. König helped me with his warm welcomes during my research stays in Bochum, provided and arranged all necessary resources during my work, removed every obstacle that he could, and made me think only about the research. He really made my dream come true.

My forever gratitude goes to Prof. Dr. Tom Schanz, late Head of the Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics at RUB, who suddenly passed away in October 2017. His scientific achievements, dedication, outstanding personality and leadership were true inspiration for everyone of us who had the privilege to be the part of his research group. His memory will always be with me as an example of what does it really mean to be a modern scientist.

Initial discussions with Prof. Dr. Rene Schäfer from Hochschule Ruhr West are acknowledged. I also thank to all Committee members for review of the thesis and the valuable comments.

My special thanks go to Prof. Dr. Mira Petronijević from the University of Belgrade, for her dedication to young researchers, involvement and coordination of the SEEFORM scholarship project, that made my visits to the RUB possible.

I am grateful to all of my colleagues at the Chair of Geotechnical Engineering for valuable discussions and continuous support during my research: Prof. Dr. Slobodan Ćorić, Dr. Selimir Lelović, Uroš Djurić, Dr. Zoran Radić, Dr. Dejan Divac and Slobodan D. Radovanović. Special thanks go to my office 160-162 colleagues, for making the everyday work great: Dr. Sanja Jocković, Dr. Snežana Maraš-Dragojević, Veljko Pujević, Nikola Obradović and Dobrivoje Simić.

I am also very grateful to many colleagues from RUB, who welcomed me friendly and made my research stays in Bochum unforgettable and exciting experience inside and outside the office: Dr. Jelena Ninić, Dr. David Oswald, Dr. Timo Kasper, Christoph Schmüdderich, Thomas Barciaga, Dr. Nina Müthing, Achim von Blumenthal, Wolfgang Lieske, Florian Christ, Max Schoen, Dr. Linzhi Lang, Dr. Elham Mahmoudi, Raoul Hölter, Dr. Kavan Khaledi, Dr. Abhishek Rawat, Debdeep Sarkar, Peyman Mianji, Alireza Jafari Jebeli, Lingyun Li, Dr. Gaziz Seidalinov, Dr. Usama Al-Anbaki, Manh Cuong Le, Dr. Wiebke Baille, Dr. Negar Rahemi, Dr. Anina Glumac, Prof. Dr. Tamara Nestorović, Vladislav Gudžulić, Ana Cvetković, Mirjana Ratkovac, Sahir Butt, Armel Meda, Reinhard Mosinski and Doris Traas. Special thanks go to my office colleagues: Dr. Arash Lavasan, Dr. Chenyang Zhao, Dr. Meisam Goudarzy and Qian Gu.

My thanks are also addressed to colleagues from the SEEFORM scholarship program, for the great moments during our meetings, seminars and research stays as well: Dr. Marko Radišić, Miloš Jočković, Dragan Kovačević, Dr. Marko Marinković and Nemanja Marković.

The writing of this thesis took a lot of time that could have been spent with my family. I am very grateful to my wife Andrea for her love, endless patience and incredible support during my PhD studies and long absences from our home. I would never complete my work without her understanding, positive thinking and encouragement in many hard moments, when things looked impossible. *Thank you for always being my better half.*

I am also very grateful to my brother, Dr. Miroslav Marjanović, who walked the same path before me, helped me with his useful comments and did the final proof read of this thesis. He was the "$in - situ$" and best possible example of what one PhD student should look like. The support and true love from him and his wife Dragana were of the great importance in this endeavour. Special thanks go to Strahinja, our star, for the laugh, fun, happiness and true purpose that he brought in our lives.

Last but not least, I am grateful to my beloved parents Slobodan and Svetlana, for showing me the true values of life, and their endless love.

Belgrade, July 2020.

Miloš Marjanović

”...When something is worth doing, it’s worth overdoing...” (Dr. Brian May)

Financial Support

The work on this thesis was supported by the Government of the Republic of Serbia - Ministry of Education, Science and Technological Development, under the Project **TR-36046** (project leader Prof. Dr. Mira Petronijević).

The financial support has been also provided through the **SEEFORM** (South Eastern European Graduate School for Master and Ph.D. Formation in Engineering) project financed by German Academic Exchange Service - DAAD (project leader Prof. Dr. Rüdiger Höffer).

The significant financial support for research stays in 2018-2019 was provided both by the Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics of Ruhr University in Bochum (RUB) and through the Collaborative Research Center **SFB 837 "Interactions Modelling in Mechanized Tunneling"** (project leader Prof. Dr. Günther Meschke).

The network licence of the PLAXIS 3D® software was provided by the Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics of Ruhr University Bochum and the PLAXIS BV Delft company.

To the Memory of
Professor Tom Schanz

Analysis of Interaction Inside the Pile Group Subjected to Arbitrary Horizontal Loading

Abstract

The analysis of pile groups subjected to horizontal loading in one of two orthogonal directions is a common problem in geotechnical engineering. However, the above analysis should sometimes be extended with the additional cases of horizontal loading, in arbitrary direction. Although the full scale experiments can provide the best insight into the above problem, they are expensive and therefore not feasible solution. Instead, numerical (i.e. finite element) analysis usually remains the ultimate tool for large scope studies.

The main objective of this research is the improvement of the analysis methodology of pile group for the case of arbitrary static horizontal loading. Its influence is investigated numerically to check for the existence of the "critical" pile group configurations and soil conditions, that may lead to the failure of the foundation structure and superstructure itself.

To reach the above objective, apriori sensitivity analysis of the considered problem and identification of the main problem parameters are conducted in the first phase. After that, series of complex 3D numerical models for laterally loaded pile group analysis have been generated using PLAXIS 3D. Model validation is done by the back-calculation of available experimental results. Parametric study of pile groups with various configurations under arbitrary horizontal loading is performed, with an emphasis on pile force distribution, bending response and pile group efficiency.

Modelling and simulation processes and the optimization of the calculation time were improved using the originally developed codes in Python. The use of multiple computers was allowed by using the author's scripts developed within the thesis. The above research resulted in thousands of numerical simulations, whose findings provided the improved design methodology of pile groups under arbitrary static horizontal loading.

Keywords: pile group, horizontal static loading, PLAXIS 3D, Hardening Soil model, Python

Scientific Field: Civil Engineering

Scientifis Subfield: Geotechnical Engineering

UDC: 624.13(043.3)

Analiza interakcije šipova u grupi opterećenoj horizontalnim opterećenjem proizvoljnog pravca

Rezime

Analiza šipova u grupi izloženoj horizontalnom opterećenju u nekom od dva ortogonalna pravca je uobičajen problem u geotehnici. Medjutim, navedena analiza ponekad se mora proširiti dodatnim slučajevima horizontalnog opterećenja, u proizvoljnom pravcu. Iako eksperimenti na realnim konstrukcijama mogu pružiti najbolji uvid u navedeni problem, oni predstavljaju skupo (i samim tim neizvodljivo) rešenje. Umesto toga, numerička analiza (npr. primenom metoda konačnih elemenata) obično postaje primarni alat za studije velikog obima.

Glavni cilj ovog istraživanja je poboljšanje metodologije za analizu grupe šipova za slučaj proizvoljnog statičkog horizontalnog opterećenja. Njegov uticaj je razmatran numerički kako bi se utvrdilo postojanje "kritične" konfiguracije šipova u grupi i uslova tla, koja može dovesti do loma temeljne konstrukcije i samog objekta.

Radi ostvarenja gore navedenog cilja, u prvoj fazi sprovedena je uvodna analiza osetljivosti razmatranog problema i identifikacija glavnih parametara problema. Nakon toga, generisane su serije složenih 3D numeričkih modela za analizu bočno opterećenih grupa šipova u PLAXIS 3D. Validacija modela izvršena je povratnim proračunom na osnovu dostupnih eksperimentalnih rezultata. Sprovedena je parametarska studija bočno opterećenih grupa šipova različitih konfiguracija, sa akcentom na raspodelu sila u šipu, odgovor šipa pri savijanju i efikasnost grupe šipova.

Postupci modeliranja i simulacije, kao i optimizacija vremena proračuna, poboljšani su primenom originalnih programa u Python-u. Upotreba više računara za proračun omogućena je primenom autorovih programa razvijenih u okviru ove teze. Navedeno istraživanje rezultovalo je u hiljadama numeričkih simulacija, čiji su rezultati doveli do poboljšanja metodologije proračuna šipova u grupi usled proizvoljnog statičkog horizontalnog opterećenja.

Ključne reči: grupa šipova, horizontalno statičko opterećenje, PLAXIS 3D, Hardening Soil model, Python

Naučna oblast: Gradjevinarstvo

Uža naučna oblast: Gradjevinska geotehnika

UDK: 624.13(043.3)

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

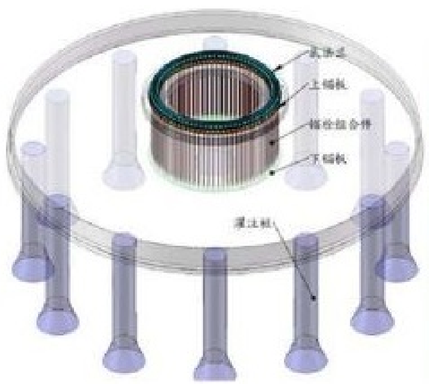
Increasing world population has led to the construction of the large number of high buildings in the last decades [1]. For the purpose of economic foundation of structures, the application of shallow foundations is usually the first option, with the use of minimum required foundation depth. However, in situations where the surface soil layers have low bearing capacity, the use of shallow foundations is not possible. Instead, the deep foundation systems are used, where the load is transmitted to the deeper soil layers with higher bearing capacity using special structural elements, mostly piles. Single piles are used in a very few cases. Instead, piled foundations are usually designed as closely spaced pile groups. Joint work of the piles is enabled by constructing the rigid pile cap that connects the pile tops and provides the load transfer from the structure. Pile cap is usually in the contact with the ground, or above the ground level (in offshore structures). The pile group configuration is mostly squared or rectangular.

Pile group under horizontal loading

Beside the primary function to transfer vertical loads to a deeper soil layers, pile foundations can be significantly loaded with horizontal loads. The examples of horizontally loaded structures are high rise ones (industrial chimneys, buildings, wind turbines), harbour structures, bridges, retaining structures, offshore platforms, among others. Horizontal loads on these structures mostly originate from:

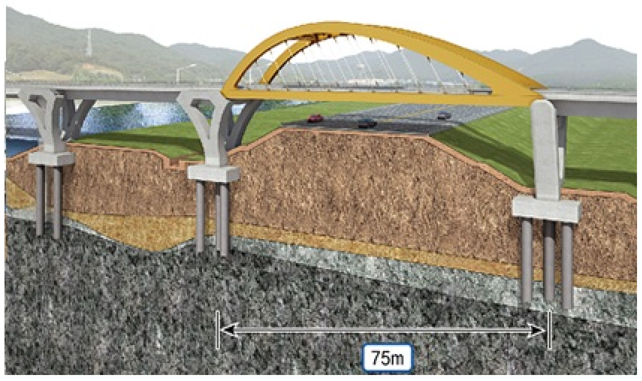
- Wind pressure,
- Earthquake,
- Wave, current and ice action,
- Impact of ships and other vessels,
- Traffic acceleration, braking and turning forces,
- Earth pressures,

- Soil displacements due to landslides or liquefaction,
- Differences in excavation levels around the foundation structures, etc.



(a) Wind turbine foundations

(b) Harbour structure foundations



(c) Bridge foundations

Figure 1.1: Examples of laterally loaded pile foundations (sources: CNBM, China, Encyclopedia Britannica and ESCO Consultant and Engineers Co., Ltd., South Korea)

The magnitude of lateral load is usually 10-15% of vertical load, and about 30% in offshore structures [2]. When the horizontal load on the pile head is caused by the superstructure, such piles are generally referred as "active piles". On the contrary, when the piles are loaded due to the horizontal soil movements (for example, in the case of pile stabilized landslides), these piles are referred as "passive piles".

The problems that can arise due to the inappropriate assessment of horizontal loadings in piles can be very large. Failure in these cases can lead to a severe damage or even collapse of the entire structure.

"Shadowing" effect

The problem of interaction between the piles inside the laterally loaded pile group has been recognized by scientific community. It is, in general, considered as more complex than the problem of axially loaded piles. When the pile group is laterally loaded, the stress-strain fields of neighbouring piles overlap. The soil in front of the piles becomes stiffer, while the soil behind the piles is consequently softened. Also, a separation ("gapping") between the piles and soil at the back of the pile occurs. This influence of the leading pile rows to the lateral response of trailing pile rows is called *"shadowing"*, because the trailing rows are in the shadow of the front pile row (as shown in Fig. 1.2).

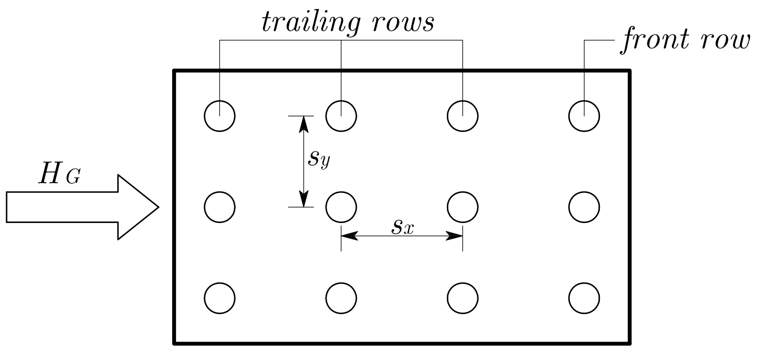


Figure 1.2: Front and trailing rows in laterally loaded pile group. H_G - horizontal force, s_x and s_y - center-to-center spacings between the piles in X and Y directions, respectively

Ideally, the piles in the group should be spaced so that the load-bearing capacity of the group is no less than the sum of the bearing capacity of the individual piles. However, due to the soil-structure interaction effects, the load-displacement behaviour of a pile inside the group is different than of the equivalent single pile. Therefore, unequal load distribution occurs inside the pile group (see Figure 1.3).

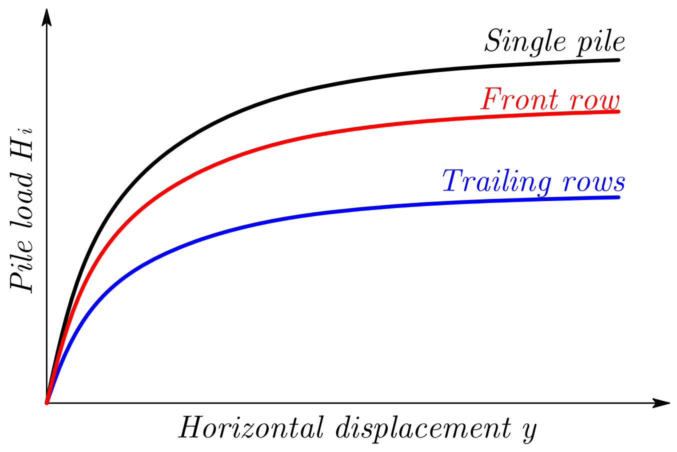


Figure 1.3: Influence of the "shadowing" on the force distribution inside the pile group (H_i denotes the horizontal force on pile i)

The maximum bending moment in a group will also be larger than that for an equivalent single pile, because the soil behaves as it has less resistance, allowing the group to deflect more for the same load per pile. When the pile spacing is increased, the effects of overlapping between the resisting zones are less significant.

Design requirements

Every foundation structure must provide the stability of the structure under all patterns and combinations of loads (ultimate limit state, *ULS*). Also, total and differential settlements, lateral displacements and rotations must remain within acceptable limits (serviceability limit state, *SLS*). While ULS analysis eliminates the threat to structures and human life, SLS analysis ensures long-term usability.

Traditional approach in the design of pile foundations, followed by the current design codes, is based on the capacity-based design, while the estimation of the deformations is treated as the secondary issue. However, although the capacity-based design is still widely used in engineering practice, many authors pointed out that the opposite, displacement-based design approach, is more appropriate in a variety of practical applications [1, 3–9].

There are two important factors for the design of laterally loaded pile foundations:

- maximum lateral displacement at the pile top,
- maximum bending moment in the pile.

In the cases of bridges or other structures founded on pile foundations, only a few centimeters of lateral displacements can cause significant stress development in these (statically indeterminant) structures [10]. According to Eurocode 7 [11], pile horizontal displacement is usually limited to 3% of a pile diameter, or max 2 cm.

1.2 Arbitrary lateral loading direction

Despite many experimental and numerical research done in the field of laterally loaded pile groups in last decades (presented in **Chapter 2**), some important questions are still unresolved.

The most of the (so-far conducted) laterally loaded pile group studies have considered rectangular pile groups, assuming that horizontal loading acts along the one of the two orthogonal directions, parallel to the edges of the group. However, the horizontal loading on the pile group may have arbitrary direction, mainly because of the stochastic nature of its source (such as wind, earthquake, wave loading, ship impact etc.). When the direction of the loading is unknown, the worst case scenario is the loading direction that results in minimum foundation capacity.

The number of the research papers dealing with the pile groups under arbitrary lateral loading is very limited. Ochoa and O'Neill [12] developed the interaction factors based on the experimental results of the free and pinned-head piles in the submerged sand. These interaction factors are dependent on the angle between the load vector and the line that connects the pile heads. Randolph [13] defined the influence of the loading direction for two laterally loaded piles. Fan and Long [14] analyzed the interaction between the piles under various loading directions, and derived the modulus reduction factors to account for the pile group interaction effects at the ultimate limit state. In this study, a procedure for determination of the individual pile response inside the pile group with arbitrary geometric layout was proposed. Su and Yan [15] formulated a multidirectional p-y model for sands that was incorporated into FEM and validated through the simulations of piles under unidirectional and multidirectional lateral loading. Mayoral et al. [16] formulated the p-y curves for piles under multidirectional loading in soft clays, emphasizing the complexity of the soil-pile interaction under multidirectional loading. The simple application of the common p-y curves in two orthogonal direction was found to be impossible. Some more complex cases of the pile-soil-pile interactions, such as the behaviour of the pile group under either eccentric lateral [17] or torsional loading [18], have also been reported. The influence of the arbitrary lateral loading on the ultimate resistance of the pile group with 2 rigid piles was analyzed by Georgiadis et al. [19], through finite element method (FEM) analysis.

Comments on the study of Su and Zhou 2015.

In their recent paper, Su and Zhou [20] presented the experimental study of laterally loaded pile groups under different horizontal loading directions. This study included square and rectangular 2x2 pile groups in sand. Individual (equivalent) single piles were also tested in order to evaluate the pile interaction effects (i.e. pile group efficiency and pile interaction factors). Model pile groups with different pile spacings were subjected to the static lateral loading. The load was applied by hanging the weights connected to the loading point with the steel rope. Only the values at the ultimate displacement level of $0.2D$ were presented, despite the fact that the pile group interaction is strongly influenced by the displacement level [21].

The results have shown that the loading direction has great influence on the redistribution of loads between the piles inside the group, as well as on the total lateral bearing capacity of the pile group. The pile group at medium pile spacing is more sensitive to the changes in loading direction. This is probably due to the fact that the variation of the overlapping zone is higher at medium spacings.

For some pile groups, the increase of the total load proportion for some piles was observed for the loading direction that was different than the usual 0/90° case, which leads to the assumption that additional critical loading cases exist, associated with new critical loading direction. The total load proportion in some individual piles may be underestimated, which can lead to the design on the unsafe side. For the 2x2 pile group, the increase was found to be from 25% to 35%, which is the relative increase of the pile force of about 40%. At the small deflection level, load proportions are almost equal for all piles.

The conclusions from this research, as well as the fact that, up to the author's knowledge, there are almost no research papers considering such loading case, have mainly motivated the further research presented within this Dissertation.

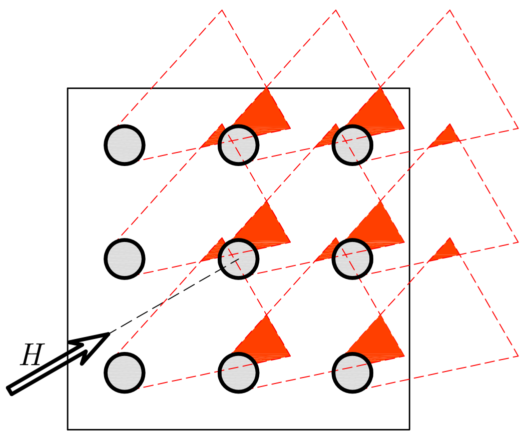


Figure 1.4: Pile group under arbitrary horizontal loading (shaded areas show the overlapping of the shear zones)

1.3 Research objectives and assumptions

Objectives

The main objective of this research is the improvement of the analysis of pile groups for the case of arbitrary static horizontal loading. The main questions that will be addressed within this Dissertation are:

- Are there the "critical" pile group configurations and soil conditions, that can lead to the failure, if the influence of arbitrary loading direction is not considered in the design phase?

- What is the influence of loading direction on the horizontal force distribution, bending response and pile group efficiency?
- Shall the common concept of analysis of pile groups subjected to horizontal loading in two orthogonal directions be extended with additional cases of loading in arbitrary direction, as shown in Figure 1.4?

Assumptions

This research is based on the following assumptions:

1. Piles are bored and have circular cross section.
2. Piles are long and flexible.
3. Load acting on the pile group is static, monotonic load in a horizontal plane at the pile top ("active piles").
4. Soil conditions are drained, homogeneous and isotropic.

These assumptions are based on the facts summarized in the following paragraphs.

Bored piles

Bored piles are more expected for the urban environment, and also they sustain larger vertical loading, so the horizontal component from such large structures is expected to be relatively high. According to [22], "bored and CFA piles account for 50% of the world pile market, while the remaining is mainly covered by driven (42%) and screw (6%) piles. Summing up, the market is equally subdivided between displacement and non-displacement piles". As given in [1], bored piles are the most common form of piling in "contemporary high rise construction".

Long flexible piles

The influence of the pile length on the pile lateral response is widely known (see Figure 1.5). Usually considered pile failure modes are:

- Failure of the soil supporting the pile ("short pile failure"),
- Structural failure of the pile ("long pile failure").

Long piles are considered in this study, as more common case in the current engineering practice [13]. Most piles supporting buildings and bridges fall in this category.

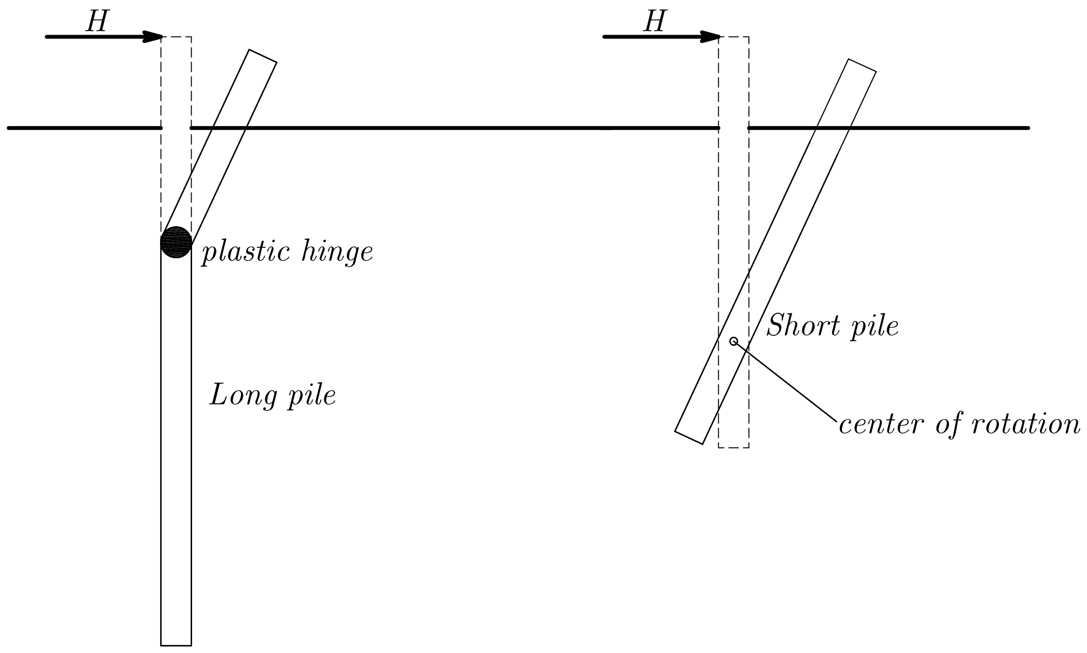


Figure 1.5: Long and short pile (after Broms 1964. [23, 24]) and their failure mechanisms

Static monotonic horizontal loading / Drained conditions

Many design codes use a pseudo-static approach to assess the action of dynamic horizontal forces on the foundation structure. In these approaches, equivalent static monotonic horizontal loading is applied to a structure [1].

Due to the fact that the static horizontal loading is considered, soil conditions are assumed as drained.

1.4 Research methodology

Full scale experiments can provide the best guidance in the behaviour of pile groups, but they are expensive and therefore not feasible solution for pile group analysis. Instead, numerical analysis usually remains the ultimate tool for large scope studies. Compared to the real experiments, numerical simulations allow for the large number of analyses to be executed faster and cheaper. The methodology used in this study follows the concept of ”numerical experiment”, where a numerical simulation is used to mimic the real experiment.

In order to achieve the defined research objectives, based on the presented assumptions, following working packages (WPs) have been established within this study:

- (WP 1) Problem introduction and definition of research objectives and methodology.

- (WP 2) Review of the state of the art regarding the laterally loaded pile group interaction effects, available calculation methods and previously done experimental studies. "Apriori" sensitivity analysis of the considered problem and definition of the main problem parameters are also conducted within this WP.

- (WP 3) Generation of numerical model for laterally loaded pile group. Validation of the proposed model is done by the back-calculation of available experimental results from the literature.

- (WP 4) Parametric study of pile groups with various configurations under arbitrary horizontal loading, with an emphasis on pile force distribution, bending response and pile group efficiency.

- (WP 5) Discussion of the achieved results and derivation of the conclusions of the research.

- (WP 6) Development of computer programs for the automatization of the process of preprocessing of numerical models, postprocessing of the results and the optimization of the calculation time through the use of multiple computers.

The process of the study is illustrated in Figure 1.6. All research steps and corresponding working packages are also followed by the Chapter numbers.

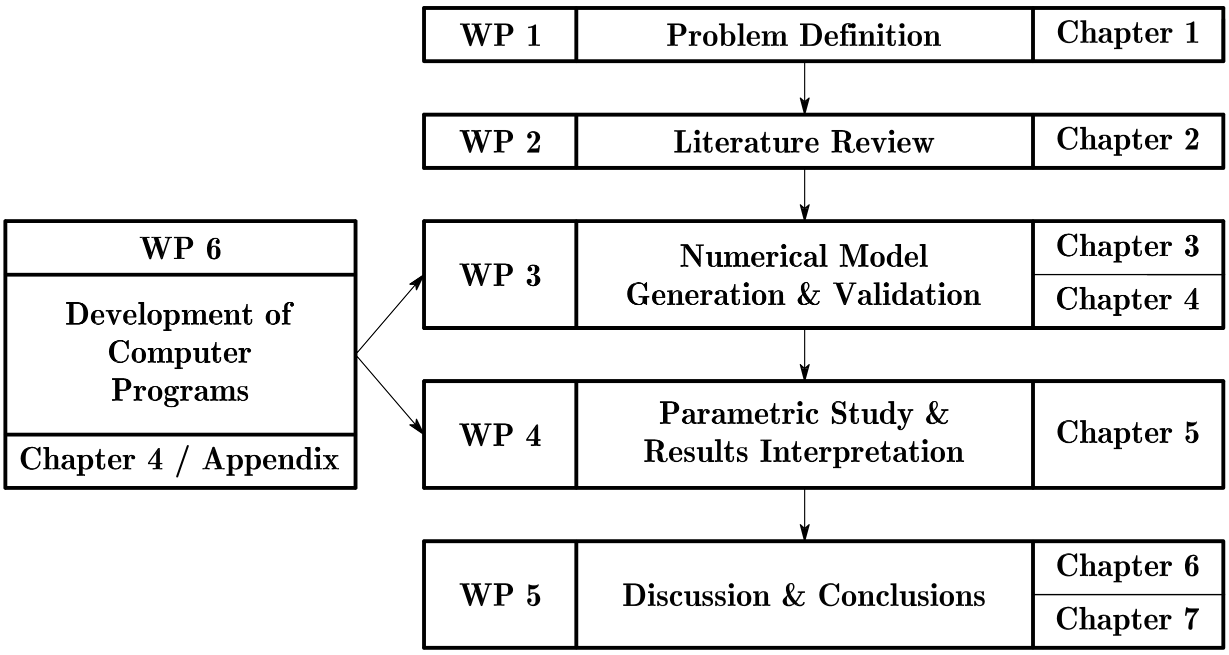


Figure 1.6: Flowchart of the study and the description of WPs

1.5 Organization of the Dissertation

This Dissertation comprises of eight Chapters and one Appendix. The whole Dissertation is organized in a progressive manner, so the final remarks in each Chapter serve as the basis for the next Chapter.

In **Chapter 2**, the review of different methods for analysis of the pile groups under lateral loading is presented. Important advantages and limitations of the current methods are explained. The most important problem parameters and observation points are identified and they serve as the basis for the generation of the numerical model. Also, current available experimental results have been summarized in order to support the scope of the research done within this Dissertation.

In **Chapter 3**, the 3D FEM modelling approach used as the basis for this study has been explained. Commercially available FEM software package PLAXIS 3D [25] has been briefly described. A detailed explanation of all numerical model components is presented, including the constitutive laws used for the representation of the soil, pile and the pile-soil interface. The limitations of the alternative available modelling techniques have been presented to scientifically support the chosen modelling approach. Developed computer programs and procedures for the support of the calculation process have been summarized.

In **Chapter 4**, calibration and validation of the proposed numerical model has been presented, based on the available results from the literature.

In **Chapter 5**, the scope and the results of the performed parametric study have been presented. The selection of the scope of the study has been explained and all calculation scenarios have been summarized.

In **Chapters 6 and 7**, obtained results were discussed and the main conclusions have been delivered. The magnitude (impact) of the pile group effects under arbitrary loading has been elaborated, in comparison with the behaviour of equivalent single pile. Additional recommendations for pile group design have been given.

In **Chapter 8**, the recommendations for the future research in in the field of laterally loaded pile groups have been given.

In **Appendix**, the developed computer programs used in this study have been elaborated to serve as a basis for further research.

2 State of the art

The pile group response under lateral loading has been investigated intensively in the past decades. Many interesting conclusions from both full scale and small scale experiments and various numerical studies have been drawn.

2.1 Differential equation of a laterally loaded pile

The most basic representation of a laterally loaded pile is the model of elastic vertical beam, laterally loaded and supported by uncoupled springs along the pile length. Winkler (1867) [26] introduced the modelling of pile behaviour using the springs to represent soil reactions against pile movement, with the hypothesis which states: "*the soil resistance at any point is a function of the displacements in that point*". The stiffness of the uncoupled springs is usually referred as the modulus of subgrade reaction k_s. In its basic form, k_s is assumed as constant, which leads to linear elastic approach. This method of analysis is usually reffered to as the *linear subgrade reaction method* or *beam on Winkler foundation* (BWF).

The differential equation of laterally loaded elastic beam is given as [27]:

$$EI\frac{d^4y}{dz^4} + P_z\frac{d^2y}{dz^2} + p(z) = 0 \tag{2.1}$$

- y - pile deflection (L),
- z - length along the pile (L),
- EI - flexural stiffness of the pile (FL^2),
- P_z - axial load of the pile (F),
- $p(z)$ - soil pressure at the depth z (F/L^2),
- k_s - modulus of subgrade reaction (F/L^3).

F and L denote the force and length units, respectively.

If we consider the pure lateral loading case (P_z=0) and introduce the aforementioned modulus of subgrade reaction k_s as proportional to soil pressure ($k_s = p(z)/y$), previous differential equation can be simplified to:

$$EI\frac{d^4y}{dz^4} + k_s y = 0 \tag{2.2}$$

Terzaghi [28] pointed out the importance of different k_s distributions for different soil types, governed by exponent a. The value of the modulus of subgrade reaction for each spring is usually defined as the function of the depth z and pile diameter D. In general, it can be defined as:

$$k_s = n_h(\frac{z}{D})^a \tag{2.3}$$

where:

- n_h - coefficient of the modulus of subgrade reaction (F/L^3),
- a - distibution exponent (-).

The following recommendations for the distribution of k_s are commonly used, as shown in Figure 2.1:

- For OC clay: $k_s = n_h$ ($a = 0$),
- For sand and NC clay: $k_s = n_h\frac{z}{D}$ ($a = 1$).

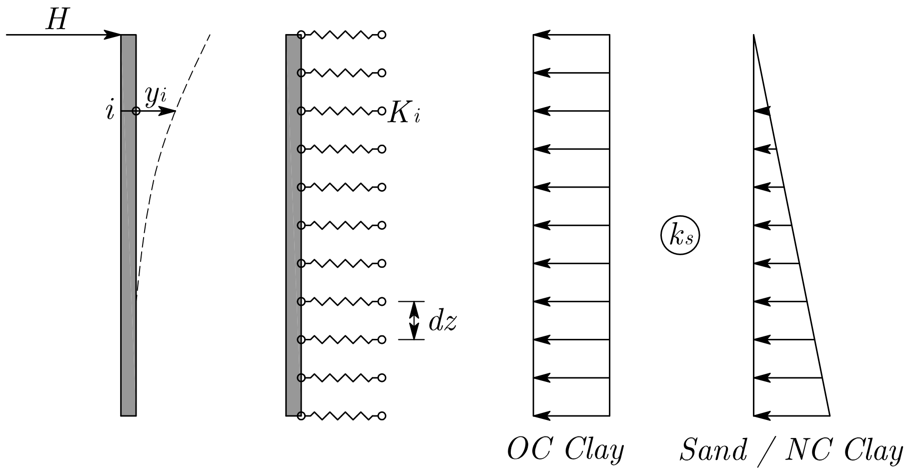


Figure 2.1: Distribution of modulus of subgrade reaction k_s for different soil types (K_i is the stiffness of the elastic spring at point i, calculated as $K_i = k_s \cdot dz \cdot D$ (F/L))

The following expressions for the shear forces Q and bending moments M for the elastic beam are well-known from the linear theory of elasticity:

$$M(z) = EI\frac{d^2y}{dz^2} \tag{2.4}$$

$$Q(z) = \frac{dM(z)}{dz} = EI\frac{d^3y}{dz^3} \tag{2.5}$$

Critical pile length

As mentioned in the research assumptions, usually considered pile failure modes are [23, 24]:

- Failure of the soil supporting the pile (”short pile failure”),
- Structural failure of the pile (”long pile failure”).

For every pile, critical pile length L_c can be defined, beyond which any additional length doesn’t affect the lateral pile response [27].

Long, flexible pile can then be defined using the following criterion:

$$L_c > 4T \tag{2.6}$$

where:

- T - relative stiffness factor, sometimes referred to as the elastic length of the pile (L).

Relative stiffness factor T is calculated according to the following expressions for sand/NC clay and OC clay, respectively:

$$T = (\frac{EI}{n_h})^{0.2} \tag{2.7}$$

$$T = (\frac{EI}{n_h})^{0.25} \tag{2.8}$$

It has to be pointed out that these critical length values will vary in real soil conditions, because of different distributions of the k_s over soil depth.

2.2 Pile group interaction effects

Interaction factors

The magnitude of the pile group interaction can be described by comparing the important parameters (horizontal displacements and bending moments) of a pile inside the group, against the parameters of equivalent single pile under the same mean loading. This concept is still used today in Germany, in everyday engineering practice, through the recommendations of DIN/EN codes [29, 30] and the recommendations of the *Working group ”Piles” (EAP)* [31].

The pile interaction factor α_i for pile i inside the pile group is defined using the following expression:

$$\alpha_i = \frac{H_i}{H_{SP}} \tag{2.9}$$

Based on the studies by Klüber and Kotthaus [21, 32], pile interaction factors are dependent on: pile spacing in two orthogonal directions, and pile position inside the pile group. Four types of the piles are distinguished.

Total interaction factor α_i is calculated using the non-dimensional partial interaction factors α_L, α_{QA} and α_{QA}, according to the Equations 2.10 and 2.11:

$$\alpha_i = \alpha_L \cdot \alpha_{QA} \tag{2.10}$$

$$\alpha_i = \alpha_L \cdot \alpha_{QZ} \tag{2.11}$$

where:

- α_L - partial interaction factor in the loading direction (-),
- α_{QA} - partial interaction factor perpendicular to the loading direction (-) - outer piles,
- α_{QZ} - partial interaction factor perpendicular to the loading direction (-) - inner piles.

The calculation of the interaction factor α_i is illustrated in the Figure 2.2:

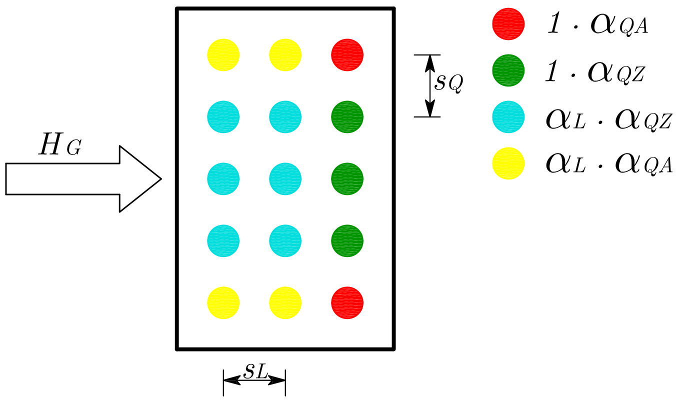


Figure 2.2: Interaction factors based on the pile position (after Kotthaus 1992. [21]). s_L and s_Q are pile center-to-center spacings in the direction of the loading H_G, and perpendicular to direction of the loading, respectively.

The proposed values of partial interaction factors α_L, α_{QA} and α_{QA} are given in Figure 2.3.

Based on the recommendations of EAP [31], the values of interaction factors can be used to calculate the modulus of subgrade reaction $k_{s,i}$ and pile bending moments M_i for the pile i inside the pile group. For the linear distribution of

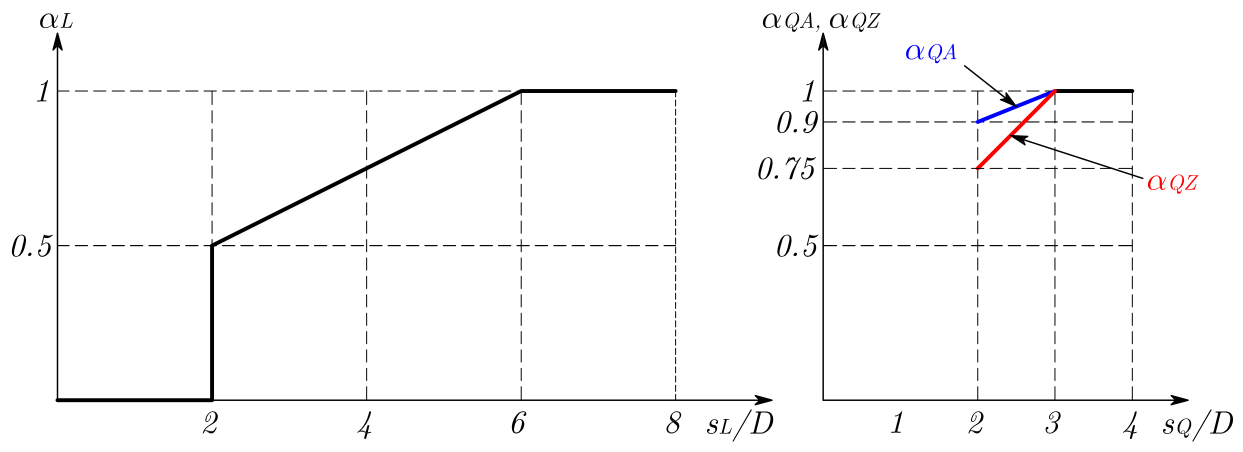


Figure 2.3: Partial interaction factor values based on the pile spacing in loading direction (s_L) and perpendicular to loading direction (s_Q), after Kotthaus 1992. [21]. Note that at the pile spacings $\alpha_L \geq 6D$ and $\alpha_Q \geq 3D$, no interaction occurs.

the modulus of subgrade reaction along the pile length ($k_s = n_h \frac{z}{D}$, for sand and NC clay), the following expressions are used:

$$k_{s,i} = \alpha_i^{\frac{5}{3}} k_{s,SP} \tag{2.12}$$

$$M_i = M_{SP} \alpha_i^{\frac{2}{3}} \tag{2.13}$$

For the constant distribution of the modulus of subgrade reaction along the pile length ($k_s = n_h$, for OC clay)

$$k_{s,i} = \alpha_i^{\frac{4}{3}} k_{s,SP} \tag{2.14}$$

$$M_i = M_{SP} \alpha_i^{\frac{2}{3}} \tag{2.15}$$

where:

- $k_{s,i}$ - modulus of subgrade reaction for the pile i inside the pile group at the point $z = D$,
- $k_{s,SP}$ - modulus of subgrade reaction for the equivalent single pile (SP) at the point $z = D$,
- M_i - bending moments for the pile i inside the pile group,
- M_{SP} - bending moments for the equivalent single pile (SP),
- α_i - interaction factor for pile i.

Pile group efficiency and load proportions

The pile group efficiency under lateral loading (non-dimensional, usually denoted as GW or η) can be described using the following equation:

$$\eta = GW = \frac{H_G}{nH_{SP}} = \frac{\sum_{i=1}^{n} H_i}{nH_{SP}} \tag{2.16}$$

where:

- H_i - lateral force acting on the pile i (F),
- H_G - total lateral force acting on the pile group (F),
- H_{SP} - lateral force acting on the equivalent single pile (F),
- n - total number of piles (-).

It can be shown that the pile group efficiency is equal to the mean value of all pile interaction factors α_i:

$$\eta = GW = \frac{\sum_{i=1}^{n} H_i}{nH_{SP}} = \frac{1}{n}\sum_{i=1}^{n} \frac{H_i}{H_{SP}} = \frac{1}{n}\sum_{i=1}^{n} \alpha_i = \alpha_{i,mean} \tag{2.17}$$

Load proportion for the pile i is defined as:

$$LP_i = \frac{H_i}{H_G} \tag{2.18}$$

It can be easily shown that the sum of all load proportions LP_i (for all piles inside the group) is always equal to 1.

2.3 Numerical methods

As the alternative to the expensive experimental studies, different numerical methods for the response prediction of pile groups under lateral loading have been developed. From a theoretical point of view, these methods can be divided into several categories:

- Closed form and empirical solutions,
- Limit equilibrium method,
- Strain Wedge (SW) method,

- Discrete load transfer methods,
- Continuum based methods.

As in the case of analysis of other geotechnical problems, the development in this field had similar progress, which is reflected in the gradual increase of the complexity of the calculation model, in order to take into account more aspects of real soil behaviour, as well as the soil-structure interaction effects.

Closed form and empirical solutions

Closed-form solutions for the problem of laterally loaded pile group are very limited. The most of these solutions assumes the soil as an elastic and isotropic half-space. Notable are the works of Winkler (1867) [26], Hetenyi (1946) [27] and Terzaghi (1955) [28].

Limit equilibrium method

Limit equilibrium method is represented by the work of Broms [23, 24]. This type of analysis is related to the ultimate (failure) conditions, where reasonable engineering assumptions regarding the soil pressure distribution can be made. These methods are nowadays mostly overcame by the load transfer and continuum based methods, especially because of the fact that the serviceability limit states under working load conditions (far from ultimate loading conditions) became the governing factor in the pile foundations design. Nevertheless, the limit equilibrium methods have founded the basis for the distinction between rigid and flexible pile response.

Strain Wedge Method

The Strain Wedge (SW) method for the analysis of laterally loaded piles and pile groups is based on the concept of mobilized passive soil wedge that resists the pile. Originally developed by Norris [33] for laterally loaded flexible free head pile in sand, it has been further improved for clay soils, pile groups, layered soils, boundary conditions and nonlinear behavior of pile material [34–37]. The main assumptions are that the deflection pattern of the pile is linear over the depth, which leads to the uniform horizontal and vertical strains. The horizontal strain ε in the soil in the passive wedge is the predominant parameter in the Strain Wedge model.

Geometry of the assumed 3D passive wedge (Figure 2.4) is defined by:

- Spreading angle (equal to the mobilized effective friction angle ϕ'_m),

- Height h (mobilized depth of a passive wedge),
- Mobilized base angle $\beta_m = 90 - \theta_m$.

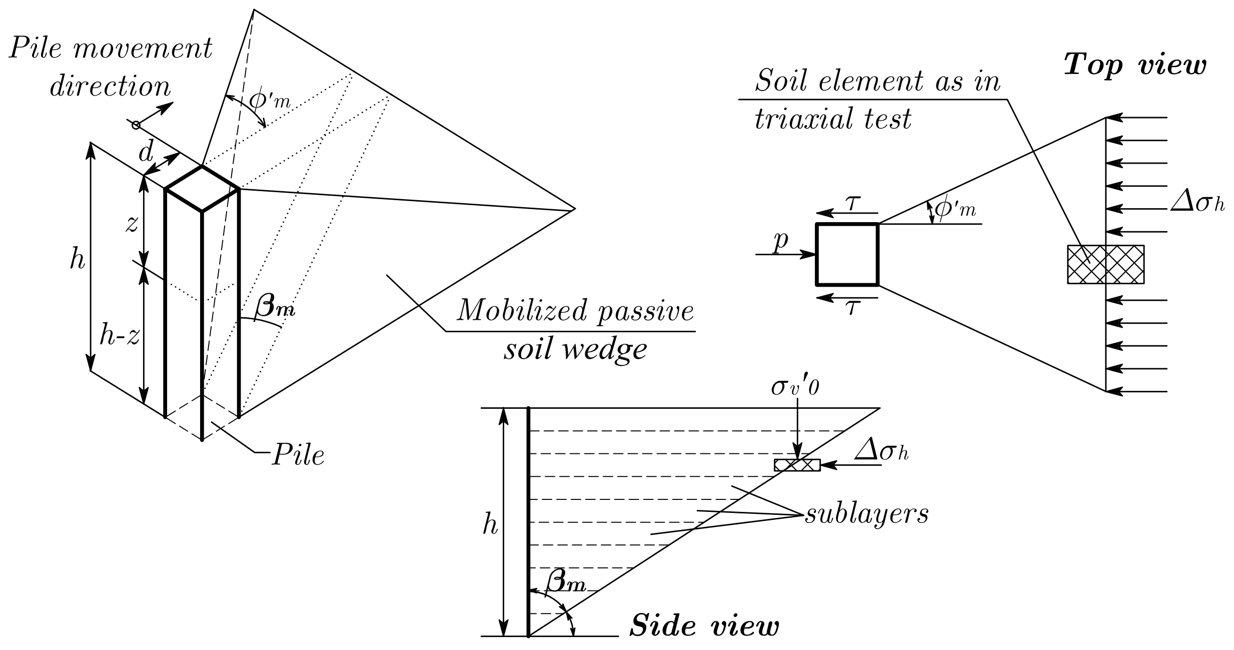


Figure 2.4: Strain wedge model parameters (after Norris 1986. [33])

The assumed resistance mechanism consists of the horizontal stress $\Delta\sigma_h$ (assumed equal to deviatoric triaxial stress) along the width of the wedge, and mobilized shear friction τ along the pile side. Every wedge is assumed to be formed of sublayers that represent the different stress states along the depth. 3D wedges are used to establish the equivalent nonlinear elastic springs for each sublayer. A power function for the stress-strain relationship is used to represent the constitutive behaviour for sand and clay, and every sublayer is considered as an uniform soil.

By introducing the equivalent springs for each sublayer, SW model connects the discrete load-transfer approach and limit equilibrium approach. The stiffness of each spring represents the secant value $k_s = p(z)/y(z)$, appropriate to the current deflection of the pile-soil system.

Discrete load transfer methods

Beside the previously mentioned elastic load transfer method (beam on Winkler foundation - BWF), further improvements of the subgrade reaction theory led to the nonlinear load-transfer approach, that included soil representation with nonlinear springs at discrete points along the pile length. This analysis is nowadays well recognized as **p-y curve method**. It was first introduced by

McCelland and Focht in 1958 [38]. It can also be denoted as beam on nonlinear Winkler foundation (BNWF).

Nonlinear behavior of the soil is defined by p-y curves associated to each spring, where p is lateral soil resistance per unit length of the pile and y is lateral deflection of the pile. Problem solution requires the nonlinear iterative analysis, in order to match the p-y nonlinear curve with the response of the one dimensional beam.

p-y curves are usually expressed using hyperbolic or power functions. They are mainly derived by back-calculation of the experimental results, which provide the confidence among the designers. In routine practice, p-y curves from 1970's [39, 40] are used as "the industry standard" [41], and these curves are implemented into the software solutions such as LPILE [42] and GROUP [43], as "standard" sand and clay curves. Additional enhancements of these curves have been attempted through the use of cone penetration (CPT), flat dilatometer [44] and pressuremeter tests [45], as well as the FEM [46, 47] for p-y curve derivation.

p-multipliers (P_m)

Pile group interaction effects in BNWF are taken into account by applying the reduction coefficients (p-multipliers) that reduce the bearing capacity of the soil in the trailing pile rows [48]. This concept is illustrated in Figure 2.5.

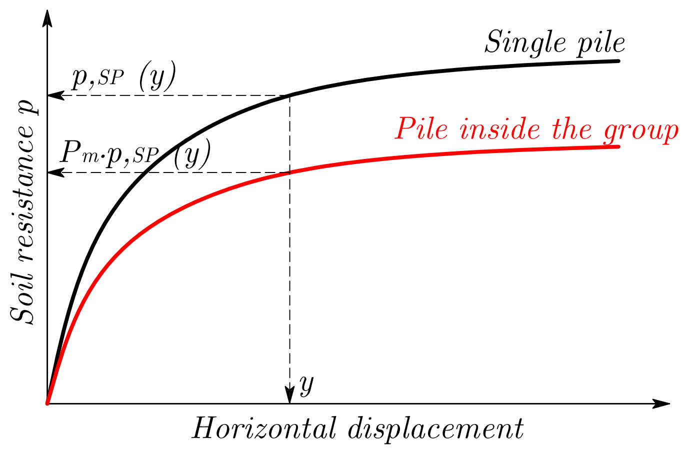


Figure 2.5: Definition of p-multipliers P_m (SP - single pile)

The calculation of p-multipliers is usually done by the adjustment of p-y curves, until the acceptable match between the experimental (measured) and calculated pile group deflection is obtained. The term "edge effect" is sometimes used to describe the effect of interaction between the piles inside the same pile row. It is common practice that the p-multipliers are defined as the average for pile rows (see Figure 2.6), despite the fact that the response of each pile inside the row is different. As an alternative, the average p-multiplier for the entire group can be used, which is called *group interaction factor* [49, 50]. Currently, there are many recommendations regarding the values of p-multipliers for different pile group configurations.

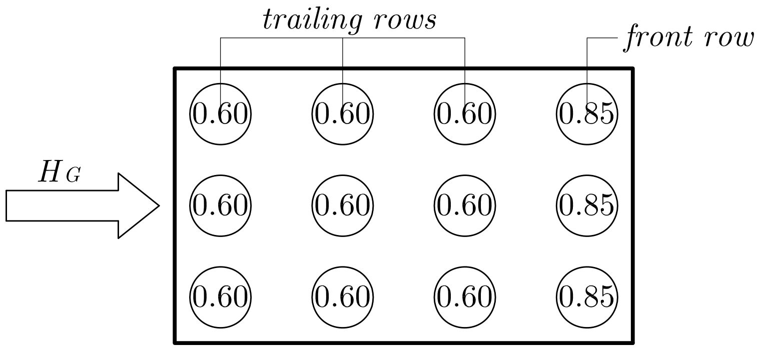


Figure 2.6: Example of p-multipliers for the pile group. The p-y curve for the pile inside the pile group is scaled for the entire displacements range.

Limitations

Linear discrete load transfer method have certain limitations, that are widely recognized in the engineering community:

- Soil resistance is modeled using discrete (discontinuous) springs, despite the fact that the soil is continuum (3D continuum interaction effects are neglected).
- Pile geometry is simplified to 1D beam element.
- The modulus of subgrade reaction k_s is not the physical soil parameter. Instead, it is the measure of the total soil-pile interaction at the specific depth, and hence it depends not only on the soil properties, but also the pile properties and loading conditions [51]. In other words, it is a convenient mathematical parameter that relates soil reaction and pile deflection [38].
- Soil resistance is linearly proportional to the pile displacements, despite widely known soil nonlinear behaviour.

Nonlinear (p-y) method has the additional limitation: it is very hard to decide which p-y curve to select. The p-y curves are mostly determined based on the experimental results and therefore are somehow bound to a certain soil type - the extrapolation to the different soil conditions and pile group configurations is questionable.

Despite the aforementioned limitations, linear and nonlinear subgrade reaction methods are widely used in the everyday engineering practice, due to their simplicity and the ability to model and predict the real pile behaviour accurately.

Continuum based methods

The limitations of the discrete load transfer methods, presented in the previous section, can be overcome using the continuum based methods. The development of computers has led to the rapid development of these methods, which allow for the full discretization of both the soil and the structure, combined with the application of advanced constitutive soil models. Applied to laterally loaded piles, representation of the soil and piles through the continuum provides a more fundamental and realistic approach to the analysis of pile-soil interaction. Also, material properties used within these methods now have clear physical meaning, and can be measured directly, using conventional geomechanical tests.

On the other hand, continuum based analysis requires more effort for the data preparation and higher computational demands. The fact that the most of the current structural design codes require the use of multiple load combinations makes the use of continuum based methods still impractical for the routine engineering design.

Continuum based solutions are mainly based on the following numerical analysis methods:

- Finite Element Method (FEM),
- Finite Difference Method (FDM),
- Boundary Element Method (BEM).

Each of these methods can be used with the different level of complexity. Usually, distinction between linear and nonlinear analysis is made.

FEM

The finite element method (FEM) is today considered as the most reliable and most widely used numerical method for engineering analysis of complex foundation systems [1, 52]. It is capable of performing 2D or 3D analysis. Regarding the laterally loaded pile, FEM allows for detail modelling of all model components: pile geometry, soil continuity and nonlinear behaviour, and especially the pile-soil interface through slippage and gapping. According to [41], "modeling lateral behavior in any way other than with three-dimensional models using nonlinear soil models and interface elements must constitute a compromise".

Many software packages such as PLAXIS 3D [25], ABAQUS [53], ETABS [54], ANSYS [55], SOFiSTiK [56] are commercially available. Regarding the pile foundations analysis, the alternative program FLAC 3D, based on the finite difference method (FDM) is also very common solution.

2.4 Notable numerical studies on laterally pile group interaction effects

Initial linear solutions

Poulos (1971)

Poulos proposed BEM model to predict the response of a laterally loaded free and fixed-head single piles [57] and pile groups [58] in 3D elastic, homogeneous and isotropic, semi-infinite continuum. Pile-soil separation was not taken into account. Pile displacements were computed using bending moment equations. Soil displacements were calculated using Mindlin's method, that accounts for additional displacements and rotations. The pile was assumed as the thin rectangular strip with the width equal to the pile diameter D, and the length L and bending stiffness EI.

Pile-to-pile interaction factors were introduced to account for the additional lateral displacements/rotations for the piles inside the pile group. The importance of pile orientation with respect to loading direction was emphasized, pointing out that the interaction between piles is the highest when two piles are aligned with the line parallel to loading direction.

Based on this solution, a series of design curves were developed [6] to present the lateral interaction factors as the function of spacing, orientation and relative pile flexibility. This solution was later improved by Banerjee and Driscoll [59], considering the simultaneous pile-to-pile interaction.

Randolph (1981)

Randolph [13] proposed interaction factors in the form of a convenient algebraic expression for flexible piles subjected to lateral loads and moments at the pile head. The FEM study, both for elastic homogeneous and Gibson (with linearly increasing stiffness with depth) soil was done. Proposed expression defines the interaction factor for two piles under inclination angle ψ.

Interaction factor $\alpha_{\rho F}$ for a free head flexible pile in homogeneous soil is expressed as:

$$\alpha_{\rho F} = 0.5(\frac{E_p}{G_c})^{1/7}\frac{D}{s}(1 + cos^2\psi) \tag{2.19}$$

where:

- $\alpha_{\rho F}$ - Poulos [58] interaction factor giving the increase in deflection of free head pile under lateral loading,
- s - pile center-to-center spacing,
- E_p – pile Young's modulus,
- G_c – average value of G^* over active length of pile, where $G^* = G(1+3\nu/4)$,
- ν - Poisson's ratio,
- ψ – angle between the line connecting the pile centers and the loading direction.

The solutions of Poulos [58] and Randolph [13] for 2 inclined piles are illustrated in Figure 2.7.

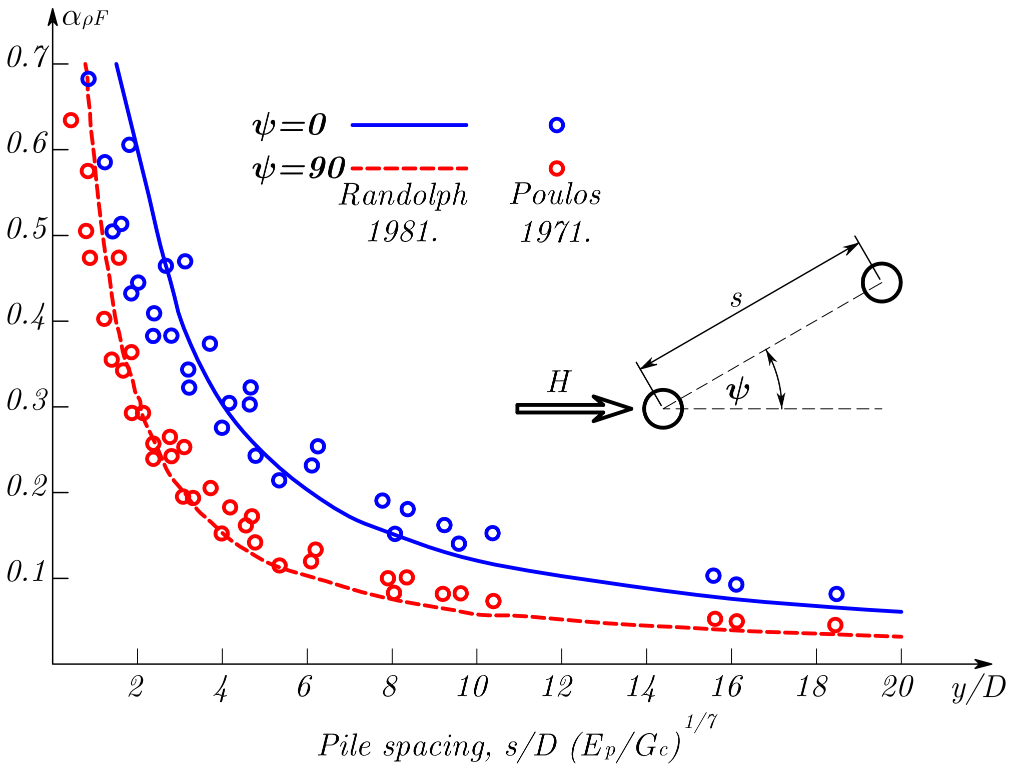


Figure 2.7: Initial elastic interaction factors $\alpha_{\rho F}$ for 2 inclined piles (after studies of Poulos 1971. [57] and Randolph 1981. [13])

Limitations of elastic solutions

Conventional structural design is still mostly based on the assumption of linear elasticity, which allows for the use of principle of superposition and therefore the ease of analysis and lower computation costs. However, in the case of very important structures, such as high rise buildings, power plants etc., the effects of nonlinear soil behavior and soil-structure interaction must be taken into account. As observed by Kotthaus [21], the interaction effects are not the same for different loading levels. Therefore, it is necessary to use the nonlinear analysis to assess the interaction effects at higher loading levels.

Elastic solutions are appropriate for the analysis of laterally loaded piles at very low loading levels. Pecker and Pender [60] noted that for 3x3 pile groups under static lateral loading and deflections up to $0.03D$, load distribution between the piles can be predicted using the elastic interaction factors.

Nonlinear numerical studies

Brown and Shie (1990-1991)

Brown and Shie [61, 62] conducted the numerical experiments using 3D FEM simulations to investigate the free head pile group effects, with group spacing of $3D$ and $5D$. Numerical model featured plastic yield, slippage and gapping at the pile-soil interface. Two soil types (clay and sand) were considered, using both Von Mises [63] and extended Drucker-Prager [64] models. Pile group effects were presented by p-multipliers, which are obtained through the polynomial fitting and differentiation of bending moment curves along the piles. This work motivated the further use of three dimensional FEM for the analysis of the pile group response. Linear elastic model was used for the representation of the pile.

Wakai et al. (1999)

Wakai et al. [65] performed the 3D elastoplastic FEM simulation of a full scale 3x3 pile group test. Soil was modeled using Mohr-Coulomb failure criterion and a non-associated flow rule. Interface slippage was modeled by means of thin elastoplastic elements in the pile-soil zone. However, the discrepancy between calculated and measured pile group response was observed at the higher loading levels (above $0.1D$). The load distribution ratio was found to be independent on the pile head fixity.

Yang and Jeremić (2003)

Yang and Jeremić [66] have analyzed the pile group interaction effects under lateral loading using 3D FEM model with separation and slippage capabilities. Layered sand soil cases were considered. Special attention was given to the bending moments in the plane perpendicular to the loading plane, and the p-y behaviour of individual piles in a group. It was shown that the pile row loads are different, as well as that the pile loads inside the same row is different. The bending moment inside the individual piles within the same row was different, as well. The level of pile group interaction was found to be dependent on the displacement level.

Comodromos et al. (2005-2013)

Comodromos and Pitilakis [10] analyzed the response of the laterally loaded fixed-head pile groups using 3D finite difference nonlinear analysis. Different pile group configurations were evaluated, and the influence of different parameters, such as number of piles, pile spacing and the deflection level was discussed. The pile installation effects were neglected. Papadopoulou and Comodromos further investigated the response of horizontally loaded fixed-head pile groups in sands [7] and clay soils [67]. The relationship between the load-deflection curve of the single pile and individual piles inside the pile group was derived by introducing the amplification factor R_a, defined as the ratio of group displacement vs. individual pile displacement for the same mean load.

Further research in this field led to the extension of the p-y method for sand [68] and clay soil [69], to account for the unequal load distribution inside the pile row. Location weighting factor lw_j was introduced to adjust the previously introduced amplification factor R_a using the effect of pile location. The values of the lw_j are slightly lower than those for the external piles, and slightly above those for the internal piles. The use of location weighting factors provides the calculation of forces and bending moments for each pile inside the pile group. This pile location weight factor lw_j can be directly associated with the previously defined interaction factors α_i and pile group efficiency GW, using simple algebraic transformations.

According to the above research, piles inside the pile group are divided into front corner (FC), front (FR), internal (IN) and perimetric (PE) piles, respectively (see Figure 2.8).

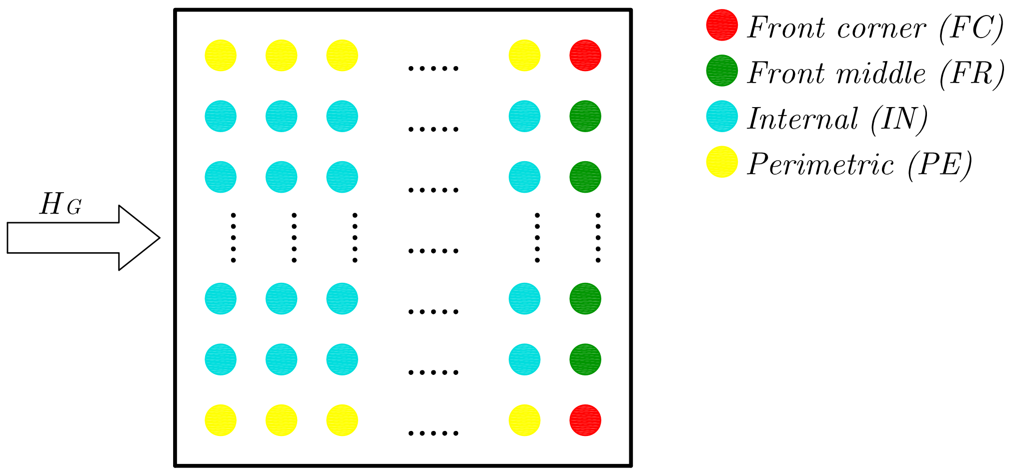


Figure 2.8: Pile types based on the location inside the pile group and the loading direction (after Papadopoulou and Comodromos, 2014 [68])

2.5 Experimental studies on laterally loaded pile group interaction effects

Finally, the behaviour of pile groups under horizontal loading is investigated using different experimental methods. Full scale tests can eliminate the most of uncertainties of the considered problem. However, due to high costs, only a limited number of full scale static pile load tests is available at this time. As an alternative, experiments with small scale models are also reported in the literature.

The summary of conducted experimental research of the laterally loaded pile groups is given in Table 2.1.

Table 2.1: Summary of experimental studies on laterally loaded pile groups

	Soil	**Test**	**Pile**	**Pile size**	**Installation**	**Pile head**	**Pile group**		**Pile row P-multipliers**				
Reference	**type**	**type**	**type**	**(cm)**	**method**	**BC**	**N x M**	**s/D**	**1**	**2**	**3**	**4**	**5**
Brown et al. (1987) [70]	Stiff clay	Full scale	Steel pipe	27.3	Driven	Free	3x3	3	0.70	0.60	0.50		
Brown et al. (1988) [48]	Dense sand	Full scale	Steel pipe	27.3	Bored	Free	3x3	3	0.80	0.40	0.30		
Brown et al. (2001) [49]	Silty sand	Full scale	Reinforced concrete	150.0	Bored	Fixed	2x3	3	0.50	0.40	0.30		
			Precast reinforced concrete	80.0	Driven	Free	3x4	3	0.90	0.70	0.50	0.40	
Christensen (2006) [71]	Sand	Full scale	Steel pipe	32.4	Driven	Free	3x3	5.65	1.00	0.70	0.65		
Huang et al. (2001) [72]	Silty sand	Full scale	Reinforced concrete	150	Bored	Fixed	3x3	3	0.93	0.70	0.74		
			Precast reinforced concrete	80	Driven	Fixed	4x4	3	0.89	0.61	0.61	0.66	
Ilyas et al. (2004) [73]	Clay (kaolin)	Centrifuge	Aluminum pipe (squared)	84.0	Jacked	Free	3x3	3	0.65	0.50	0.48		
							4x4	3	0.65	0.49	0.42	0.46	
							2x2	3	0.96	0.78			
							1x2	3	0.80	0.63			
Kotthaus et al. (1994) [74]	Dense sand	Centrifuge	Aluminum pipe	150	Sand rainfall	Free	1x3	3	0.75	0.45	0.45		
				150		Free	1x3	4	0.95	0.65	0.65		
McVay et al. (1995) [75]	Sand	Centrifuge	Aluminum pipe	43.0	Sand rainfall	Free	3x3	5	1.00	0.85	0.70		
							3x3	5	1.00	0.85	0.70		
							3x3	3	0.65	0.45	0.35		
							3x3	3	0.80	0.40	0.30		
Morrison and Reese (1986) [76]	Medium dense sand	Full scale	Steel pipe	27.3	Driven	Free	3x3	3	0.80	0.40	0.30		
Rollins et al. (1998) [77]	Clay	Full scale	Steel pipe	30.5	Driven	Free	3x3	3	0.60	0.38	0.43		
Rollins and Sparks (2002) [78]	Silts and clays	Full scale	Steel pipe	32.4	Driven	Fixed	3x3	3	0.60	0.38	0.43		
Rollins et al. (2005) [79]	Sand	Full scale	Steel pipe	32.4	Driven	Free	3x3	3.30	0.80	0.40	0.40		
Rollins et al. (2006) [80]	Stiff clay	Full scale	Steel pipe	61.0	Driven	Free	5x3	3	0.82	0.61	0.45		
				32.4			3x3	5.65	0.95	0.88	0.77		
				32.4			3x4	4.40	0.90	0.80	0.69	0.73	
				32.4			3x5	3.30	0.82	0.61	0.45	0.45	0.51
Ruesta and Townsend (1997) [81]	Loose sand	Full scale	Prestressed concrete (squared)	76.0	Driven	Free	4x4	3	0.80	0.70	0.30	0.30	
Snyder (2004) [82]	Soft clay	Full scale	Steel pipe	32.4	Driven	Free	3x5	3.92	1.00	0.81	0.59	0.71	0.59
Walsh (2005) [83]	Sand	Full scale	Steel pipe	32.4	Driven	Free	3x5	3.92	1.00	0.50	0.35	0.30	0.40

Beside the fact that full scale pile group tests are the most realistic approach, summarized experimental results show the lack of the studies for the arbitrary loading case, as well as some inconsistency (see Figure 2.9). The limitations in the available experimental database support the use of numerical methods to study the pile group interaction effects.

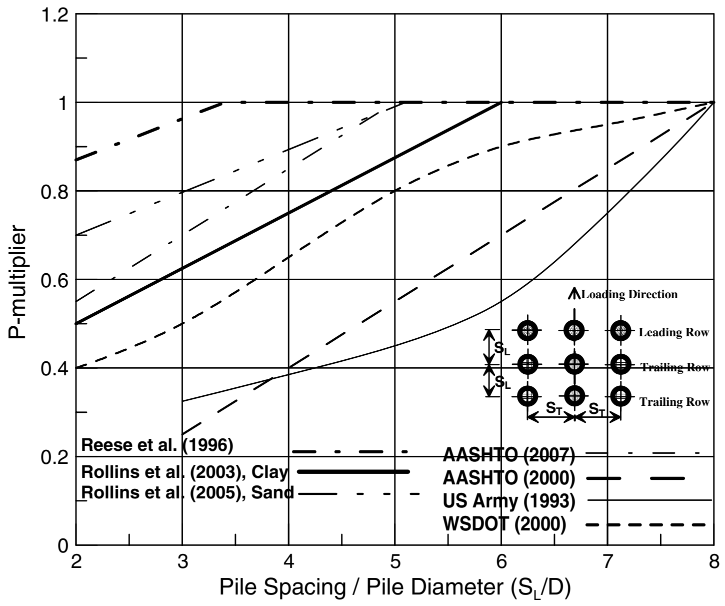


Figure 2.9: P-multipliers for different pile spacings from different sources (after Ashour and Ardalan, 2011 [84])

Deformation patterns

Recent experimental studies in the field of laterally loaded pile groups have analysed and visualized the soil deformation patterns around the piles. The improvement of the CT (computer tomography) and PIV (particle image velocimetry) technologies allowed to precisely define the shape of the deformed soil zone during lateral loading. Works by Otani et al. [85], Hajialilue-Bonab et al. [86, 87], Iai et al. [88], Morita et al. [89], are notable. The findings of these works clearly indicate the three-dimensional nature of the problem of laterally loaded pile group and indicate the almost conical shape of the soil deformation pattern. The soil failure was found to be progressive, and the failure zone is enlarged when the lateral load is increased.

2.6 Problem sensitivity analysis

As shown in previous sections, various factors can influence the interactions inside the laterally loaded pile group. Significant number of research papers

[48, 49, 72, 75, 77, 90–93] indicate and rank the most important factors that influence this problem. The most recognized parameters are:

- pile spacing in the loading direction,
- pile arrangement, position and total number of piles,
- pile stiffness, length and diameter,
- pile installation method,
- soil type and density,
- load/displacement level,
- pile head conditions,
- pile soil slippage and separation,
- presence of axial loading.

As stated by Mokwa and Duncan [94], in the adoption od p-multipliers, effects of pile-head fixity, soil type and density, as well as the displacement level, can be considered as the secondary issue that do not need to be considered for the most of the engineering purposes. According to Klüber [32], pile group efficiency is not the function of pile bending stiffness, relative density and pile head conditions.

The overview of the problem influencing factors is shown in Figure 2.10. The listed parameters are elaborated in more detail in the following paragraphs.

Pile spacing and configuration

It has been shown both theoretically and experimentally that the previously described shadowing effect becomes smaller when the pile spacing increases [10, 51]. There is general agreement in the literature that group effects are small when center-to-center pile spacing exceeds $6D$ in a direction parallel to the load, and $3D$ measured in a direction perpendicular to the load.

From the practical standpoint, if the pile spacing is higher than $5D$, the costs of the pile cap will increase to a very high level. According to McVay et al. (1995) [75], "pile spacing higher than $5D$ is considered as an extreme." Pile spacing of $2D$ is also not recommended due to the problems regarding the installation of closely spaced piles.

Based on the results of Comodromos and Pitilakis [10], both the number of pile rows and total number of piles play an important, but less affecting role in the pile group interaction effects. Same authors introduced the pile location

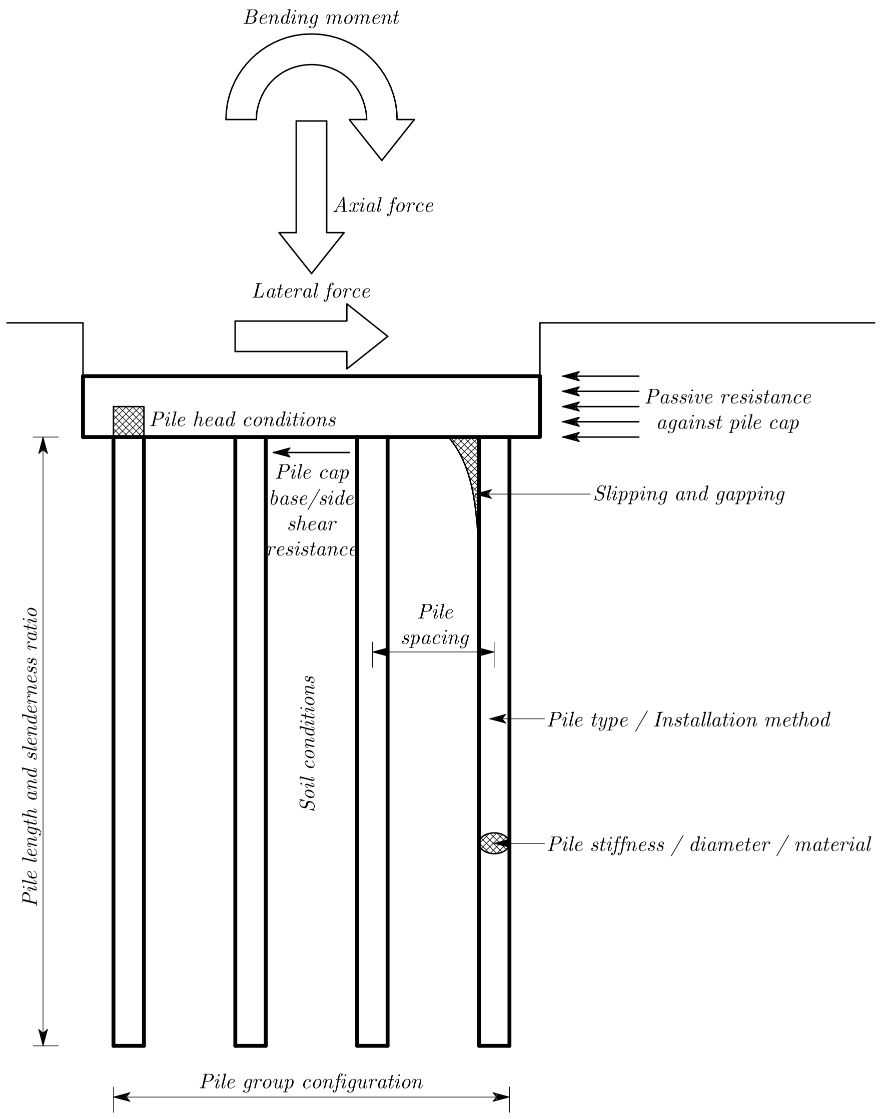

Bending moment
Axial force
Lateral force
Passive resistance against pile cap
Pile head conditions
Pile cap base/side shear resistance
Slipping and gapping
Pile spacing
Pile type / Installation method
Soil conditions
Pile length and slenderness ratio
Pile stiffness / diameter / material
Pile group configuration

Figure 2.10: Influencing factors for the laterally loaded pile group

weight factor lw_j, that follows the similar concept as given in German codes [29, 30], distinguishing between the front center, front corner, internal center and side piles, respectively.

Pile length and diameter

The effects of the pile length on the interaction inside the pile group were analyzed by Comodromos and Papadopoulou [67]. They shown that the pile length do not have the significant impact on the level of interaction inside the pile group, compared to the other problem parameters. However, it was pointed out that these statements should be verified. In German code DIN 4014 [30], pile length was not considered as important model parameter for the pile force distribution. Randolph [13] also pointed out that the pile length is not a relevant parameter for the analysis of the laterally loaded flexible pile group response.

According to Comodromos and Papadopoulou [67], it was shown that the pile diameter do not have the significant influence on the interaction level inside the pile group, compared to other parameters. However, some experimental studies [95, 96] have indicated the stiffer pile response with increasing pile diameter. In the work of Terzaghi [28], the subgrade modulus was considered independent of the pile diameter.

Pile stiffness and concrete cracking

Bowles [97] pointed out the possibility of concrete cracking as the problem in realistic modelling of the laterally loaded pile. Due to decrease in pile stiffness, nonlinear foundation response can be affected. Comodromos and Papadopoulou [98] analysed the influence of concrete cracking on the response of laterally loaded piles. As expected, this problem is more important in the case of free head pile response, because the cracks develop earlier due to the higher bending curvature. Free head piles are also affected to a higher depth, compared to a fixed head pile. However, for the working load level, which is mostly expected to occur for the laterally loaded piles, fully elastic assumption of the pile behaviour will not lead to the significant change in the interaction effects and stress redistribution. As the conclusion, pile cracking should be considered when extremely high accuracy is required and when the loading level is significant.

Displacement level

Many researches [10, 21, 80, 99] have shown that the pile group efficiency under lateral loading is decreased with the increase of the total pile group displacement level. Based on the studies of Comodromos and Papadopoulou [7, 67], a displacement of about 10% for sandy and about 15% for clay soil was sufficient to reach the constant level of interaction. Same authors also note that the interaction is significantly changed inside the displacement range of 1-5%. The level of displacement of 0.10D is, in general, higher than the level of displacements that is allowed in most practical cases. The serviceability limit state usually corresponds to a deflection level of $0.05D$ [69].

Pile head boundary conditions

In classical geotechnical engineering, two limiting pile head boundary conditions are usually considered within the analysis of laterally loaded pile groups:

- Free head pile, when pile head rotation is not constrained,

- Fixed head pile, when pile head rotation is constrained by the pile cap.

According to Mokwa and Duncan [99], boundary conditions at the pile head are somewhere in between the fixed and free head pile. Same authors pointed out that the pure fixed-head conditions are hard to achieve in reality, even when the pile cap is very stiff. In piled raft foundations, according to Katzenbach [100], "the connection between the piles and the raft is seen as flexible joint".

It is generally accepted that, due to different boundary conditions, the response of the free and fixed head pile groups subjected to lateral loading is significantly different [98]. Namely the deflection of the free head pile is higher than the fixed head pile for the same mean load.

Despite the different deflection response, pile load redistribution inside the pile group can be considered independent on the pile head conditions. Various researches, such as Matlock (1970) [39], Reese and Van Impe (2001) [101], or Comodromos and Papadopoulou (2012) [67], confirm this statement, especially within the practical ranges of the pile group displacements (working load conditions). The most of the recommended p-y curves in the current engineering practice are independent on the pile head conditions.

It should be noted here that the pile cap is usually in the contact with the soil, and therefore will contribute to the lateral bearing capacity of the pile group. However, this effect is usually neglected, because the cap-soil contact may be lost due to soil erosion or excavation [102].

Pile-soil slippage and separation

In reality, the contact between the piles and the surrounding soil is not fully rigid, but weaker, so there are small relative displacements along the pile-soil interface. Many authors [10, 41, 61, 103–105] pointed out the importance of simulating the slippage and separation between the soil and the pile on the more accurate model response. According to Comodromos [106], pile-soil separation is considered as the main reason for non-linear interaction inside the pile group. Regarding the affected zone, pile group response is mainly governed by the properties of the top soil layer (near-surface soil conditions) [1, 10].

Presence of axial loading

Pile foundations are always subjected to simultaneous vertical and lateral loading. Due to pile-soil interaction, there is an interaction between the axial and lateral response of the piles. However, in current design practice these interaction effects are usually neglected, and pile group interaction is analysed independently for vertical and lateral loading cases. Several studies have shown the interest in the analysis of combined axial and lateral loadings, both for clay and sand soils [107–113]. The general conclusions are that in the case of clays, the presence of vertical loading reduces the lateral capacity of piles, while the lateral capacity in sands is increased. These effects are observed at higher loading levels, while for working loading levels these combined effects can be neglected.

2.7 Concluding remarks

Based on the presented literature review and problem sensitivity analysis, the following conclusions for the next research phase were derived:

- The problem of laterally loaded pile group is, in general, a 3D engineering problem, so the 3D model will be used for the numerical study. Eurocode 7 [11] strongly suggests the use of soil-structure interaction in the numerical analysis, especially in the case of complex foundations such as piled raft foundations, laterally loaded pile foundations and flexible retaining structures. The use of advanced constitutive models is also suggested. The presented overview of previously conducted numerical studies has shown that a 3D continuum approach can provide a greater degree of realism. This is even more emphasized considering the arbitrary loading direction, because the symmetry modelling simplifications cannot be made.
- Only free head model will be validated and used in the parametric study. Despite the fact that the displacement level is influenced by the pile top boundary conditions, the presented literature review suggests that pile load redistribution is independent on the pile head fixity.
- For the considered working loading conditions within this study, fully elastic assumption of the pile behaviour will not lead to the significant change in interaction effects. Pile section properties will be modelled as constant along the pile length.
- Pile-soil slippage and separation effects must be considered in order to properly assess the pile group response.
- Soil profile will be considered as homogeneous, because only the top soil layer properties significantly influence the lateral pile group response. This is even more pronounced in the considered case of long piles.
- Pile length and diameter can be considered as the secondary problem parameters and they will be kept constant throughout the parametric study. Sand soil profile with two different densities will be analyzed.

- Maximum displacement level in the numerical model will be kept at about $0.10D$, which slightly exceeds the working load limits, but doesn't reach the ultimate limit state. For the problem under consideration, the working loads are more interesting. Also, by reducing the maximum displacements, the speed of numerical simulations is optimized.
- Influence of the axial load can be neglected in the numerical model. For considered working load conditions, the axial load will not significantly influence the lateral response of the pile group, or, in the case of the sand soil, neglecting of the axial load will be on the safe side.

3 Numerical model generation

3.1 PLAXIS 3D overview

Within this thesis, the response of the pile groups under arbitrary horizontal loading was computed using FEM code PLAXIS 3D [25], in a series of extensive numerical computations. It is a world-wide famous code for the stress-strain, stability and groundwater flow analysis in geotechnical engineering. Its development began in 1987 at Delft University of Technology, and in subsequent years it was extended to cover many areas of geotechnical engineering. 3D analysis support was introduced in 2001.

PLAXIS 3D features fully 3D graphical pre- and post-processor, that allows for an easy graphical input of the models with complex geometry, as well as the illustrative presentation of the results. FE mesh is mainly generated automatically, with the state-of-the-art algorithms for check of mesh regularity. A range of other structural members, such as beams, anchors, plates and geogrids can be modelled, as well. Staged (phase) analysis is also supported, so realistic simulation of excavation and construction process, starting from the initial stress states, can be modelled.

Special feature is the simulation of the conventional soil tests. This feature, denoted as *SoilTest*, allows for the calibration of the constitutive model parameters using the results of the standard soil laboratory tests. It is based on the single point algorithm, and it works for all soil models in PLAXIS 3D. The parameter optimization and local sensitivity analysis can be done in order to best fit the experimental results with the constitutive model.

3.2 Constitutive models

PLAXIS 3D features the constitutive models with different level of complexity. Also, user defined models can be implemented. Constitutive models used in this study are presented in the following sections. It must be noted here that the stress path around the laterally loaded pile does not match the conventional geomechanical laboratory tests [106].

Linear elastic (LE) model

The linear elastic model in PLAXIS 3D is based on the generalized Hooke's law of isotropic elasticity. It includes 2 basic elastic parameters – Young's modulus E and Poisson's ratio ν. Stress states in linear elastic model in PLAXIS 3D are not limited, which means that the simulated material can have infinite strength.

However, in order to simulate the piles using the linear elastic model, the one should ensure that the pile material (e.g. concrete) strength is not exceeded in the considered problem. This assumption is usually satisfactory in the case of the working load conditions, where concrete cracking can be neglected.

The parameters of the LE model used in this study are given in Table 3.1, and denoted as $E\_CONCRETE$, $NU\_CONCRETE$ and $W\_CONCRETE$.

Mohr-Coulomb (MC) model

The linear elastic-perfectly plastic Mohr-Coulomb (MC) model is based on the well-known Mohr-Coulomb failure criterion. This model usually represent the first order of the approximation of the soil and rock behaviour [25], and can be relevant for simple stress paths. This model involves five input parameters: Young's modulus E and Poisson's ratio ν for soil elasticity, and cohesion c', angle of internal friction ϕ' and dilation angle ψ for soil plasticity.

The stiffness in Mohr-Coulomb model is constant stiffness, independent of stress and strain levels (only controlled by Young's modulus E and Poisson's ratio ν). The volume changes and the rate of plastic strain variation are controlled by a non-associated flow rule, where the potential surface is defined by dilation angle ψ.

MC model is commonly used in practice due to the straightforward parameter determination using conventional laboratory tests. However, this model cannot account for the stress dependant stiffness, the stress-path dependant stiffness, as well as for the soil anisotropy. This means that the results will be questionable in the case of complex stress paths and large domains with wide stress variation.

In general, this model is considered adequate for stability analysis (that include shear strength of frictional soil), but inadequate for the problems that include deformation analysis or cyclic loading [114].

In this study, the constitutive behaviour of pile-soil interface is defined by the elastic-perfectly plastic Mohr-Coulomb model, in conjunction with a non-associated flow rule and zero tension cut-off criterion (when tension develops, a gap between the pile and the surrounding soil is created).

The parameters of the MC model used in this study are given in Table 3.1 (denoted as $COHESION$, PHI, NU and $RINTER$).

Hardening Soil (HS) model

The Hardening Soil (HS) constitutive model [115] is an advanced model. As in the case of the MC model, the limit stress states are defined through the parameters of MC failure criterion (c', ϕ', ψ). However, the soil stiffness in the HS model is described more accurately, through the three advanced stiffness parameters: the triaxial loading stiffness E_{50}, the triaxial unloading stiffness E_ur and the oedometer loading stiffness E_{oed}. These three values are not mutually independent: usually the E_{oed} is assumed equal to E_{50}, and E_{ur} is assumed equal to $3E_{50}$. These default relations are implemented into PLAXIS 3D code, but very soft and very stiff soils tend to give other relations. All three input stiffnesses are related to the reference stress $p_{ref} = 100$ kPa.

HS model is isotropic hardening model, that cannot account for and simulate cyclic behaviour. It also does not account for different stiffness at different strain levels – the user should input the stiffness according to the expected strain level. From the calculation point of view, this model also needs more calculation time. The HS model stiffness parameters are more precisely explained in Figure 3.1.

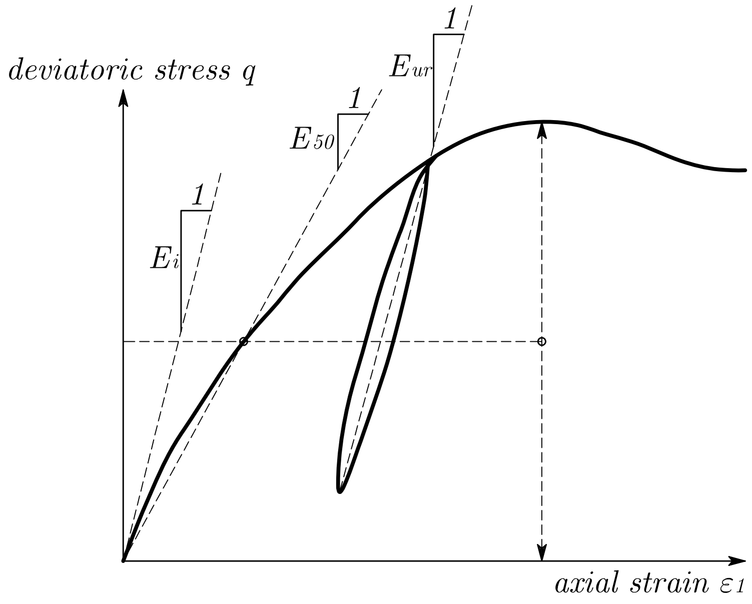


Figure 3.1: HS model stiffness parameters (after Schanz and Vermeer 1999. [115])

In contrast to MC model, the yield surface of the HS model is not fixed in the principal stress space, but it can expand due to plastic straining. Two types of hardening are contained in the model: shear hardening (due to primary deviatoric loading) and compression hardening (due to oedometer and isotropic loading). As soon as ultimate deviatoric stress q_f is reached, the MC failure criterion is satisfied and perfectly plastic yielding occurs. When subjected to primary deviatoric loading, soil shows a decreasing stiffness and simultaneously irreversible plastic strains develop.

The main property of Hardening Soil model is the hyperbolic relationship between deviatoric stress q and the axial strain ε_1 in primary triaxial loading. This type of soil behavior is observed in the drained triaxial test.

Likewise MC model, the HS model parameters can be extracted from conventional laboratory tests (i.e. triaxial and oedometer tests for non-monotonic loading).

Stress dependent stiffnesses (E_{50}, E_{oed}, E_{ur}) in the HS model are defined via the following equations:

$$E_{oed} = E_{oed}^{ref}(\frac{\sigma_1 + cctg\varphi}{p^{ref} + cctg\varphi})^m \tag{3.1}$$

$$E_{50} = E_{50}^{ref}(\frac{\sigma_3 + cctg\varphi}{p^{ref} + cctg\varphi})^m \tag{3.2}$$

$$E_{ur} = E_{ur}^{ref}(\frac{\sigma_3 + cctg\varphi}{p^{ref} + cctg\varphi})^m \tag{3.3}$$

where:

- E_{oed}^{ref} - reference tangent oedometric stiffness modulus, corresponding to the reference confining pressure p_{ref} (F/L^2),
- E_{50}^{ref} - reference secant stiffness modulus corresponding to the reference confining pressure p_{ref} (F/L^2),
- E_{ur}^{ref} - reference Young's modulus for unloading/reloading, corresponding to the reference confining pressure p_{ref} (F/L^2),
- σ_1 - first principal stress (F/L^2),
- σ_3 - third principal stress (F/L^2),
- c - cohesion (F/L^2),
- φ - angle of internal friction (degrees),
- p^{ref} - reference confining pressure (F/L^2),
- m - power for stress level dependency of stiffness (F/L^2).

The parameters of the HS model used in this study are given in Table 3.1 (denoted as $COHESION$, PHI, NU, $SOIL\_WEIGHT$, $E50\_EOED$, $EUR\_MULTIPLIER$ and $POWERM$).

3.3 FEM model of the pile group under lateral loading

Numerical prediction of the laterally loaded pile group is a challenging task, because it requires realistic modelling of the deformation behaviour of the soil, pile and the interface between the soil and the pile. Two pile modelling techniques are mainly used in PLAXIS 3D: full 3D (solid) pile model and recently implemented embedded beam (EB) model.

Full 3D (solid) pile model

In full 3D analysis, piles are discretized using 3D (solid) finite elements. The advantage in this type of modelling is the fact that the pile shaft geometry and soil-structure interaction along it are modelled accurately. However, this often leads to very large models (by means of computational complexity), which can be time consuming in everyday engineering practice.

The soil volumes in PLAXIS 3D are modelled using 10-node tetrahedral elements. These elements have three translational degrees of freedom (u_x, u_y, u_z) per node. Nodes are located in the corners and the middles of the tetrahedron, as shown in Figure 3.2 (left). This type of element provides a second-order (quadratic) interpolation of nodal displacements.

Embedded beam model (EB)

The embedded beam model was introduced by Sadek and Shahrour (2004) [116]. In this concept, pile volume isn't discretized using 3D elements, but replaced with advanced formulation. EB is a beam element that can be inserted (embedded) at arbitrary direction into the existing FE mesh of solid elements. After insertion, additional "virtual" nodes are generated at penetration points, inside the existing FE mesh of soil elements (Figure 3.2). Therefore, they don't affect the discretization of soil continuum. As an improvement of initial formulation [116], an elastic zone is assumed around the EB element [117], where plasticity in soil elements cannot occur. This elastic zone bounds the space occupied by a real pile, and its size is governed by the pile diameter D.

Pile-soil interaction at both the pile shaft and the pile tip is modelled using special interface 3-node spring elements in axial and lateral directions. These elements "connect" the EB nodes with the virtual soil nodes.

The main advantage of EB is increased calculation speed and easy determination of section forces along the pile. However, because no real discretization of pile volume is made, EB doesn't take into account the sliding between the pile and the surrounding soil. The soil-structure interaction is modeled along

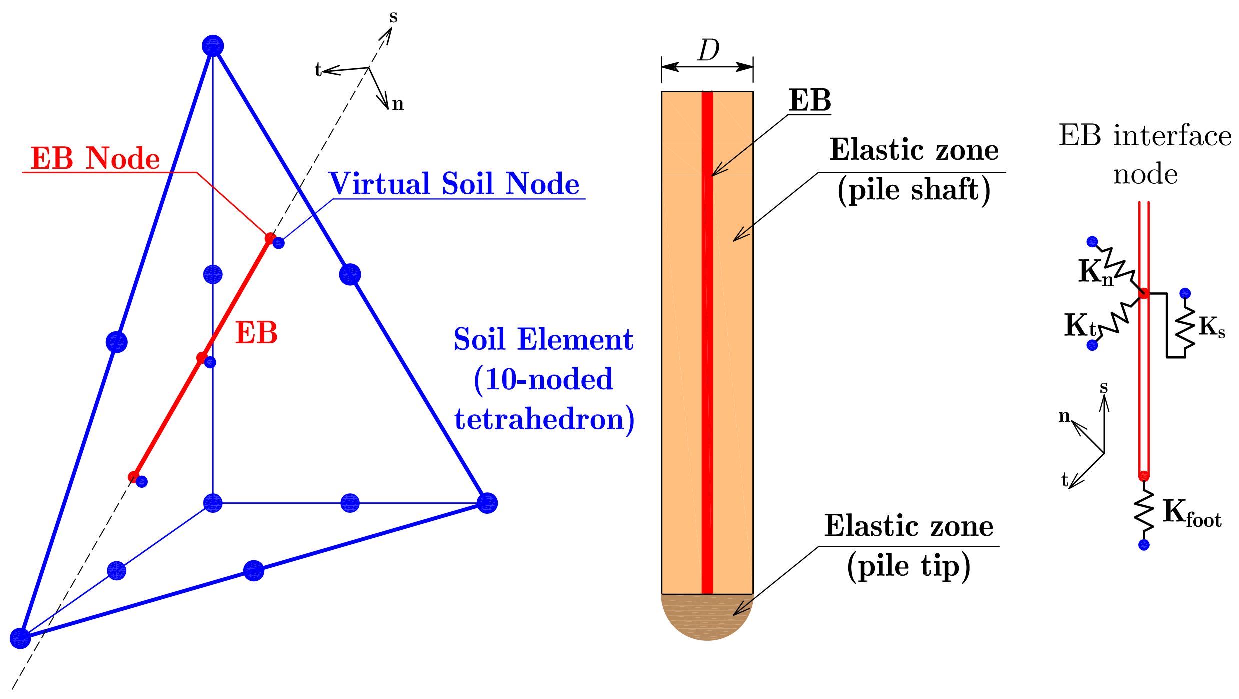


Figure 3.2: EB model in PLAXIS 3D (after Brinkgreve, 2015 [25]). K_n, K_t, K_s and K_{foot} denote the interface spring stiffness at pile shaft (in 3 directions) and pile tip (in vertical direction).

the pile axis, instead the pile shaft, so the gapping between the soil and the pile can not be accurately modeled. Also, published results by author [118] and Dao [119] show that embedded pile model is sensitive to FEM mesh coarseness. This is probably due to the different number of virtual nodes that are generated in different finite element meshes, which leads to a different level of approximation.

EB model vs. full 3D (solid) pile model

Comparison of two described modelling techniques (EB vs. 3D (volume) pile model, denoted as VP, was presented by author et al. [118] on idealized example of laterally loaded 2x2 pile group with center to center pile spacing of $4D$. Piles were 10m long, with diameter of 0.5m. Two different types of soil (loose and dense sand) were considered. Numerical simulations were performed as displacement control tests with prescribed displacement of $0.2D$ at the top of the piles, applied in 8 equal increments. The case of fully rigid interface, as well as soft pile-soil interface, were simulated.

Difference in bearing capacity between the front and trailing rows was observed in both EB and VP models, which means that both models can qualitatively resemble the laterally loaded pile group behaviour. The results of analysis have shown also that interface properties don't influence the lateral behaviour of EB models, while the VP models are influenced, as expected. This is associated with the formulation of EB interface, where only the shear stresses in axial direction are governed by interface input parameters. It was presented that EB single pile model agrees well with the volume pile model with rigid interface, while the weaker pile-soil contact cannot be resembled with EB interface parameters.

The obtained results concludes that the interface properties in the EB model formulation in PLAXIS 3D do not influence the lateral response of the pile group. However, when shear pile forces become large, plasticity will occur in the surrounding soil elements, outside the elastic zone. In other words, pile-soil interaction in lateral directions can only be controlled by the adjustments of surrounding soil stiffness parameters.

Concluding remarks

Recent researches by several authors in the direction of improvement of initial EB model to account for the lateral interface interaction have been done and the results are very promising [120–122]. However, due to the current the limitations of the PLAXIS 3D EB model for lateral loading and lack of software packages with implemented improved EB formulations, full 3D pile analysis has been done within this Dissertation.

Pile-soil interface

The modelling of pile-soil interface (contact) is another important factor in realistic pile group simulation. In general, the pile-soil interface strength depends on the type of pile material (wood, steel or concrete) and pile installation method (driven or bored). Due to the fact that this study only considered the bored piles, installation effects and loading history were not taken into account. In the case of driven piles, installation effects are found to be important [72, 81].

Because the soil has limited or no capacity in sustaining tension, slippage and separation (gapping) between the pile and the soil will occur. In order to simulate these effects, thin 2D interface elements are inserted between the soil and pile elements. These elements are different from the regular finite elements: they have pairs of nodes instead of single nodes (the distance between the two nodes of a node pair is zero), and each node has three translational degrees of freedom (u_x, u_y, u_z). As a result, these elements allow for differential displacements between the node pairs to simulate both slipping and gapping on the pile-soil contact [25].

The shear strength parameters of the pile-soil interface in PLAXIS 3D are related to the strength reduction factor R_{inter}, that reduces the parameters of MC model (soil cohesion c' and angle of internal friction φ'), according to the following expressions:

$$c_{int} = R_{inter} c' \tag{3.4}$$

$$tan\varphi_{int} = R_{inter} tan\varphi' \tag{3.5}$$

where:

- c_{int} - cohesion of the pile-soil interface,
- φ_{int} - angle of internal friction of the pile-soil interface.

In general, there is the lack of the experimental data for real pile-soil interface parameters. Suitable values of R_{inter} are recommended in the literature [25] for different soil types, and usual value is around $R_{inter} = 0.5$ (that yields the well-known relationship $\varphi_{int} = 2/3\varphi'$).

"Dummy" beams

The extraction of the pile group results from the 3D FEM simulation is not a simple task. This is mainly related to determination of pile section forces. That is theoretically done by integration of component stresses over the pile cross section. Despite the fact that the calculation of the section forces from volumes is implemented in PLAXIS 3D, this step cannot be done automatically, but requires that the user perform the whole procedure manually for each pile volume.

In order to simplify the extraction of pile section forces and displacements and to automatize the post-processing as well, the concept of "dummy" beams is used. This modelling concept is common when the laterally loaded piles are modelled using 3D elements [123, 124]. The elastic beam elements with very small bending stiffness (10^6 times smaller than the pile bending stiffness) are inserted along the pile axis. Because the beam stiffness is very small, system stiffness matrix remains almost unchanged. On the other hand, "dummy" beam is enforced to deform together with the pile axis. Because the pile displacements and forces are directly coupled through the basic equations of elastic beam, real values of pile section forces are now easily obtained by multiplication of section forces in "dummy" beams with 10^6 (the bending stiffness ratio between real pile and "dummy" beam).

3.4 Calculation stages

Numerical simulations within the thesis were done as the staged (phase) analysis and consisted of the following stages:

1. **Initial (K_0) phase** - initial stress field in the soil, due to the soil self weight, was established. Horizontal stress state is calculated based on the well-known Jaky's formula [125]: $K_0 = 1 - sin\varphi'$.

2. **Construction stage** - after initial stress state has been established, soil volume is replaced with 3D pile finite elements (wished-in-place concept). Thin 2D pile-soil interface elements are also activated in this phase.

3. **Prescribed displacements (Displacement control test) - 4 successive phases** - the incremental prescribed displacements up to 10% of the pile diameter were applied at the pile top. The direction of loading was governed by the loading direction angle β (denoted as parameter $ANGLE$ in Table 3.1).

Optimization of the PLAXIS 3D calculation algorithm

Computation time and size of the saved raw simulation on computer drive in PLAXIS 3D is directly proportional to the number of calculation stages. Because of this, the optimization has been done, in order to reduce the storage needs of calculation computers, as well as the calculation speed. The trial-and-error sensitivity analysis was executed to find the optimal solution. The trials with equally and non-equally spaced prescribed displacements were done. It was found that the model with 4 prescribed displacements and 20 saved steps provide the same quality of results (load-displacement curves and bending moments) as the 10 stages model, but with significantly less storage space and calculation time. Finally, the following non-equally prescribed displacements were adopted as the optimum solution: $0.015D$, $0.03D$, $0.06D$ and $0.10D$.

The calculation speed can also be adjusted by setting the tolerated error of the numerical algorithm in PLAXIS 3D. The default value is 1%, but it can be changed by the user. This parameter is also included in the numerical model of the pile group (denoted as parameter $TOLERANCE$ in Table 3.1).

3.5 Proposed model parameters

Proposed FEM model of the pile group contains 28 different parameters that fully describe pile group configuration, geometry, material properties, loading conditions, numerical algorithm, model domain, boundary conditions and finite element mesh. The model parameters are presented in Table 3.1.

The most of the model geometry parameters have been designed as the multiplicators of pile diameter D, so the whole model geometry is parametrized as the function of the pile diameter D.

Table 3.1: Proposed parameters of the FEM model of laterally loaded pile group. Integer and float are common numerical data types used in many programming languages.

Parameter group	Parameter	Unit	Description	Data type
Pile dimensions	D	L	Pile diameter	Float
	L1		Pile length below ground surface	Float
	L2		Pile length above ground surface	Float
Pile group configuration	N	-	Pile rows (X direction)	Integer
	M	-	Pile columns (Y direction)	Integer
	SX	Diameters	Normalized c-c pile spacing in X direction	Float
	SY		Normalized c-c pile spacing in Y direction	Float
Model domain and discretization	BXFRONT	Diameters	Distance from pile group edge to the model sides	Float
	BZDOWN		Distance from pile bottom to the model bottom	Float
	REF_ZONE_XFRONT		Size of mesh refinement zone, measured from pile group edge	Float
	REF_ZONE_BOTTOM		Distance between the bottom of refinement zone and the model bottom	Float
	SMALL_FE_SIZE		FE size in refinement zone	Float
	BIG_FE_SIZE		Global FE size	Float
Soil and interface parameters	CMODEL	-	Constitutive model index (1-MC model, 2-HS model)	Integer
	SOIL_WEIGHT	F/L^3	Unsaturated soil volumetric weight	Float
	COHESION	F/L^2	Cohesion	Float
	PHI	degrees	Angle of internal friction	Float
	E50_EOED	F/L^2	E_{50}=E_{oed} (Hardening Soil Model)	Float
	EUR_MULTIPLIER	-	E_{ur}/E_{50} (Hardening Soil Model)	Float
	NU	-	Poisson's ratio	Float
	POWERM	-	m (Hardening Soil Model)	Float
	RINTER	-	Interface strength reduction factor	Float
LE model parameters	E_CONCRETE	F/L^2	Young's modulus (Concrete)	Float
	NU_CONCRETE	-	Poisson's ratio (Concrete)	Float
	W_CONCRETE	F/L^3	Volumetric weight (Concrete)	Float
Calculation	TOLERANCE	%	Tolerated error for PLAXIS 3D calculation algorithm	Float
Loading	MAX_D	-	Max displacement (0.XX of pile Diameters)	Float
	ANGLE	degrees	Loading direction angle β	Float

3.6 Model geometry

Domain and boundary conditions

Model boundaries should be far enough from the center of the pile group, so the stress change at the boundary does not affect the model response. Sensitivity analyses were carried out to determine the optimum size of the model. The results are presented in **Chapter 4**.

Because the model geometry will change for different pile group configurations, it was adopted that the distances to the model boundaries remain the same (governed by independent input parameter $BFRONT$ (see Table 3.1), and scaled by pile diameter D). This can be justified with the fact that in the displacement control test, the same level of deformation is "introduced" to the model.

The displacements on the model boundaries were fully fixed in all directions at the bottom of the model, while the displacements on the side planes were limited to vertical direction. According to Fayyazi et al. [50], the pile model can be truncated at the pile bottom. The reason is due to the fact that the lateral pile group response is mostly governed by the top soil layer, in the case of flexible piles. According to [98], pile-soil separation will occur near the top and behind the pile, no deeper than 20% of the pile length (depending on the pile and soil stiffness), but not less than $8D$ [126].

Discretization

FE mesh in PLAXIS 3D is mainly automatically generated, and it is governed by a single parameter *coarseness ratio*, that relates the model domain size and the size of finite elements. Alternatively, average FE size can be used as meshing parameter. PLAXIS 3D does not allow for very precise control of the model discretization. However, it allows the user to define "refinement zones" inside the model volume, where the mesh discretization can be finer. Every refinement zone is controlled using individual coefficient of the refinement. This coefficient relates the size of the FE inside the refined zone with the global FE size. Around the structural model components, such as beams, anchors and interfaces, FE mesh is (automatically) refined by the PLAXIS 3D.

The FE mesh size can be optimized for both accuracy and computational cost based on the analyses of several meshes with different numbers of elements and mesh sizes. In order to optimize the mesh discretization without influencing the model precision, trial analyses were executed on models with different discretization setups. Generally, the optimal grid size is taken to be the largest one that will resolve all the important features of the problem, giving the most economical solution that is still accurate.

Very important fact regarding the size and the quality of the FE mesh is that the finite elements have regular shapes and approximately equal dimensions. Around the pile group, rectangular prismatic mesh refinement zone is adopted (governed by parameter REF_ZONE_XFRONT). The model domain extension around the pile group towards the boundaries was chosen to consist of finite number of pile diameters (scalable with D).

The sketch of the model boundaries, with the position of the refinement zone and governing model parameters is displayed in Figures 3.3 and 3.4.

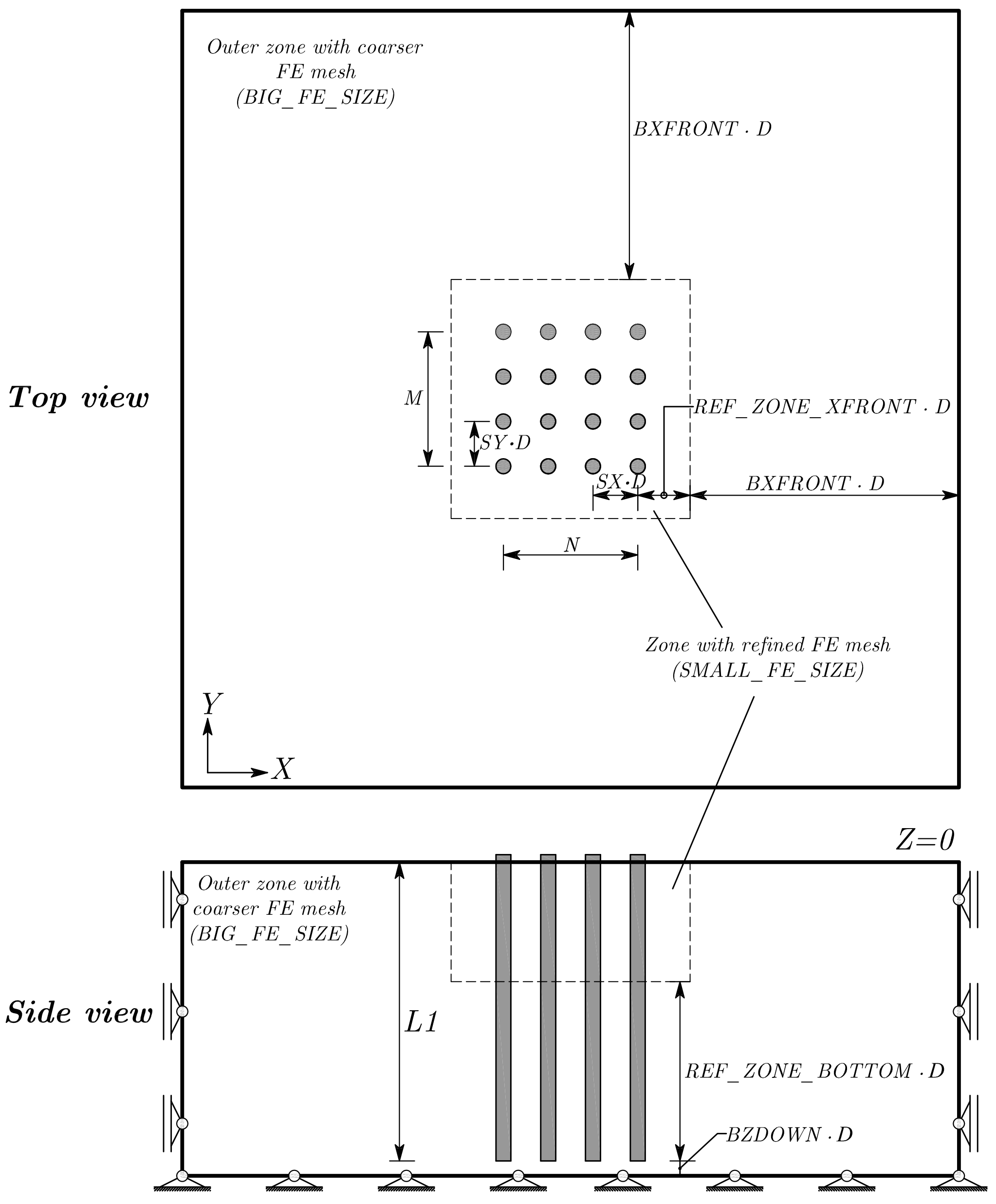

Outer zone with coarser FE mesh (BIG_FE_SIZE)
BXFRONT · D
Top view
M
SY·D
REF_ZONE_XFRONT · D
SX·D
BXFRONT · D
N
Zone with refined FE mesh (SMALL_FE_SIZE)
Y
X
Z=0
Outer zone with coarser FE mesh (BIG_FE_SIZE)
Side view
L1
REF_ZONE_BOTTOM · D
BZDOWN · D

Figure 3.3: Sketch of the model with parametrized geometry, assigned boundary conditions and corresponding model parameters

Postprocessing of the results

Beside the complexity of model generation and simulation itself, data management after calculations was equally challenging, because all raw data had to be reordered and reorganized after each simulation in order to perform its interpretation in a suitable format. In simple words, both data management and post-processing were the equally important tasks as the analysis itself.

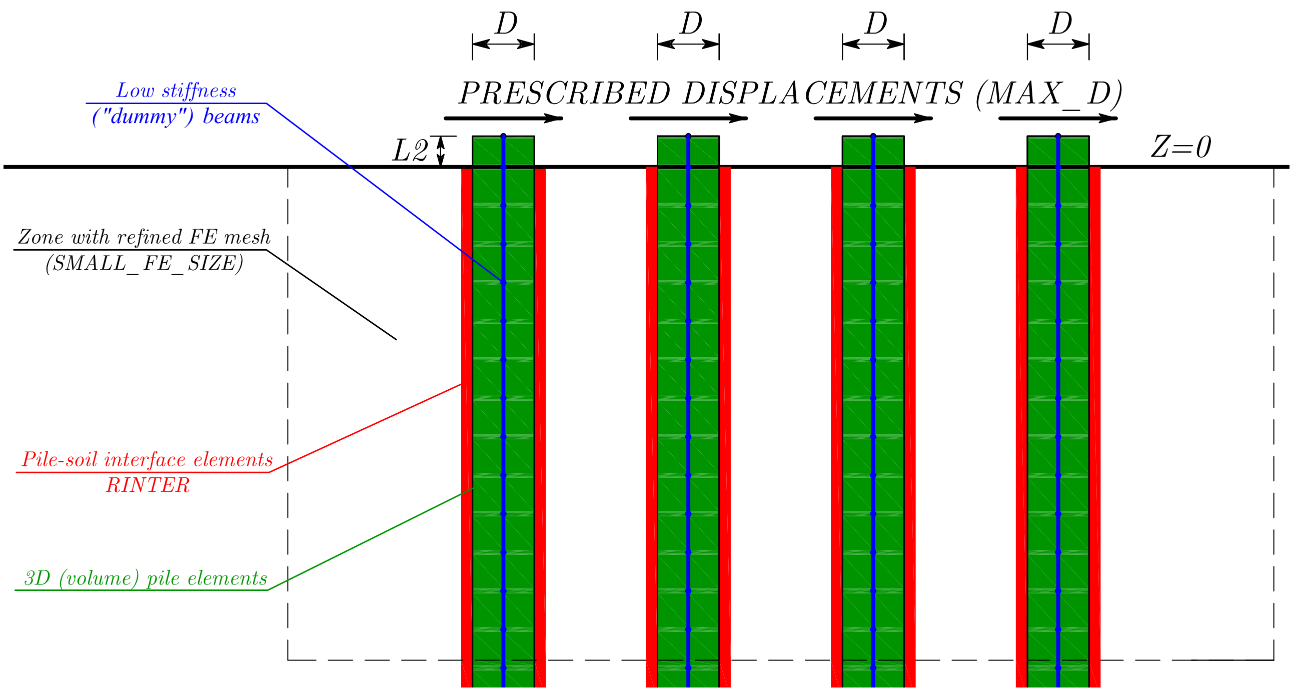

D
D
D
D
PRESCRIBED DISPLACEMENTS (MAX_D)
Low stiffness ("dummy") beams
L2
Z=0
Zone with refined FE mesh (SMALL_FE_SIZE)
Pile-soil interface elements RINTER
3D (volume) pile elements

Figure 3.4: Model parameters in the mesh refinement zone

Maximum displacements, shear forces and bending moments

PLAXIS 3D post-processor allows the extraction of all results in Euclidean XY space. The most important results, such as displacements and bending moments are extracted separately in X and Y direction. However, for the arbitrary loading case, the resultant maximum displacements and bending moments must be calculated separately (Figure 3.5).

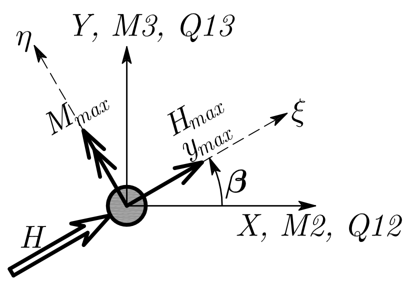


Figure 3.5: Directions of the maximum displacements y_{max}, shear forces H_{max} and bending moments M_{max}. Loading direction is governed by the loading direction angle β. X, Y, $Q12$, $Q13$, $M2$ and $M3$ denotes the directions of displacements, shear forces and bending moments in Euclidean XY space in PLAXIS 3D, respectively.

Based on simple vector algebra, displacements and section forces extracted directly from "dummy" beams in PLAXIS 3D Euclidean XY space (displacements U_x and U_y, shear forces $Q12$ and $Q13$ and bending moments $M2$ and

$M3$), are transformed towards the loading direction. The corresponding axes are given in Figure 3.5. Transformations are done using the following equations:

$$y_{max} = U_x cos\beta + U_y sin\beta \tag{3.6}$$

$$H_{max} = Q12cos\beta + Q13sin\beta \tag{3.7}$$

$$M_{max} = M3cos\beta – M2sin\beta \tag{3.8}$$

Displacements, shear forces and bending moments in loading direction (Equations 3.6, 3.7 and 3.8) are assumed as maximum values and are used as main observation points for the interpretation of the pile group interaction analysis. However, comparison was made on a trial model with the "absolute" maximum values (e.g. $y_{abs,max} = \sqrt{U_x{}^2 + U_y{}^2}$), and the observed differences were about 2%. However, above expressions were adopted as the more straightforward.

Derivation of section forces

The most important step in the post-processing of the results is the evaluation of shear forces in the pile. These forces must be precisely calculated, in order to obtain the load-displacement curves and further evaluate the pile group interaction effects. However, the shear forces extracted from "dummy" beams in PLAXIS 3D are slightly unrealistic ("zig-zag"). This issue was analyzed by Tedesco (2013) [123], who concluded that such behaviour could be associated with the PLAXIS 3D beam elements, that compute the shear forces using the bending moment derivative along the pile length.

In order to evaluate the shear forces properly, the pile displacements and bending moments are approximated (fitted) using the polynomial or spline approximations. After fitting, shear forces can be evaluated by differentiation of the bending moments or displacements. Python [127] scripts written by the author are used for this step.

Data fitting and integrity check

In experimental testing of the laterally loaded piled foundations, the bending moments are commonly measured at a finite number of discrete strain gauges along the pile. In order to obtain the continuous bending moment line, it is common practice to fit the measured data using a smooth fitting curve. Continuity is obtained using different fitting techniques, such as splines and polynomials. Hussien et al. [128] analyzed the optimal positioning of the strain gauges and the different curve fitting techniques (i.e. cubic splines, B-splines, polynomials) for the analysis of laterally loaded piles and the derivation of the p-y curves. Cubic and cubic B-splines were found to be the most appropriate fitting methods for the considered problem. In this study, the same concept was used for the derivation of shear forces from "dummy" beams.

Polynomial fitting uses the least square method to approximate the measurement points, using 3rd to 10th order polynomials. The quality of the fitting is described by R^2 (coefficient of determination). On the other hand, spline fitting is the approximation method that interpolates the polynomial inside subintervals between 2 or 3 points. The most common way of interpolation is cubic spline, where the cubic polynomial is interpolated. B-splines create the continuity between the adjacent spline fittings. One also important parameter in the spline fitting is the number of interior knots that are used for better interpolation. Interior knots are equally spaced inside the single interval. For approximation of displacements, minimum 6th order of the polynomial is recommended, so the 4th derivative is a smooth function.

Trial and error estimations have been done in order to select optimum fitting method for this study. Simple visual inspection proved to be enough for the data integrity check. Even all of these checks can be considered as routine ones, they enforced some important decisions in the whole analysis process. The checks of the pile bending moments, shear forces and displacements were major inspections. The fitting methods that gave the unexpected results have been excluded. The analysis was done on both free head single pile and 2x2 pile group models. It is important to point out that the observation point for load-displacement behaviour is the ground point. After inspection checks, it was found that the spline fitting is the best option for the fitting of the bending moments curves. The polynomial fitting was also tried, but discrepancies that occurred were higher than in the case of spline fitting. It was also concluded that all important parameters should be calculated by differentiating the bending moments, instead of differentiating the displacements.

Following fitting parameters have been selected: B-splines fitting, with 10 interior knots and 5th order spline interpolation (as recommended by Haiderali and Madabhushi 2016 [129]) was adopted.

In order to do that, additional computer scripts were written in order to quickly present the results in a suitable format.

The pile displacements and section forces for free head pile should follow the well-known behaviour pattern, as shown in Figure 3.6:

Verification example for the single pile is presented in Figures 3.7 and 3.8.

In order to calculate the interaction factor for any desired load level, load-displacement curves for each pile were approximated by 10th order polynomials.

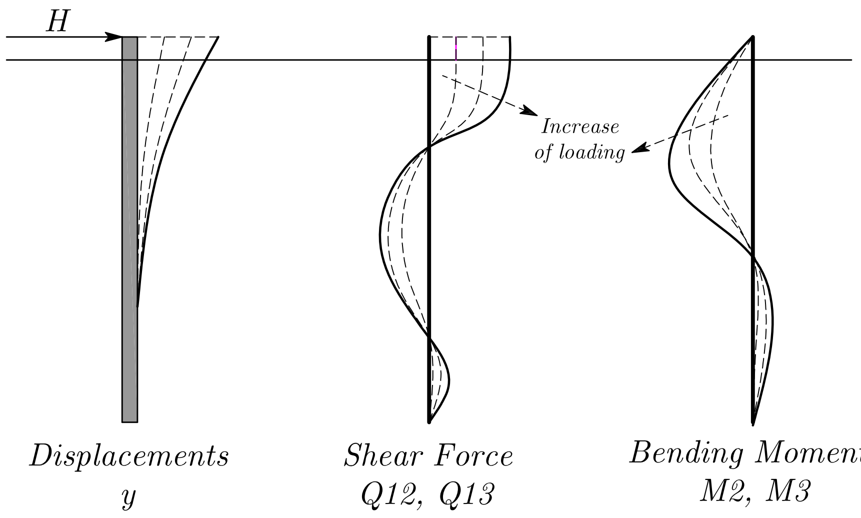


Figure 3.6: Shape of the characteristic diagrams for a free head pile

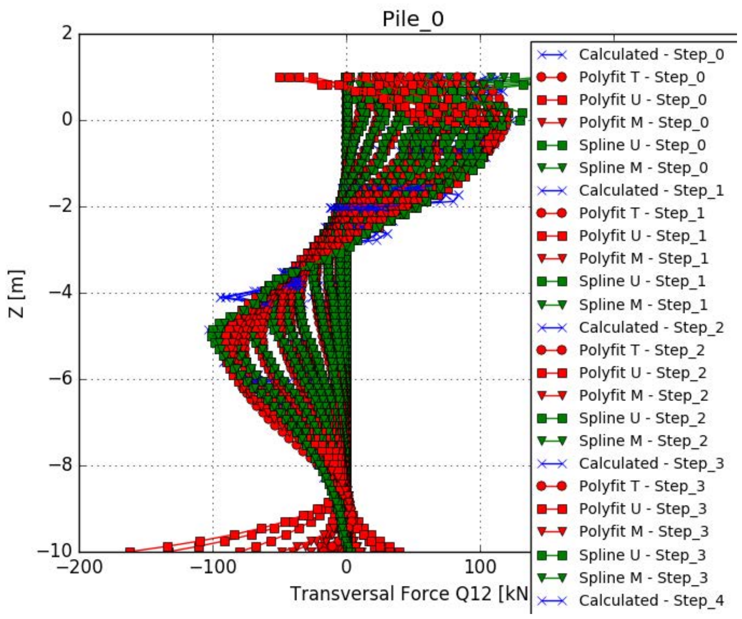


Figure 3.7: Verification of shear forces fitting for a single pile (screenshot from author's computer program). Spline fitting of bending moments and displacements provides more logical results, especially at the end of the pile, where polynomial fitting shows unrealistically high shear forces. The shear forces calculated by the spline fitting and differentiation of bending moments (marked with green triangles) is the most realistic, so this fitting technique was adopted.

The quality of the fitting have been found to be correct, so the fitting error can be neglected.

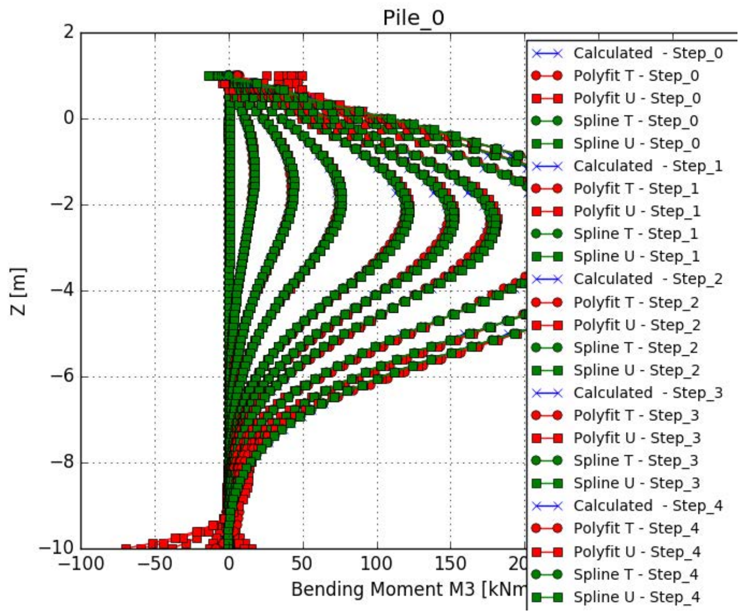


Figure 3.8: Verification of bending moments fitting for a single pile (screenshot from author's computer program). Spline fitting of bending moments and displacements provides more logical results, especially at the end of the pile, where polynomial fitting shows unrealistically high bending moments.

3.7 Automatization of calculation procedures

Parametric study in the Dissertation include multiple FEM simulations, with different combinations of input data. When multiple numerical simulations are needed, one of the most critical "bottlenecks" are the repetitive preprocessing and postprocessing activities. This part of the process is usually prone to errors. Despite the fact that the PLAXIS 3D is user-friendly, the preparation of multiple numerical models of pile groups is very time-consuming.

Scripting is a popular way for the automatization of numerical analysis steps: model generation, calculation, post-processing and analysis of results. Such approach can eliminate the need for manual model input, while ensuring the full control of all simulation features. Scripting features are nowadays implemented in various software packages, such as ABAQUS, PLAXIS 3D, ANSYS, SOFiSTiK etc.

Since 2015, PLAXIS 3D can be operated through a scripts programmed in Python. This feature allows the user to control both the Input and Output modules, both by the direct user input commands on-the-fly or the script files. To establish a connection between Python and PLAXIS 3D, a so-called *boiler-plate* code provided by PLAXIS 3D must always be included and executed at the beginning of the script. This code establishes a connection to the PLAXIS 3D server and imports the necessary libraries. Python commands for every specific action can be found in PLAXIS Command Reference Manual, which comes with the installation of the software. Along with this, required amount of Python syntax knowledge is necessary.

In order to automatize the entire calculation procedure in this study, the broad implementation of original Python scripts was done throughout every numerical analysis step. All scripts were written using the object-oriented paradigm, in order to improve the code readability, error tracking and future improvements. The idea was to fully automate the calculation process, so PLAXIS 3D code becomes a "problem calculator", while the simulation parameters and data analysis are handled by the separate computer programs. This concept cannot increase a single simulation calculation time, neither prevent the conceptual errors in the model, but data handling in all simulation stages (input and output) is substantially faster.

Use of multiple computers

PLAXIS 3D allows for the multiple computers access through the LAN network. In this study, due to the availability of multiple licenses of PLAXIS 3D at the Ruhr University Bochum, it was decided that the use of all of them at the same time would be an optimum solution. To acomplish this task, the set of additional computer programs has been developed by the author. The multiple computer simulation distribution algorithm is implemented in this phase, based on multithreading routine.

The distributed computer analysis has been performed on 4-7 computers (tested on up to 30 computers). Every computer can calculate only one simulation at a time, but as soon as the simulation is finished, the next simulation is started automatically. This process lasts as long as the simulation queue contains at least one simulation. In this manner, if every simulation has the same level of complexity, the total computation time is directly proportional to the number of used computers. Bearing in mind the relatively high cost of the PLAXIS 3D license, as well as the dedicated computers, this solution has provided the optimum use of expensive resources. To maximize the efficiency, computations were mainly done during the night, without distracting the other collaborators.

All developed Python scripts are provided as the **Appendix** at the end of this thesis. All programs are schematically presented in Figure 3.10. Graphical user interface (GUI) is implemented in all modules, in order to simplify the use of program. The short overview of all developed programs is given in the following sections.

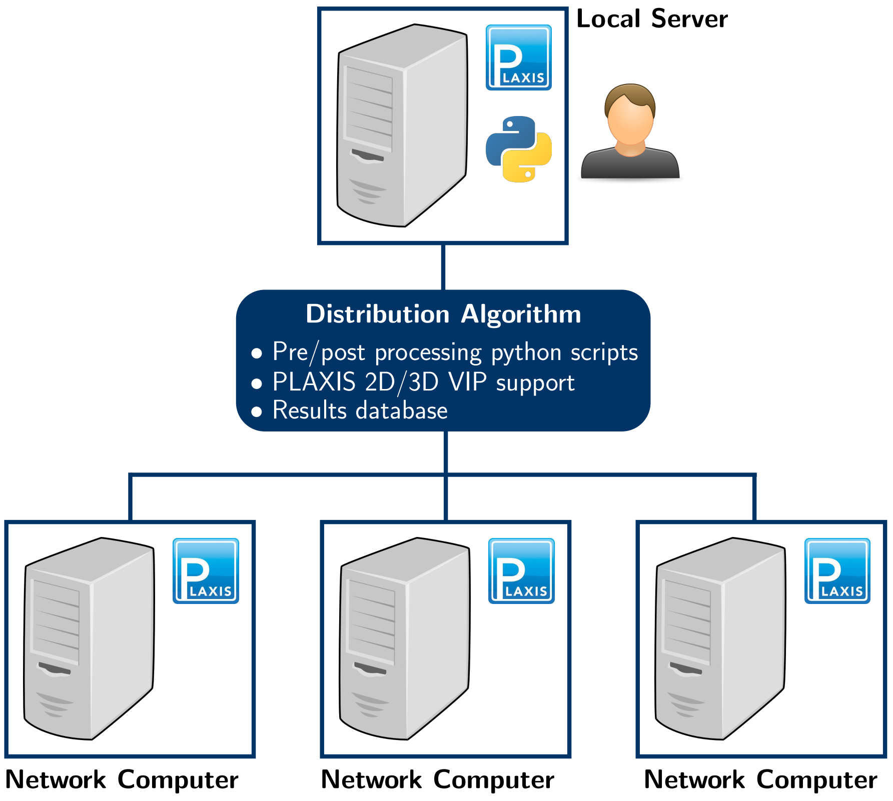


Figure 3.9: Scheme of the computer network for distributed computing solution. Main program is installed on the server computer and controlled by user, while the network computers contain PLAXIS 3D and they are only accessed remotely, using Python scripts

Main Program - Multiple Simulations Processor

Main computer program written in this thesis consists of the following modules:

- Module 1 - Calculation Scenario Maker,
- Module 2 - Simulation Runner,
- Module 3 - System Identification Package,
- Module 4 - Multiple Dataset Postprocessor.

Module 1 - Calculation Scenario Maker

Module 1 features automatic dataset preparation, according to the input parameters. Input data can be imported from Excel file. Multiple simulation input parameters are stored inside *Dataset* object class, that contain multiple *Simulation* objects.

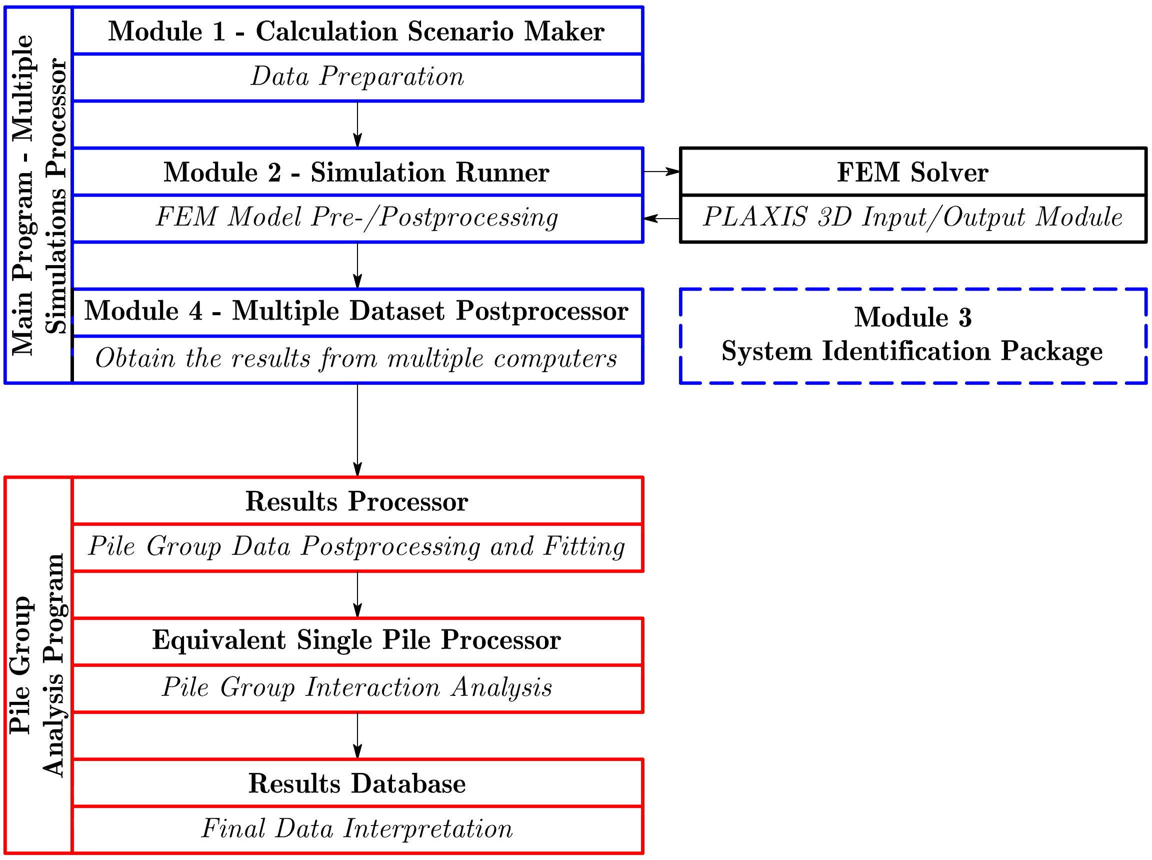

Main Program - Multiple Simulations Processor
Module 1 - Calculation Scenario Maker
Data Preparation
Module 2 - Simulation Runner
FEM Model Pre-/Postprocessing
FEM Solver
PLAXIS 3D Input/Output Module
Module 4 - Multiple Dataset Postprocessor
Obtain the results from multiple computers
Module 3
System Identification Package
Pile Group Analysis Program
Results Processor
Pile Group Data Postprocessing and Fitting
Equivalent Single Pile Processor
Pile Group Interaction Analysis
Results Database
Final Data Interpretation

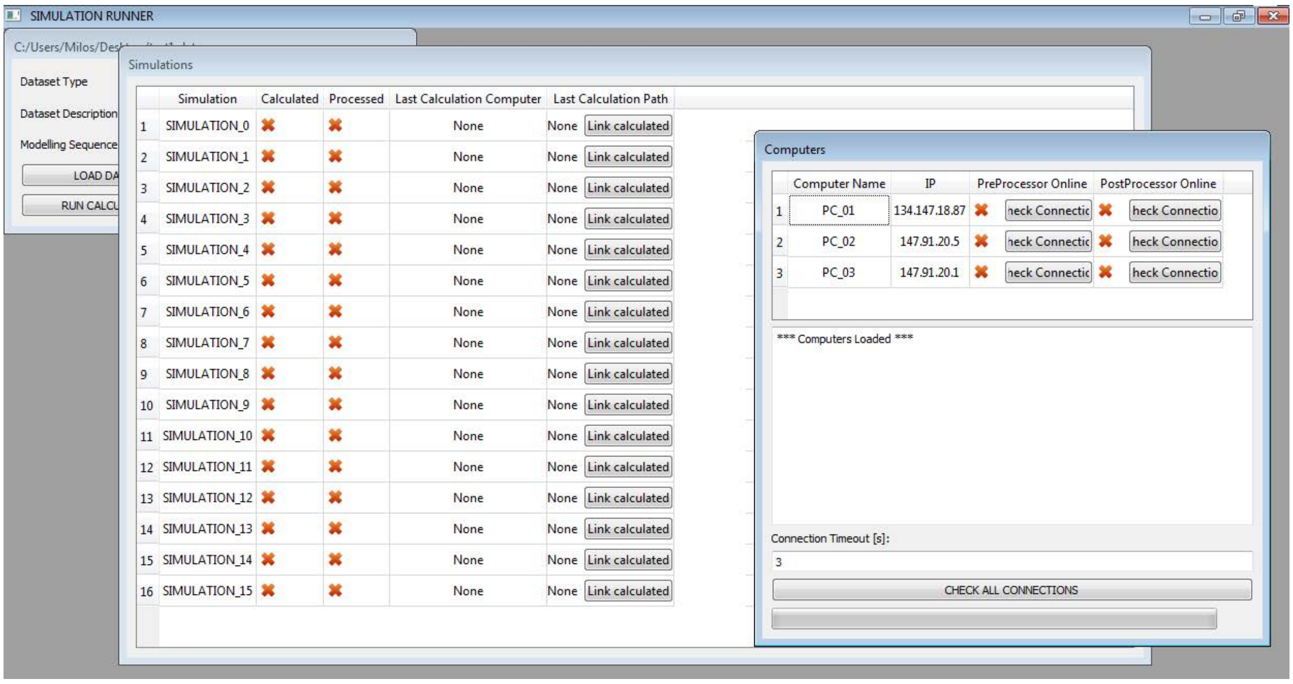
Figure 3.10: Modules of computer programs used for the automatization of numerical simulations (developed by author)

Module 2 - Simulation Runner

Module 2 provides scripts for model preprocessing, running the simulation on multiple computers and the postprocessing of raw simulations. The whole calculation process can be paused and later continued, swapped to a different PC configuration and fully downloaded for archiving. Additional configuration files (such as IP addresses of remote computers, remote folder paths for data storage and PLAXIS 3D connection port names) is obtained through the separate configuration files.

Finally, all completed simulations are saved into the separate directory. All simulations can also be stored on joint network storage device, under unique filename, so the overwriting is avoided. The simulation scenario, as well as the extracted data results, are stored inside the single dataset file. After the numerical simulations are done and the "raw" results are extracted into the results database, further postprocessing of the results is done in the separate program for pile group analysis.

Screenshots of the Module 2 before and during simulation running are shown in Figures 3.11 and 3.12, respectively.



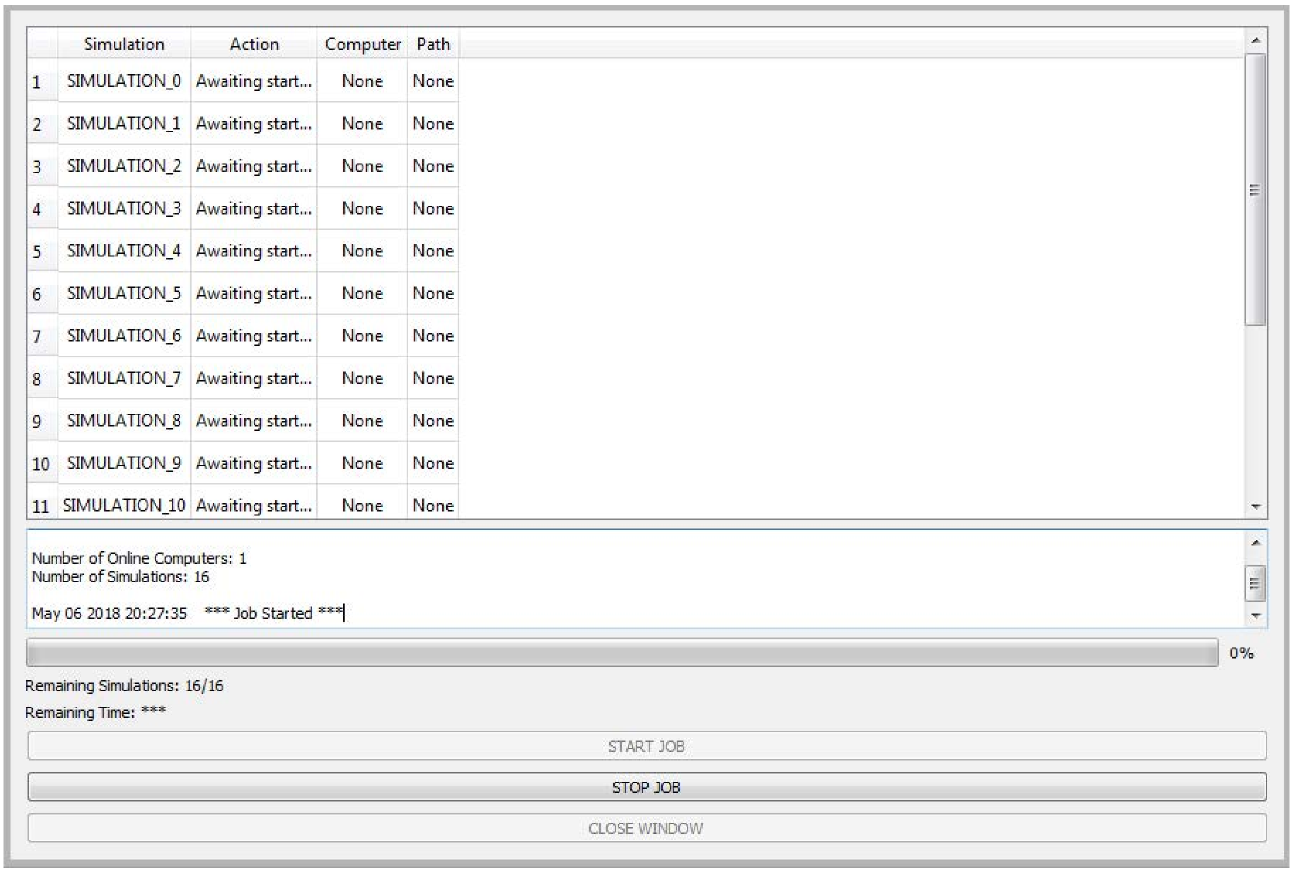
	Simulation	Calculated	Processed	Last Calculation Computer	Last Calculation Path
1	SIMULATION_0	✗	✗	None	None Link calculated
2	SIMULATION_1	✗	✗	None	None Link calculated
3	SIMULATION_2	✗	✗	None	None Link calculated
4	SIMULATION_3	✗	✗	None	None Link calculated
5	SIMULATION_4	✗	✗	None	None Link calculated
6	SIMULATION_5	✗	✗	None	None Link calculated
7	SIMULATION_6	✗	✗	None	None Link calculated
8	SIMULATION_7	✗	✗	None	None Link calculated
9	SIMULATION_8	✗	✗	None	None Link calculated
10	SIMULATION_9	✗	✗	None	None Link calculated
11	SIMULATION_10	✗	✗	None	None Link calculated
12	SIMULATION_11	✗	✗	None	None Link calculated
13	SIMULATION_12	✗	✗	None	None Link calculated
14	SIMULATION_13	✗	✗	None	None Link calculated
15	SIMULATION_14	✗	✗	None	None Link calculated
16	SIMULATION_15	✗	✗	None	None Link calculated

	Computer Name	IP	PreProcessor Online	PostProcessor Online
1	PC_01	134.147.18.87	✗ heck Connectic	✗ heck Connectio
2	PC_02	147.91.20.5	✗ heck Connectic	✗ heck Connectio
3	PC_03	147.91.20.1	✗ heck Connectic	✗ heck Connectio

```
*** Computers Loaded ***
```

Connection Timeout [s]: 3

Figure 3.11: Screenshot of Module 2 before distributed computing is started



	Simulation	Action	Computer	Path
1	SIMULATION_0	Awaiting start...	None	None
2	SIMULATION_1	Awaiting start...	None	None
3	SIMULATION_2	Awaiting start...	None	None
4	SIMULATION_3	Awaiting start...	None	None
5	SIMULATION_4	Awaiting start...	None	None
6	SIMULATION_5	Awaiting start...	None	None
7	SIMULATION_6	Awaiting start...	None	None
8	SIMULATION_7	Awaiting start...	None	None
9	SIMULATION_8	Awaiting start...	None	None
10	SIMULATION_9	Awaiting start...	None	None
11	SIMULATION_10	Awaiting start...	None	None

```
Number of Online Computers: 1
Number of Simulations: 16

May 06 2018 20:27:35    *** Job Started ***
```

0%

Remaining Simulations: 16/16

Remaining Time: ***

Figure 3.12: Screenshot of Module 2 during simulations

Model "template" concept

Module 2 allow for the use of so-called "template" modelling concept. This concept is used mostly in order to speed up the multiple simulation analysis. In this concept, initial numerical model is formed, and the most important, the finite element meshing (very slow step), is done. Then the model is saved and used as the basis (start point) for the generation of other models. This is possible when the difference between the model is in the order of material parameters or calculation phases. The pre-processing Python script in this case is written to open the template model, perform adjustments of the initial template model, and then to calculate and save a new model under another filename.

Because the remote computer cannot be fully accessed through the LAN connection, before calculating the model template case, the existence of the TEMPLATE.p3d files inside the local/remote folders on each PC must be ensured.

Module 3 - System Identification Package

The another part of developed program includes routines for global model sensitivity analysis (Variance-Based [130]) and parameter back calculation. Mathematical algorithms such as Proper Orthogonal Decomposition using Extended Radial Basis functions (POD-ERBF and ERBF [131]) are included for replacing of the FEM model with computationally less expensive *metamodel*, using limited number of performed FEM simulations. For solving the optimization problem and parameter back calculation, population-based Particle Swarm Optimization algorithm [132] is implemented within the program.

Pile Group Analysis Program

The main part of computer program, written by the author within this thesis, is the pile group postprocessing script, that consists of several phases:

- Loading of the raw pile group simulation,
- Extraction of results and their splitting to each pile (pile separation algorithm),
- Approximation of all pile results using B-splines,
- Calculation of y_{max}, Q_{max} and M_{max}
- Creation of load-displacement curves for each pile and their approximation using the polynomials for further easier determination of interaction factors for any displacement level.

All results are extracted for each loading step. The multidimensional data arrays were used to store and manipulate the large amount of data. The data export into Excel tables is provided as well, in order to simplify the data management. After postprocessing is done and results are inspected, there is no need for raw simulation data anymore.

Equivalent single pile

The calculation of the most important pile group results is based on the combination of the results for the pile group and the equivalent single pile. Pile is equivalent to a pile group if pile diameter, length, stiffness, head fixity and surrounding soil are the same. Equivalent single pile calculations are done in separate dataset, and then the results are linked to the appropriate simulation.

3.8 Concluding remarks

Based on the presented literature review and critical approach to presented methodologies of FEM modelling, numerical model is defined to simulate the pile group behaviour under lateral loading. Total of 28 model parameters were introduced to account for all aspects of considered engineering problem. Computer programs for automatization of the entire process of numerical analysis, including the use of multiple computers, were developed by the author. Additional computer programs were developed and verified for the fast postprocessing of the simulation results.

Based on the available results in the literature, the proposed numerical model is validated in the next **Chapter 4**.

4 Numerical Model Validation

In order to assess the capabilities of the proposed FEM numerical model to accurately predict the pile group response in PLAXIS 3D, model calibration and validation were done. Validation was based on the back-calculation of the small-scale centrifuge pile group test.

The model validation procedure consisted of the following phases:

- Constitutive model calibration using the results of laboratory tests.
- Domain and FE mesh size selection. In this phase, model domain and the size of the finite elements were chosen through the simplified sensitivity and convergence analyses, in order to optimize precision of the results and speed of numerical simulation.
- Validation of numerical model by comparing with the available experimental results. Trial and error analysis was used to match the experimental results with the model response.

4.1 Model validation - Kotthaus 1992.

Quasi-static load test in the geotechnical centrifuge at 50g, done by Kotthaus (1992) [21], was used for model validation. Results for a single pile and the pile row at $3D$ spacing were available. Load-displacement curves at the ground level were used as the main observation parameter for comparison with the experimental results.

Centrifuge modelling may be considered to be the best alternative to the full scale tests, because the stress and strain conditions at similar locations in the centrifuge model and the prototype are the same. For parametric studies, centrifuge modelling can provide data for understanding the behaviour of the pile groups at low costs. In the case of centrifuge experiment, the most important goal is to preserve the representative stress states on the small scale model, due to the fact that the soil stiffness and strength is, in general, stress dependent.

The Kotthaus experimental setup was prepared by sand rainfall method, to achieve the desired relative sand density [21]. Experimental aluminium piles were fabricated as hollow tubes. Strain gauges along the pile axis were used for the determination of bending moments. Load cell at the pile top allowed for the calculation of pile group displacement. Free head pile top conditions were enforced using the screws that allowed for the rotation of pile head, but enforced the same pile top displacements. Therefore, the whole experimental setup followed the displacement control setup. In order to increase the pile-soil friction inside the dense sand, soil particles were glued onto the pile shaft. Because of the experimental setup, the point of loading application was above the ground level.

The section properties and the experimental setup model geometry are given in Table 4.1. The outside pile diameter was 30 mm, and the pipe wall thickness was 2 mm. Equivalent centrifuge model parameters are calculated using the scaling laws [133].

Table 4.1: Centrifuge model dimensions [21]. Prototype model corresponds to real concrete pile (between C20/25 and C25/30 concrete classes)

	Centrifuge model	Prototype scale	Real structure
D (m)	0.030	1.50	1.50
L1 (m)	0.600	30.00	30.00
L2 (m)	0.085	4.25	4.25
t (m)	0.002	0.10	
d (m)	0.026	1.30	
E (MPa)	70000	70000	30508
I (m^4)	1.73E-08	1.08E-01	0.2485
EI (kNm^2)	1.2130	7581448	7581448

FEM model geometry was adopted to mimic the centrifuge model. Free head pile with prototype size of 4.25+30m and the diameter of 1.5m was modelled.

Constitutive model calibration

For the accurate numerical simulation of geotechnical problem, first step is selection and proper calibration of the selected soil constitutive model. The available experimental soil triaxial testing results for the dense sand, denoted as *Normensand 942e* ($C_U = 2.08$, $D_{10} = 0.12$mm, $D_{50} = 0.23$mm, $e_{max} = 0.914$ and $e_{min} = 0.583$), that was used in centrifuge study by Kotthaus (1992), were used for Hardening Soil model calibration.

The effective friction angle of the dense sand used in experimental study was initially assumed to be 41 degrees. Oedometer modulus was initially adopted as equal to 28 MPa, based on the study of the same sand by Kayalar 2012 [134].

However, the trial and error adjustments of the laboratory determined values had to be done in order to better match the measured pile group response. This alteration can be justified by the fact that the stress states around the laterally loaded piles are not fully matched by conventional triaxial tests.

Finally adopted constitutive model parameters are given in Table 4.2.

FE mesh

After trial and error analysis, the global size of finite elements (outside the refinements zone) was adopted as $3D$. Smaller sizes of $1D$ and $2D$ were also investigated, but model precision was not justified with increased computation time. The size of the finite elements inside the refinement zone has shown to be more important for the model response. Final size of the elements inside the refinement zone was adopted as $1D$.

Due to the fact that global finite element size was $3D$, the domain size was adopted to be scalable by 3, in order to preserve the equality of the finite element dimensions inside the mesh. Adopted model geometry and mesh parameters are given in Table 4.2.

Results of model validation

After a series of trial and error simulations, satisfactory agreement between computed and measured response was obtained both for the single pile (Figure 4.1) and pile row models (Figures 4.2 - 4.4). The final calibrated model parameters are displayed in Table 4.2.

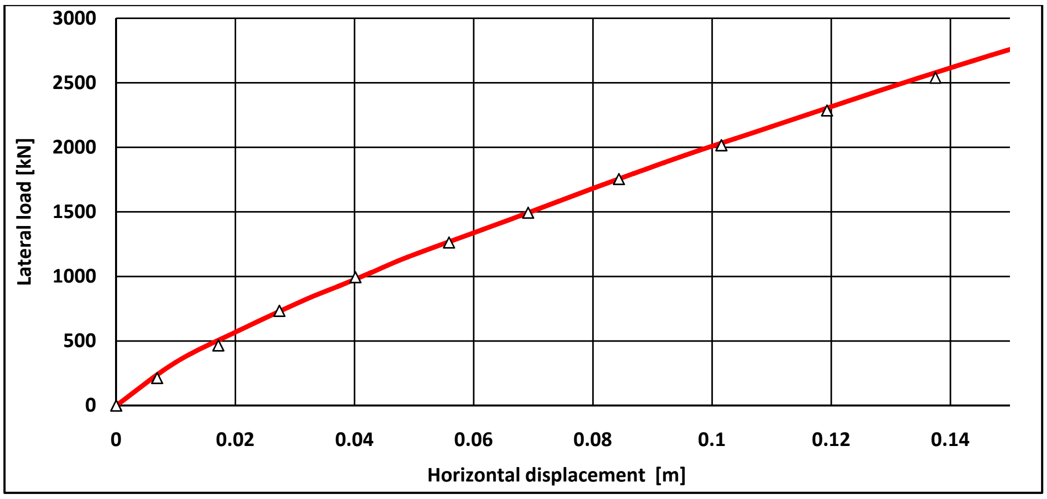


Figure 4.1: Measured (triangles) vs. computed (red line) load-displacement response - single pile

Table 4.2: Final model parameters after back-calculation of experimental results. Model parameters are explained in the previous Chapter.

Parameter	**Value**
BXFRONT	$21.5D$
BZDOWN	$1D$
REF_ZONE_XFRONT	$3.5D$
REF_ZONE_BOTTOM	$12D$
SMALL_FE_SIZE	$1D$
BIG_FE_SIZE	$3D$
SOIL_WEIGHT	$16.6\ kN/m^3$
COHESION	0
PHI	35
E50_EOED	$28800\ kN/m^2$
EUR_MULTIPLIER	4
NU	0.25
POWERM	0.47
RINTER	0.654
E_CONCRETE	$30508\ kN/m^2$
NU_CONCRETE	0.2
W_CONCRETE	$25\ kN/m^3$
TOLERANCE	1.5%

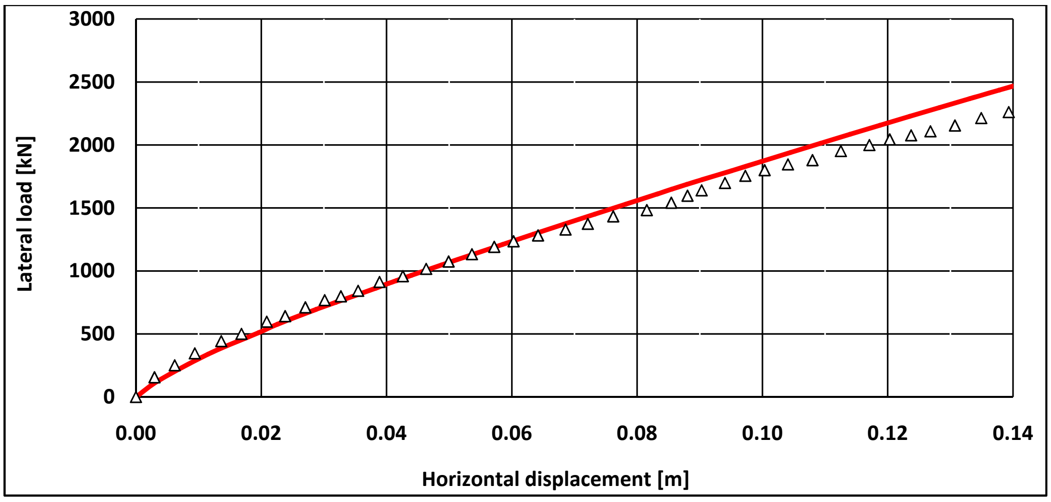


Figure 4.2: Measured (triangles) vs. computed (red line) load-displacement response - piles in a row - front pile

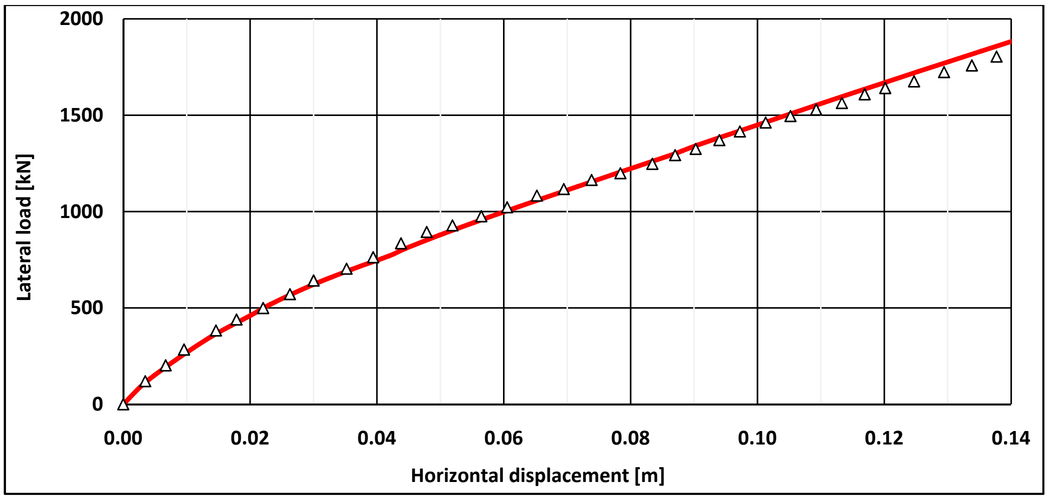


Figure 4.3: Measured (triangles) vs. computed (red line) load-displacement response - piles in a row - front pile

4.2 Concluding remarks

Presented results of the single pile and pile row validation show satisfactory match with the experimental results in the domain of interest. The overall performance of the proposed numerical model is therefore considered to be acceptably accurate for the parametric study of the problem, despite the slight discrepancies.

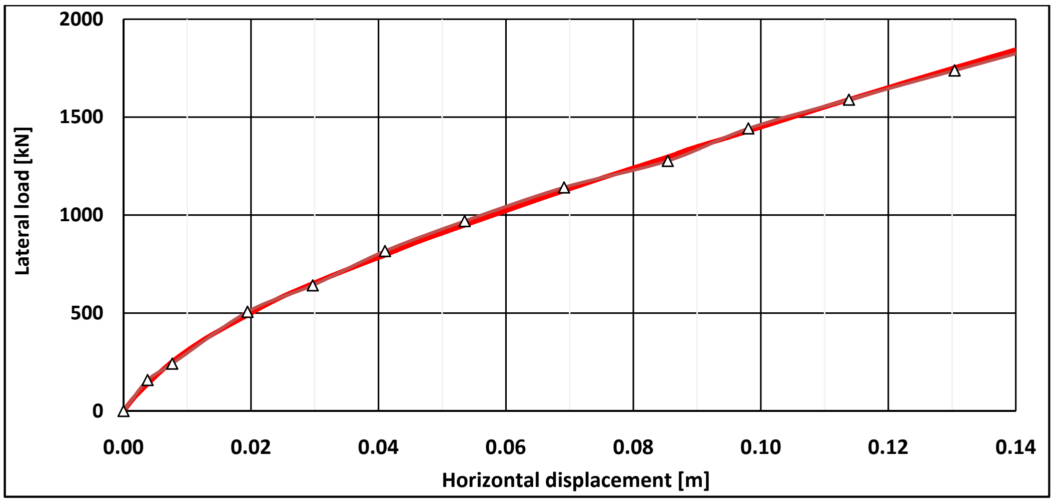


Figure 4.4: Measured (triangles) vs. computed (red line) load-displacement response - piles in a row - front pile

5 Parametric Study and Results Interpretation

The extensive parametric study using the synthetic soils was carried out to assess the influence of the arbitrary horizontal loading direction on the lateral response of different pile groups.

Adopted numerical model parameters

In all simulations, a pile elastic modulus of 30.5 GPa was adopted to simulate the most common concrete class used in the routine engineering practice. Regarding the pile length, Duncan [126] showed that for the common pile–soil stiffness ratios, piles are considered long when their lengths exceed 6–14D. In this study, pile diameter D=1m was adopted. The length of L_1=20m (L/D=20) was adopted to provide the flexible pile behaviour. Finally, L_2=0.25m was taken as small as possible above the ground level to avoid numerical instabilities caused by the high concentrated forces at the place of monitoring the horizontal displacements. The adopted numerical model parameters are elaborated in Table 5.1.

Problem parameters

The following major problem parameters, according to presented apriori sensitivity analysis, were selected for this study: loading direction, soil type, pile group configuration and spacing. All these parameters are further explained and their range is discussed, as follows. They are illustrated in Figure 5.1.

Loading direction

Loading direction was simulated by varying the inclination angle of the loading at the pile tops in horizontal plane.

Soil type

The focus on the sand soil types was made to reflect the usual scope of the sand types from the literature. Great distinction between loose and dense sand was adopted in order to compare the soil stiffening effects to the pile group

Table 5.1: Adopted numerical model parameters (based on the parameters of the validated numerical model)

Parameter	Value
BXFRONT	21.5D
BZDOWN	1D
REF_ZONE_XFRONT	3.5D
REF_ZONE_BOTTOM	12D
SMALL_FE_SIZE	1D
BIG_FE_SIZE	3D
E_CONCRETE	30500
NU_CONCRETE	0.2
W_CONCRETE	25 kN/m^3
TOLERANCE	1.5%

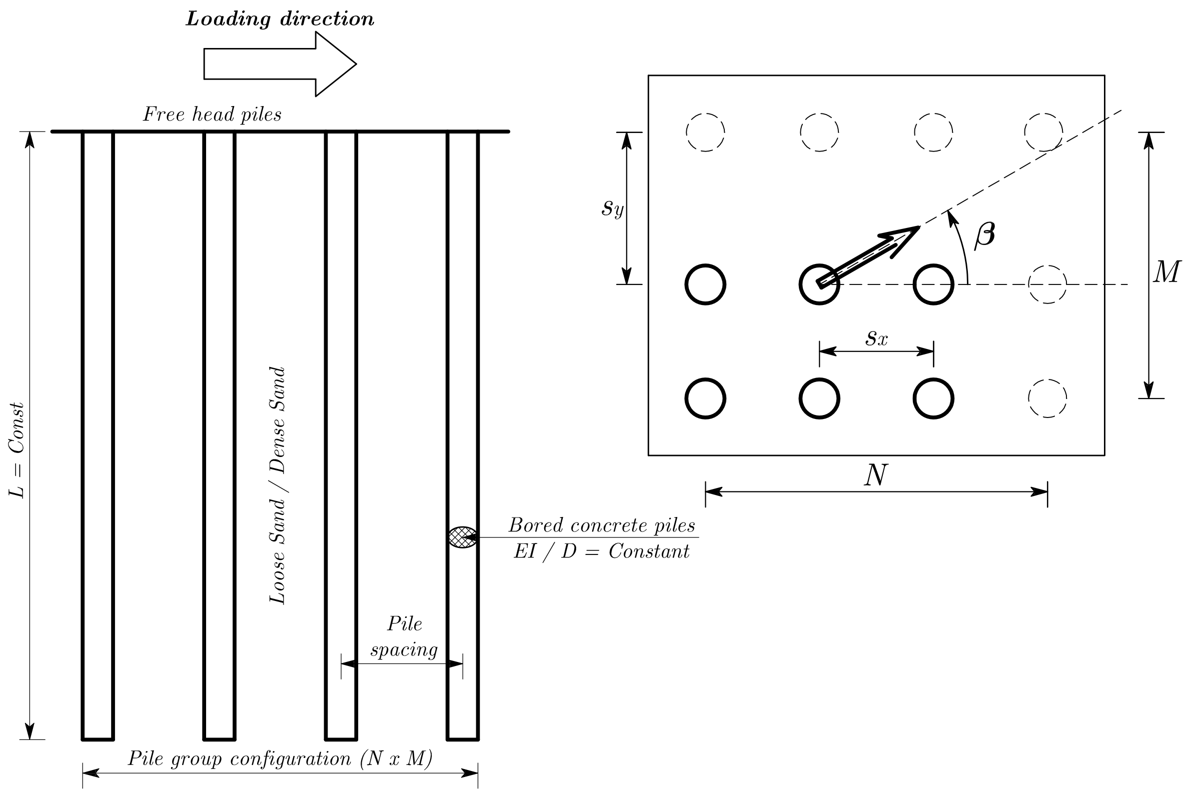


Figure 5.1: Input parameters for parametric study

interaction mechanism. Brown et al. (1988) [48] noted that the shadowing effect was more pronounced in sand compared with stiff clay, and given that this was based on the same pile group setup, it must be considered well-founded. Constitutive model parameters for loose and dense sand are given in Table 5.2. Soil parameters are adopted based on recommendations given in Zarev, 2016. It is important to point out that the most sensitive parameters are chosen as significantly different of each other, in order to better describe the influence of different soil types on model response.

Table 5.2: Hardening Soil model parameters for the validated numerical model, compared with the synthetic soil parameters

Model parameter	Model Validation	Parametric Study	
	Dense Sand	Loose Sand	Dense Sand
Volumetric weight (kN/m^3)	16.6	16	18
Cohesion c (kN/m^2)	0	0	0
Angle of internal friction φ (degrees)	35	30	42
Dilation angle ψ (degrees)	5	0	12
Poisson's ratio ν (-)	0.25	0.25	0.33
E_{50} (kN/m^2)	28800	15000	30000
E_{oed} (kN/m^2)	28800	15000	30000
E_{ur} (kN/m^2)	115200	60000	120000
R_{inter} (-)	0.654	0.65	0.65
m (-)	0.47	0.5	0.5

Pile group configuration and spacing

Pile groups with different pile arrangements (configurations) and center-to-center spacings were analyzed. Squared pile groups were investigated in the first stage, and rectangle-shaped pile groups were investigated in the second stage. Equivalent single piles were also simulated, in order to calculate the pile group efficiency.

As mentioned before, the symmetry conditions could not be used. In order to effectively analyze and discuss the interaction effects, as well as to limit the calculation time, it was decided to limit the total number of piles inside the pile group to max 10 (up to 3x3 configuration).

The results of the parametric study have been evaluated in order to determine the governing factors of the interactions inside the laterally loaded pile group. The characteristic pile group interaction properties have been plotted.

From the practical standpoint, it is more common case that the configuration of the group (N, M) can be different (square, rectangular, circular etc.), while the spacing between the piles is kept on the lowest possible level.

As given in [75], pile group spacing smaller than $3D$ is not recommended by the USA authorities (FHWA [135] and AASHTO [136]), due to difficult pile installation. According to Das [102], the minimum pile center-to-center spacing in practice is $2.5D$, and in ordinary situations, it is about 3-$3.5D$. Even starting from $3D$ spacing, the costs of the pile cap become very large, while 5D spacing is considered as an extreme case [75].

Calculation Scenarios

Previously described and validated FEM model was used for the parametric study. The model input parameters inside the parametric study slightly differ from the validated model, as shown in Table 5.2. Mainly, all validated model parameters are the same as in the parametric study, except for the soil.

Several scenarios of the different pile group configurations were investigated. In the process of selection of pile group configurations to be investigated, total pile cap size was limited, and large cap area cases were eliminated as unrealistic. The summary of study scenarios with all varied model parameters are given in Table 5.1.

Results Catalogue

The results of conducted parametric study are summarized in the following Results Catalogue, in the form of pile interaction factors for each pile and maximum bending moments for each pile inside the pile group, with respect to loading direction. Full lines denote the results for loose sand, and dashed lines denote results for dense sand, respectively.

Pile positions on diagrams are numbered according to the Figure 5.2

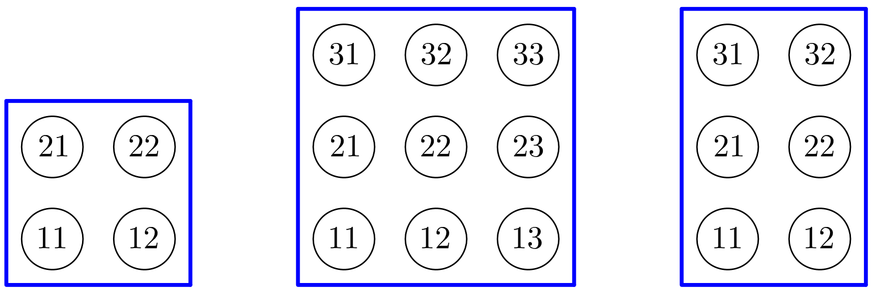


Figure 5.2: Pile numbering

Table 5.3: Parametric study scenarios

PG Shape	Analysis Scenario	N x M	Sx (D)	Sy (D)	Loading Direction (degrees)
Square	1	2 x 2	2	2	0 / 15 / 30 / 45
		3 x 3	3	3	
Rectangle	2	2 x 2	2	3	0 / 15 / 30 / 45 / 60 / 75 / 90
			2	4	
			2	5	
			3	4	
			3	5	
			4	5	
	3	2 x 3	2	2	
			2	3	
			2	4	
			3	2	
			3	3	
			3	4	
			3	5	
			4	2	
			4	3	
			4	4	
			4	5	
			5	3	
			5	4	
			5	5	
	4	3 x 3	2	3	
			2	4	
			2	5	
			3	4	
			3	5	
			4	5	

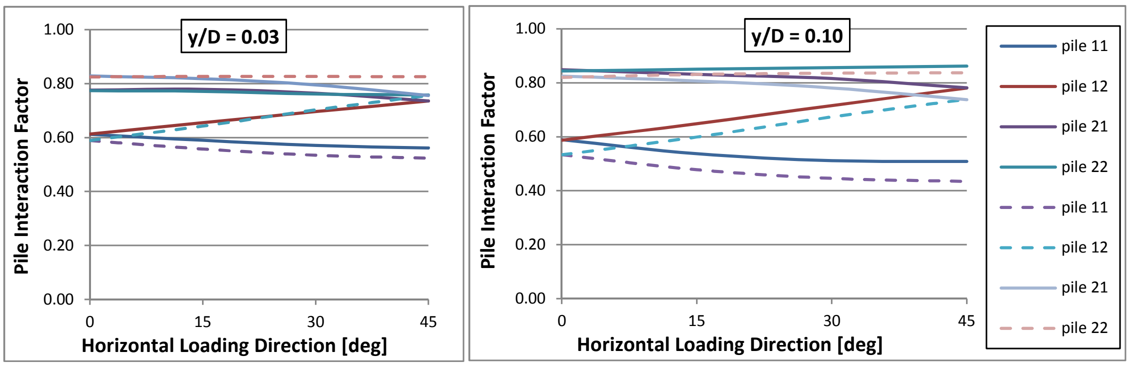


Figure 5.3: Pile Interaction Factors for 2x2 pile group (s_x=$2D$, s_y=$2D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

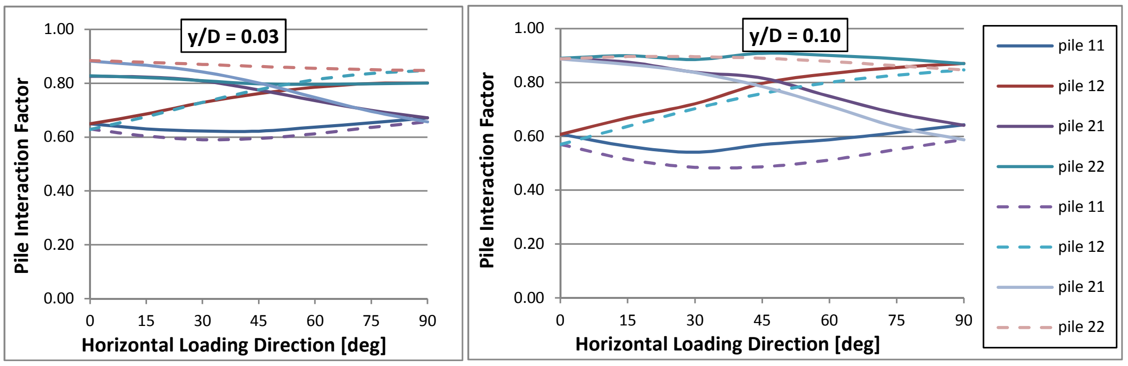


Figure 5.4: Pile Interaction Factors for 2x2 pile group (s_x=$2D$, s_y=$3D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

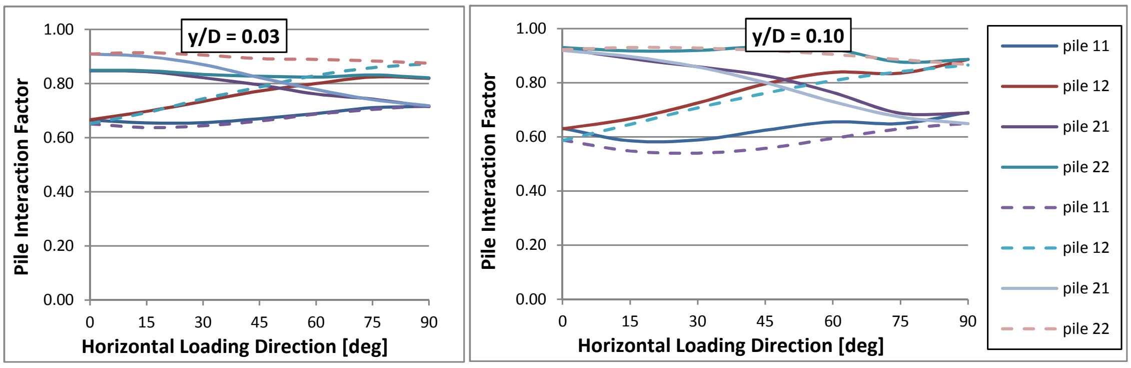


Figure 5.5: Pile Interaction Factors for 2x2 pile group (s_x=$2D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

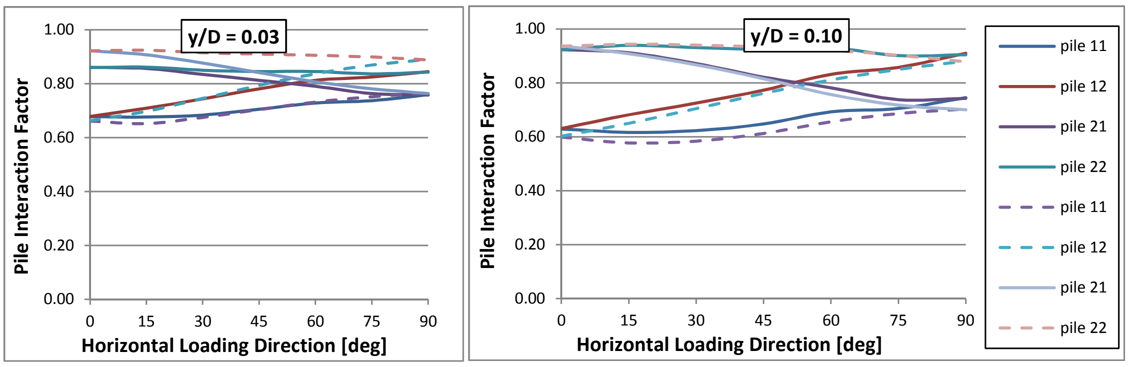


Figure 5.6: Pile Interaction Factors for 2x2 pile group (s_x=$2D$, s_y=$5D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

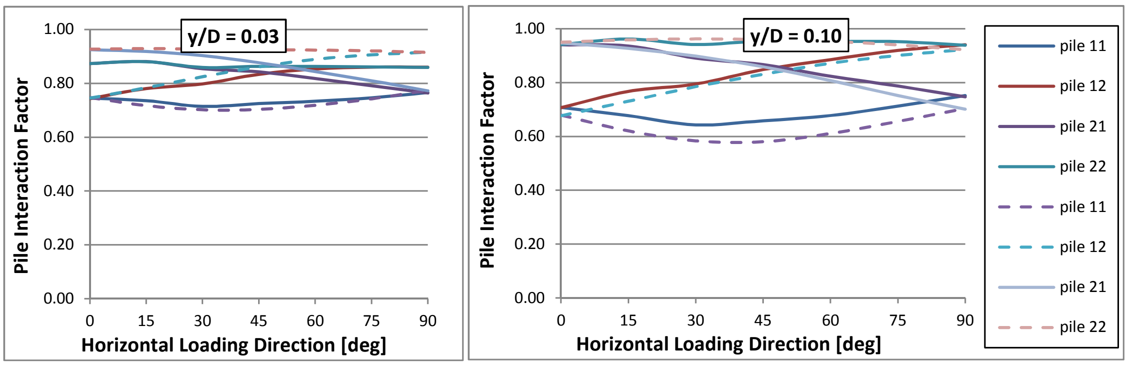


Figure 5.7: Pile Interaction Factors for 2x2 pile group (s_x=$3D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

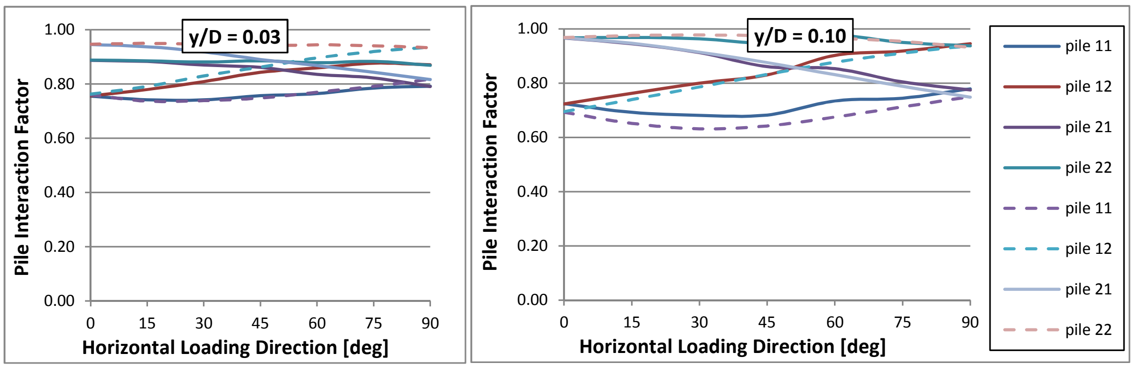


Figure 5.8: Pile Interaction Factors for 2x2 pile group (s_x=$3D$, s_y=$5D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

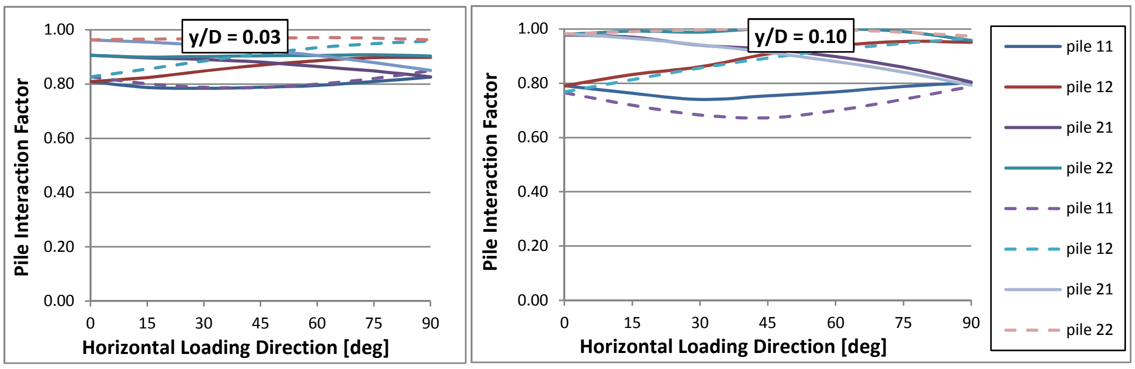


Figure 5.9: Pile Interaction Factors for 2x2 pile group (s_x=4D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

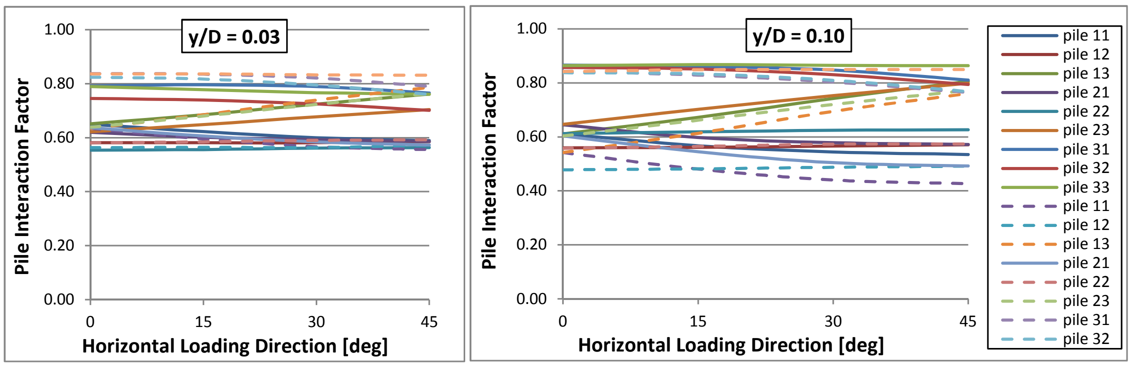


Figure 5.10: Pile Interaction Factors for 3x3 pile group (s_x=3D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

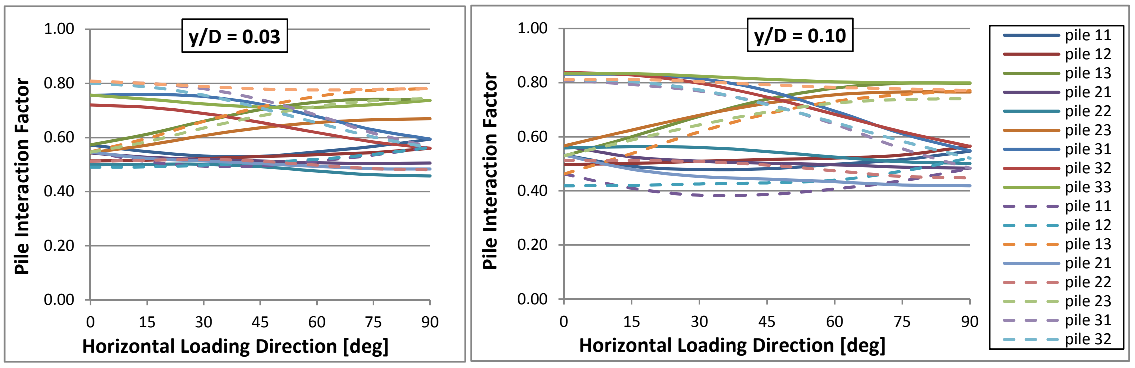


Figure 5.11: Pile Interaction Factors for 3x3 pile group (s_x=2D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

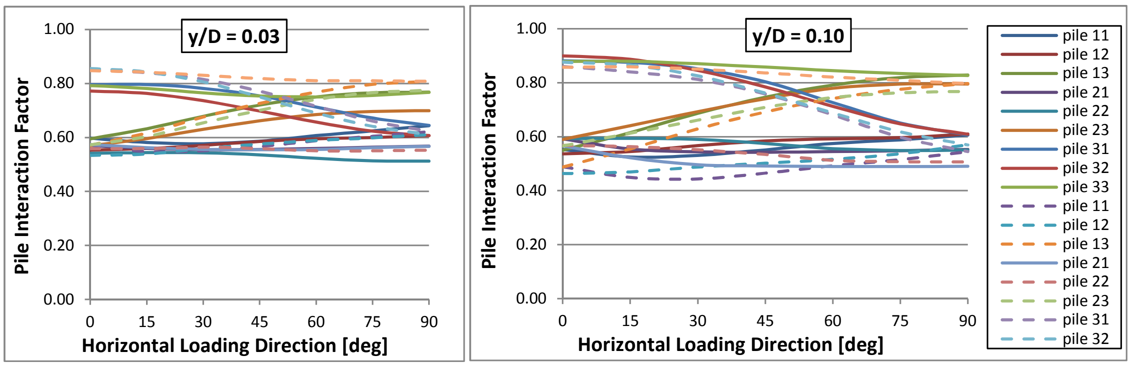


Figure 5.12: Pile Interaction Factors for 3x3 pile group (s_x=$2D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

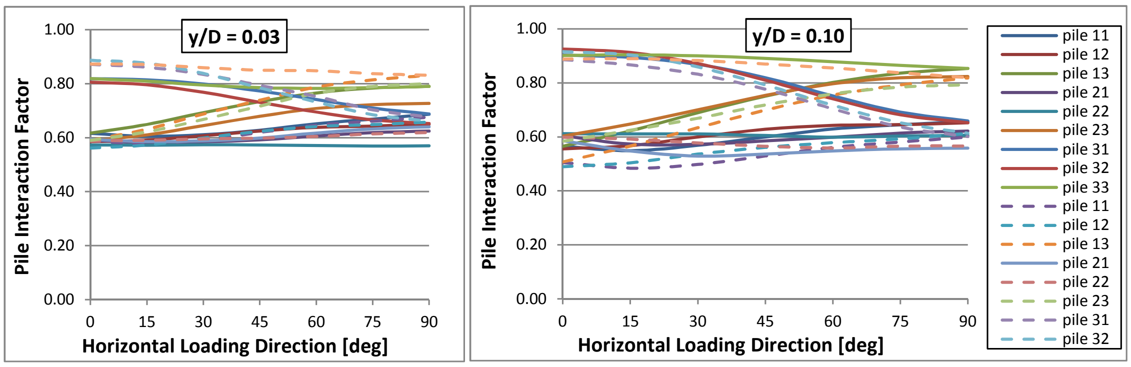


Figure 5.13: Pile Interaction Factors for 3x3 pile group (s_x=$2D$, s_y=$5D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

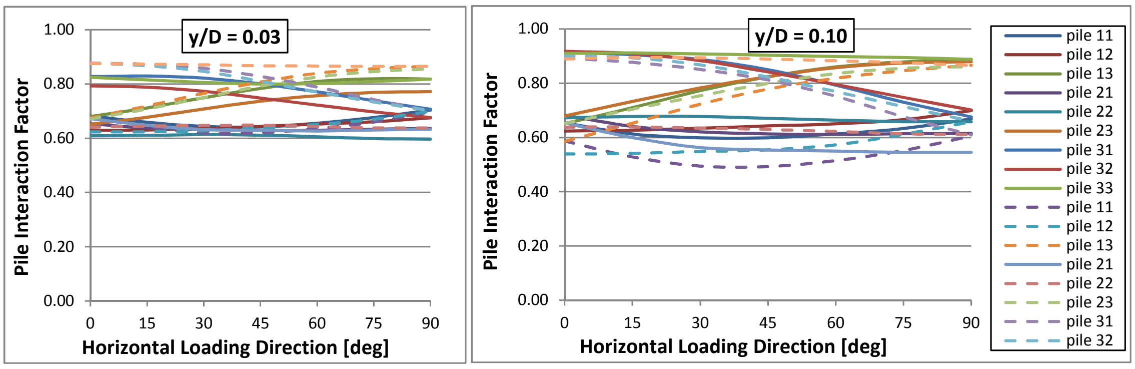


Figure 5.14: Pile Interaction Factors for 3x3 pile group (s_x=$3D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

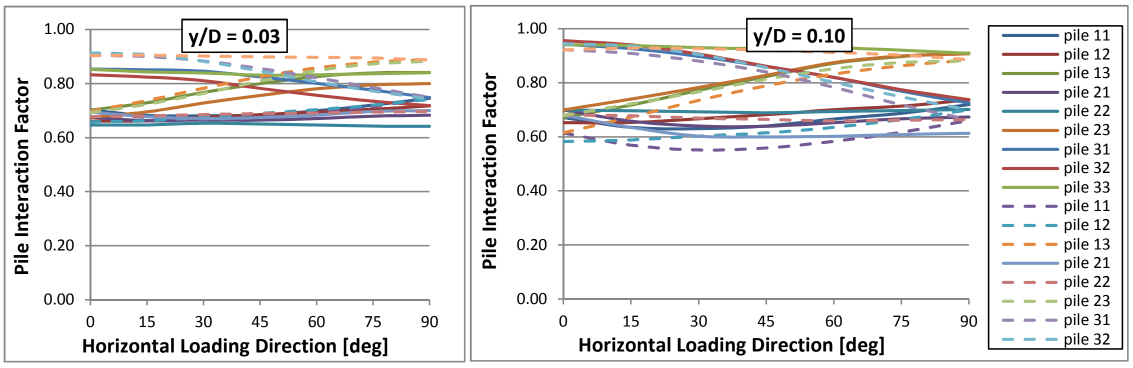


Figure 5.15: Pile Interaction Factors for 3x3 pile group (s_x=3D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

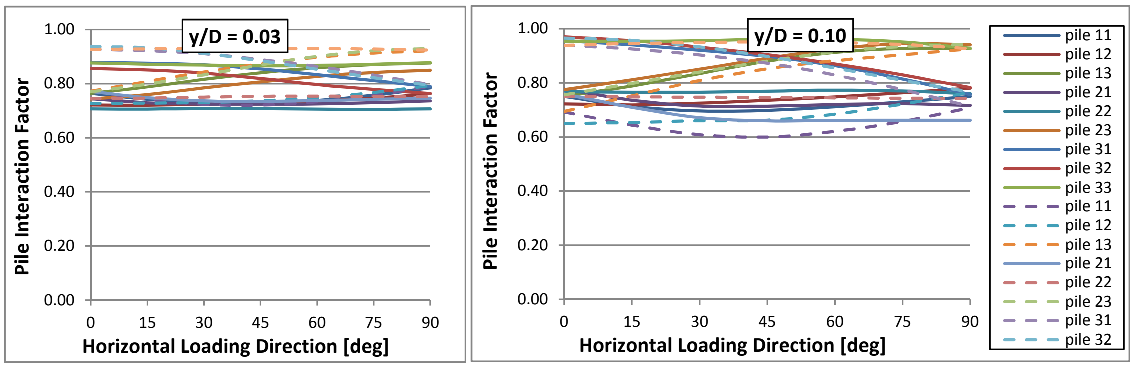


Figure 5.16: Pile Interaction Factors for 3x3 pile group (s_x=4D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

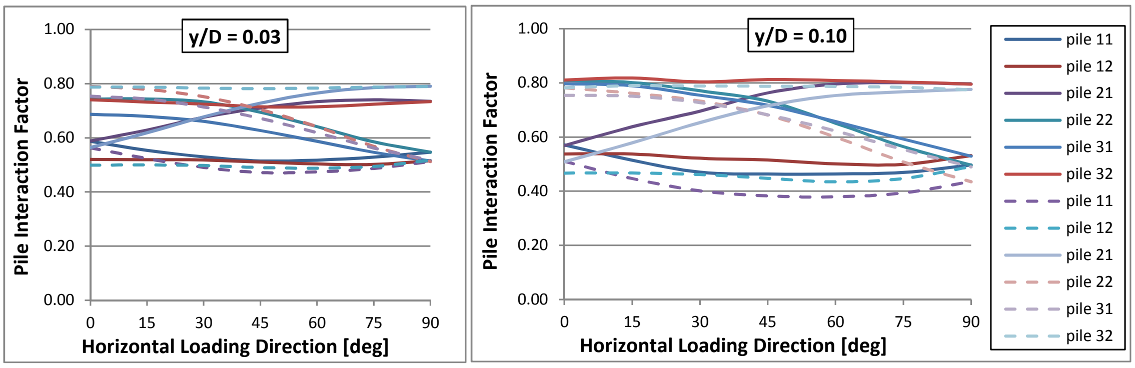


Figure 5.17: Pile Interaction Factors for 2x3 pile group (s_x=2D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

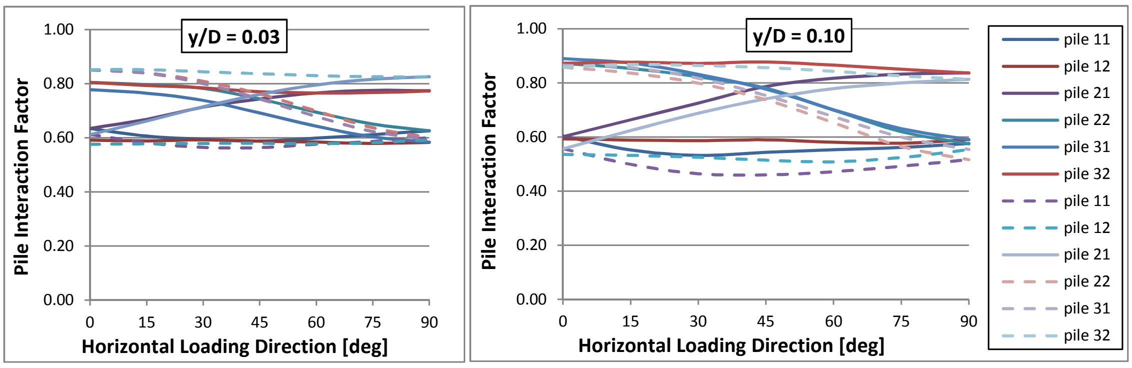


Figure 5.18: Pile Interaction Factors for 2x3 pile group (s_x=2D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

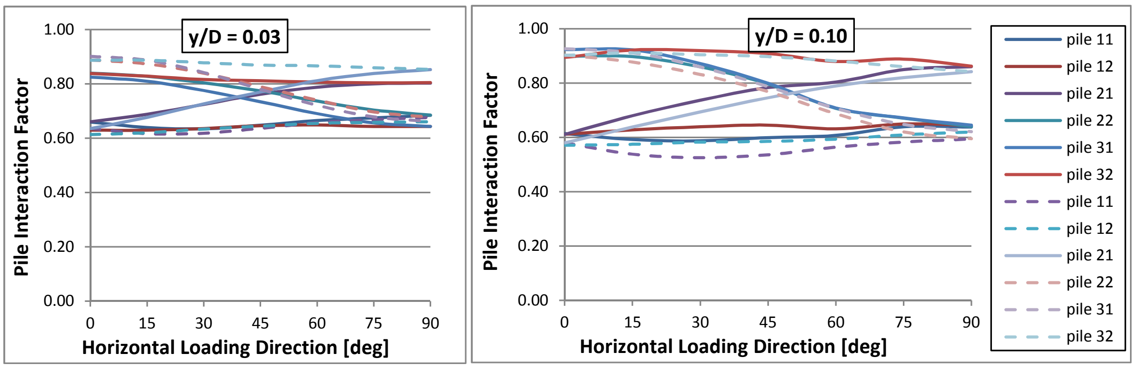


Figure 5.19: Pile Interaction Factors for 2x3 pile group (s_x=2D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

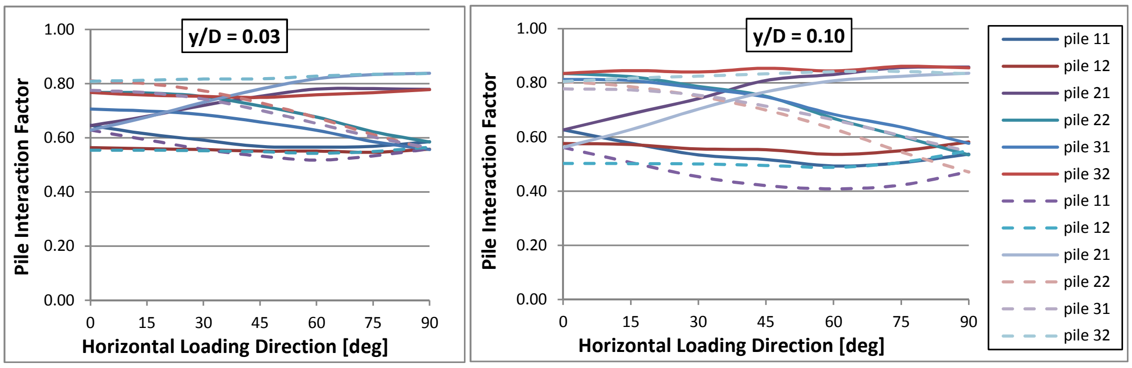


Figure 5.20: Pile Interaction Factors for 2x3 pile group (s_x=3D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

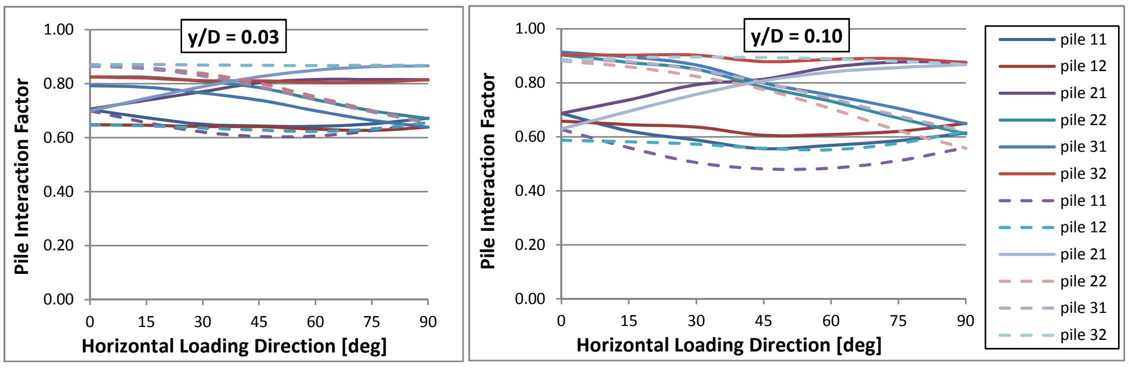


Figure 5.21: Pile Interaction Factors for 2x3 pile group (s_x=3D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

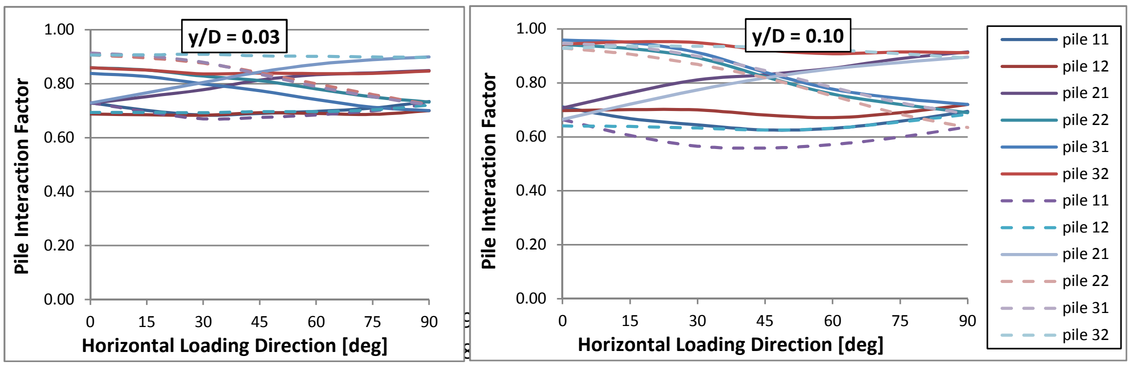


Figure 5.22: Pile Interaction Factors for 2x3 pile group (s_x=3D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

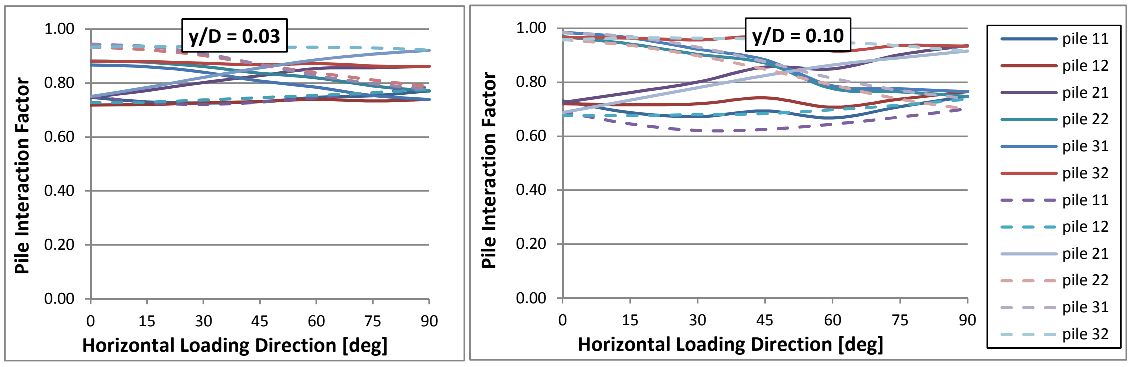


Figure 5.23: Pile Interaction Factors for 2x3 pile group (s_x=3D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

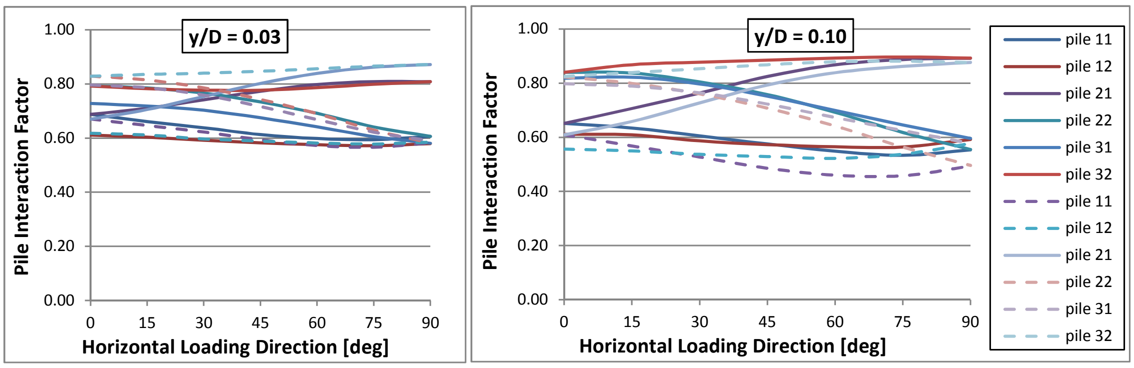


Figure 5.24: Pile Interaction Factors for 2x3 pile group (s_x=4D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

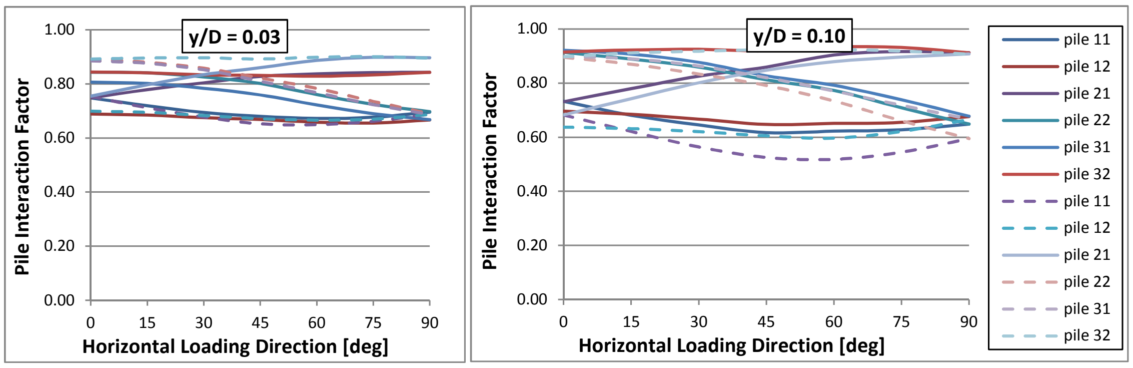


Figure 5.25: Pile Interaction Factors for 2x3 pile group (s_x=4D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

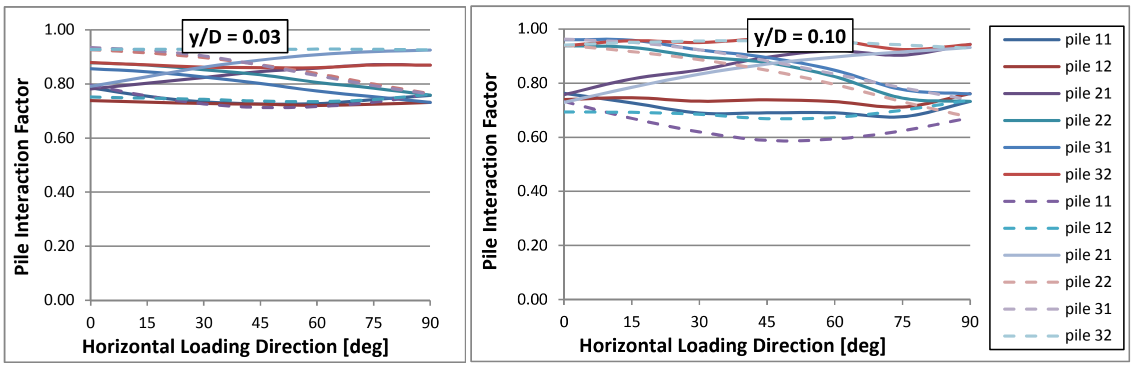


Figure 5.26: Pile Interaction Factors for 2x3 pile group (s_x=4D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

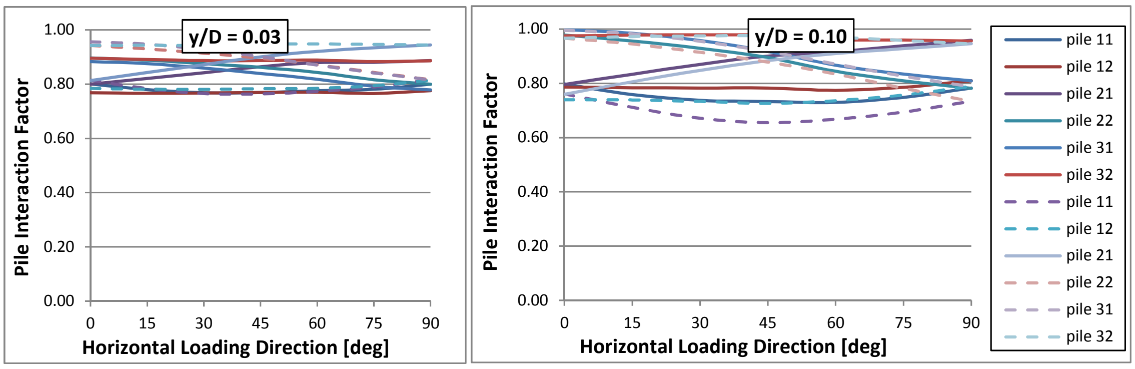


Figure 5.27: Pile Interaction Factors for 2x3 pile group (s_x=$4D$, s_y=$5D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

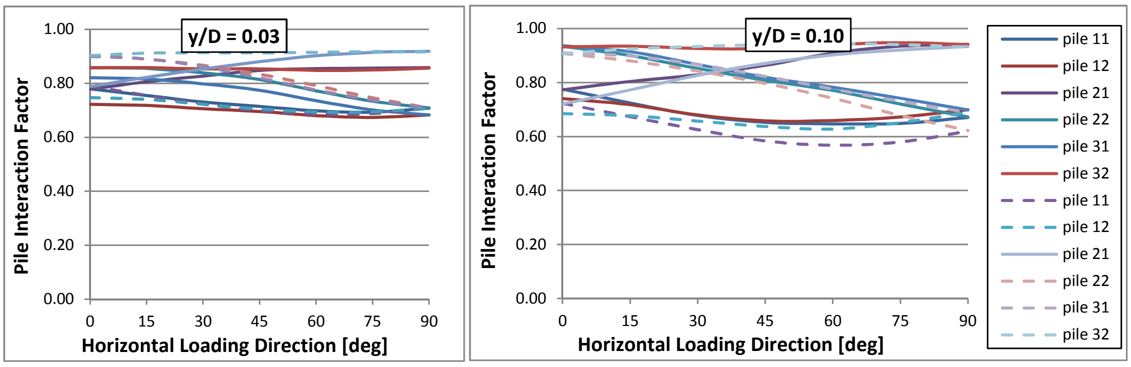


Figure 5.28: Pile Interaction Factors for 2x3 pile group (s_x=$5D$, s_y=$3D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

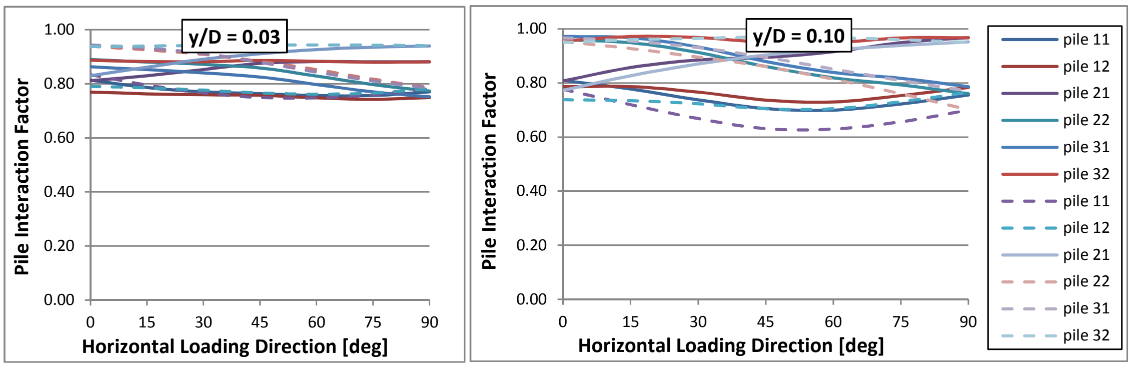


Figure 5.29: Pile Interaction Factors for 2x3 pile group (s_x=$5D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

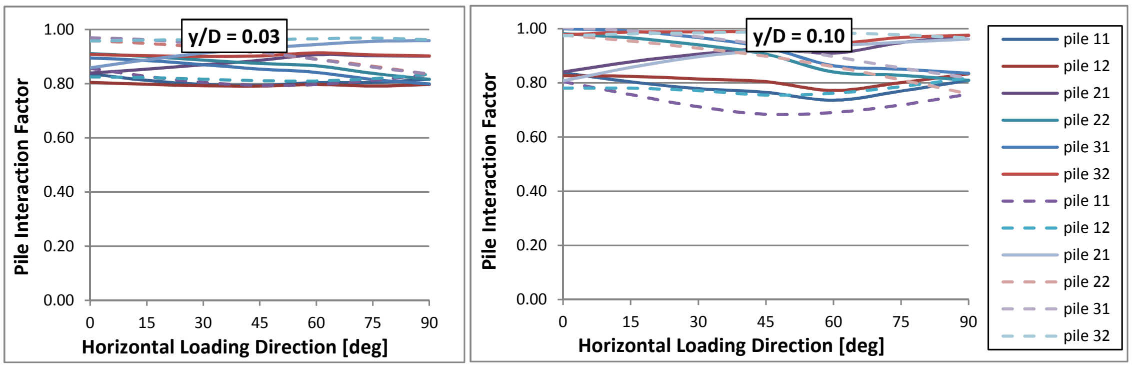


Figure 5.30: Pile Interaction Factors for 2x3 pile group (s_x=5D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

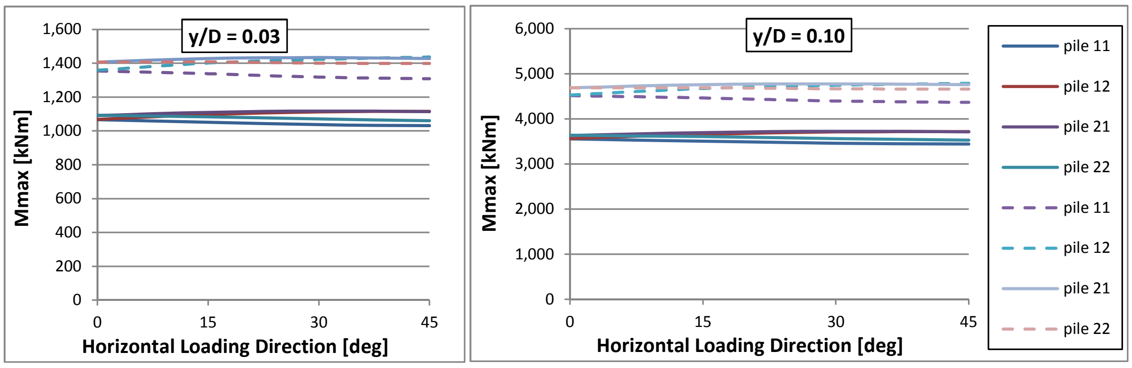


Figure 5.31: Maximum Bending Moments for 2x2 pile group (s_x=2D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

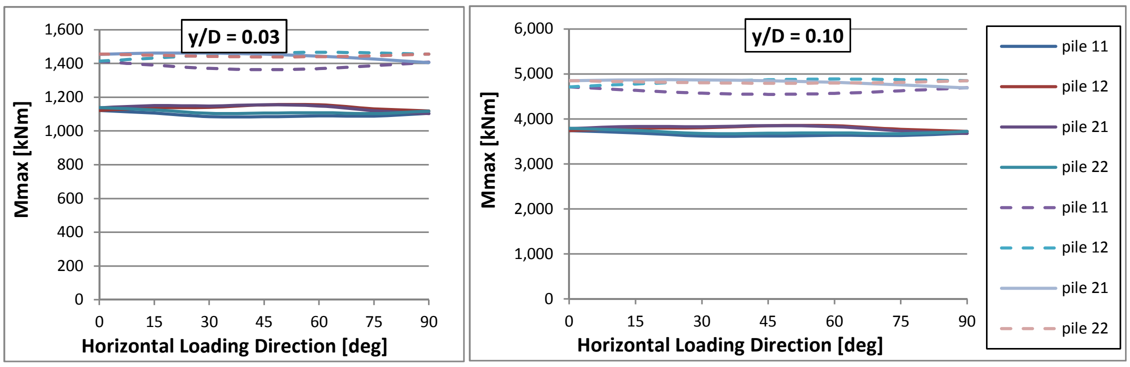


Figure 5.32: Maximum Bending Moments for 2x2 pile group (s_x=2D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

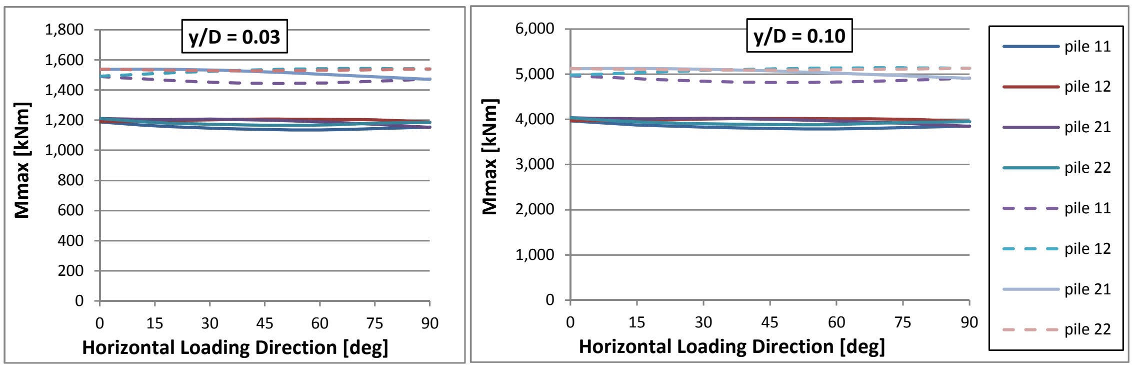


Figure 5.33: Maximum Bending Moments for 2x2 pile group (s_x=2D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

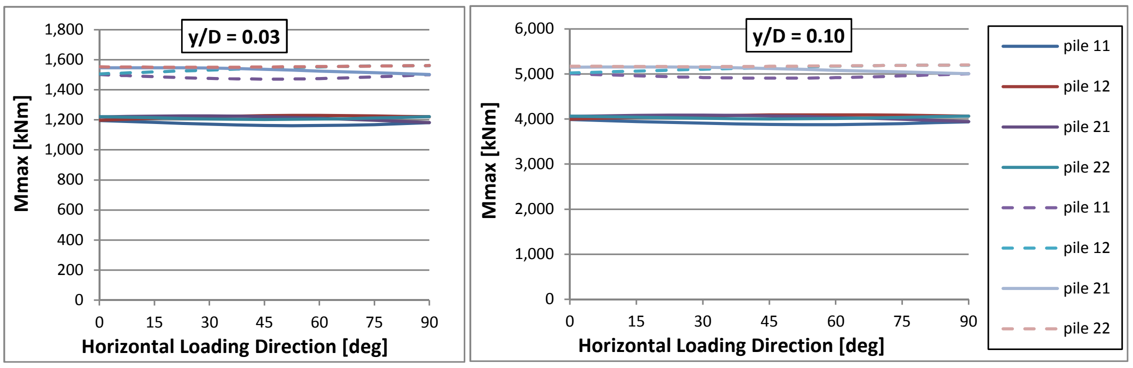


Figure 5.34: Maximum Bending Moments for 2x2 pile group (s_x=2D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

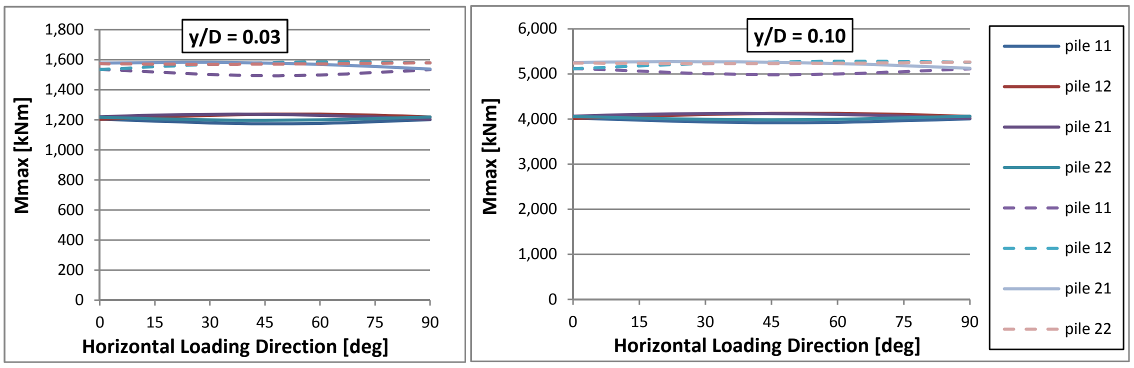


Figure 5.35: Maximum Bending Moments for 2x2 pile group (s_x=3D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

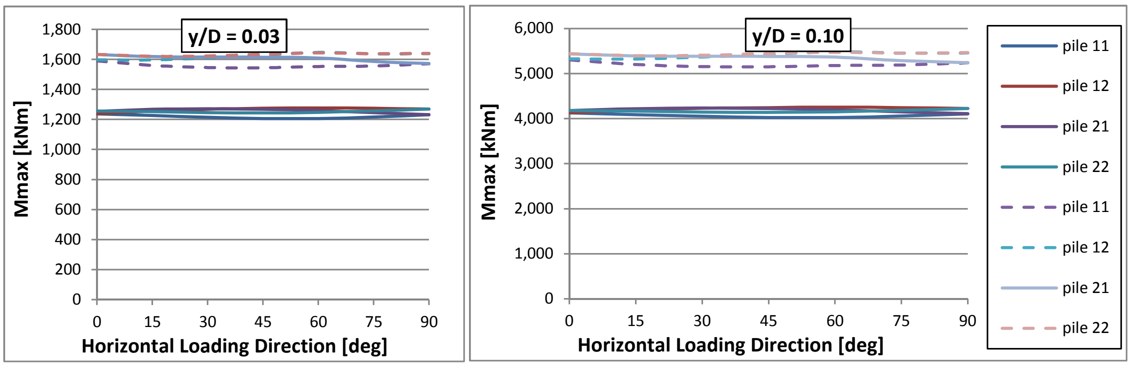


Figure 5.36: Maximum Bending Moments for 2x2 pile group (s_x=3D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

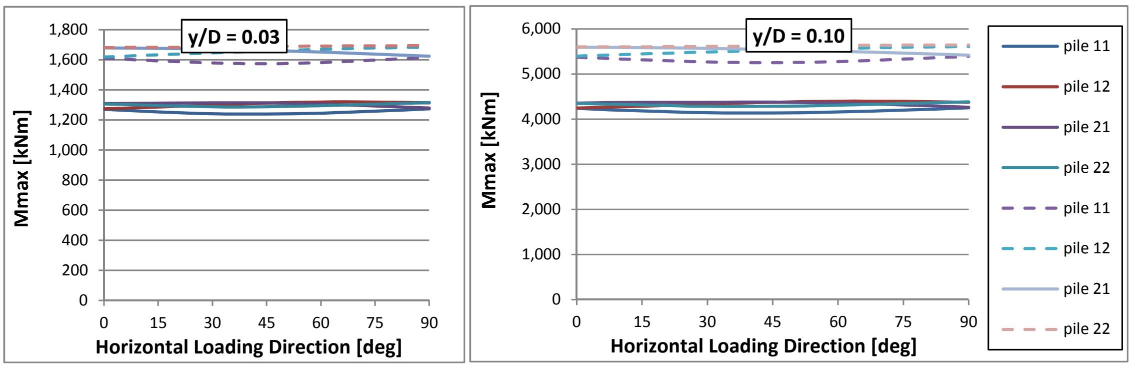


Figure 5.37: Maximum Bending Moments for 2x2 pile group (s_x=4D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

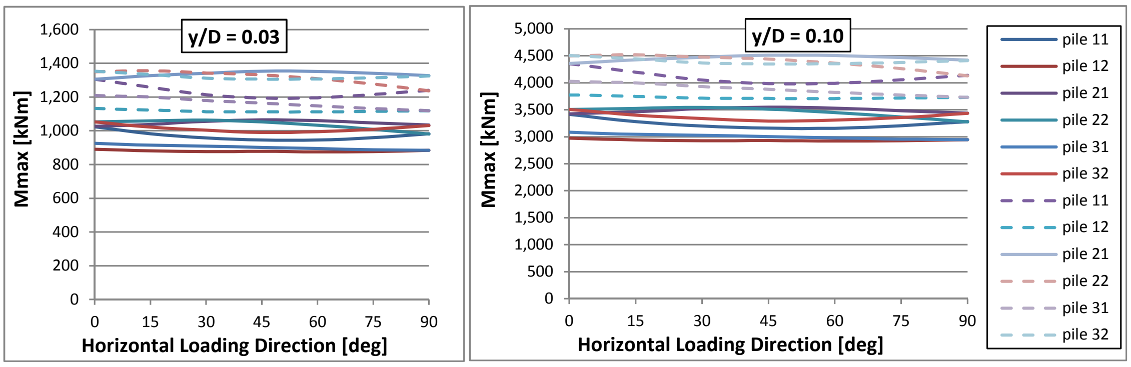


Figure 5.38: Maximum Bending Moments for 2x3 pile group (s_x=2D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

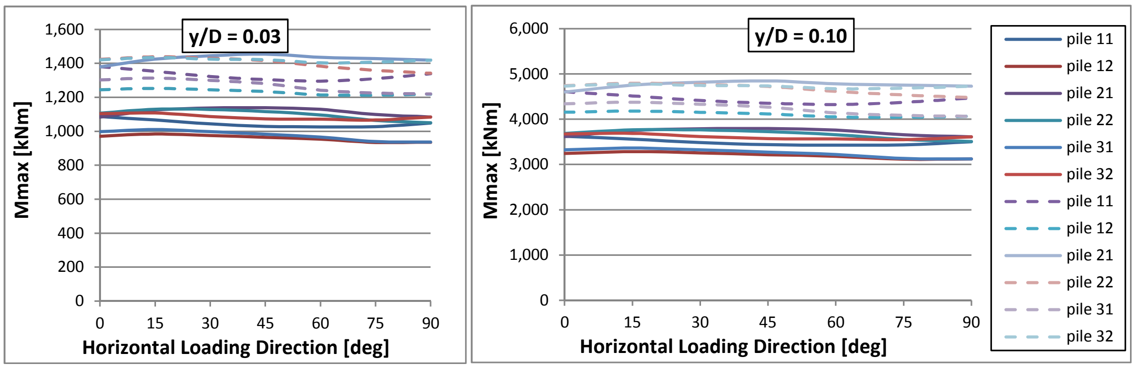


Figure 5.39: Maximum Bending Moments for 2x3 pile group (s_x=$2D$, s_y=$3D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

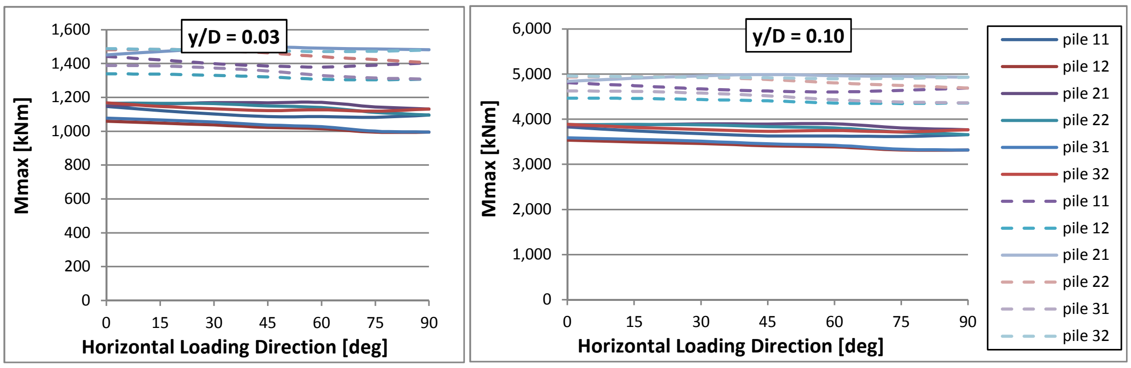


Figure 5.40: Maximum Bending Moments for 2x3 pile group (s_x=$2D$, s_y=$4D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

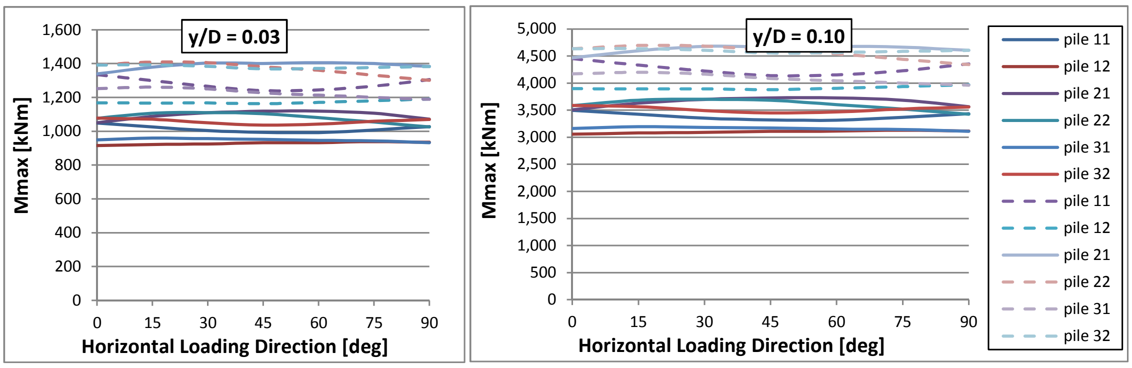


Figure 5.41: Maximum Bending Moments for 2x3 pile group (s_x=$3D$, s_y=$2D$), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

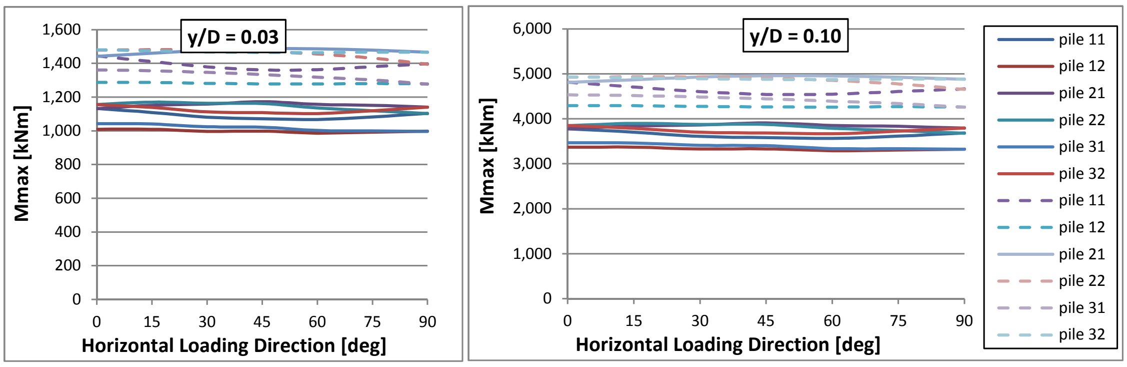


Figure 5.42: Maximum Bending Moments for 2x3 pile group (s_x=3D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

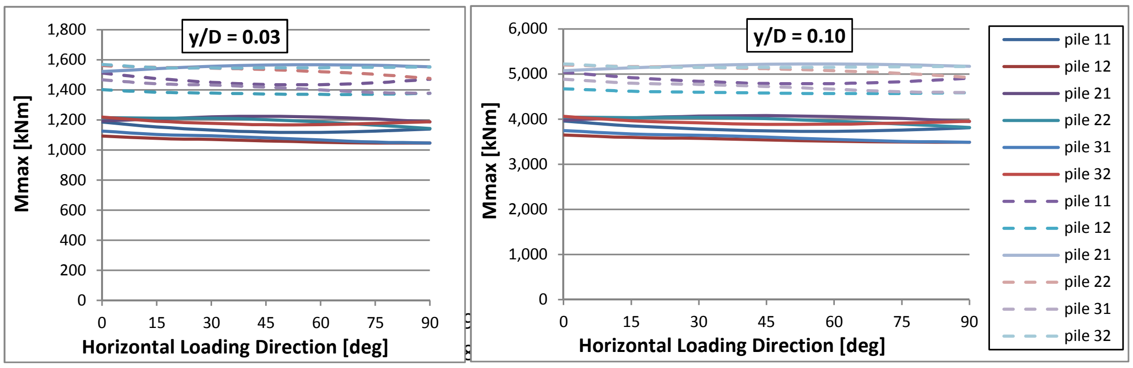


Figure 5.43: Maximum Bending Moments for 2x3 pile group (s_x=3D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

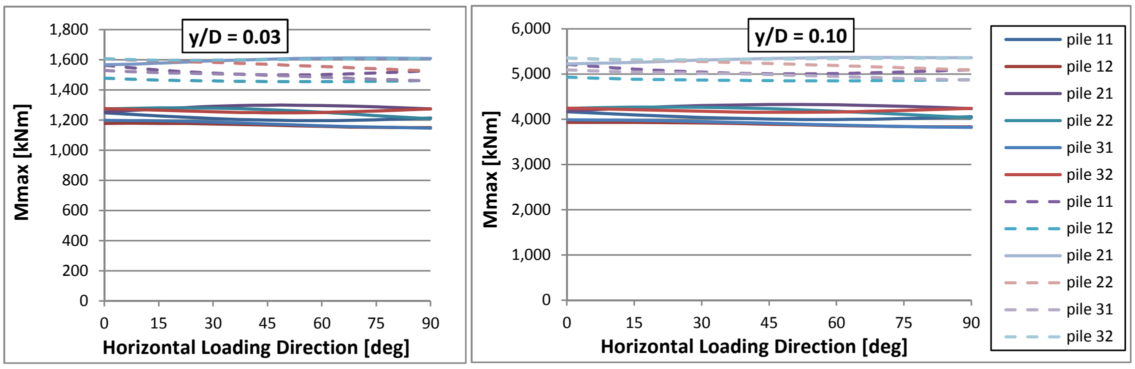


Figure 5.44: Maximum Bending Moments for 2x3 pile group (s_x=3D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

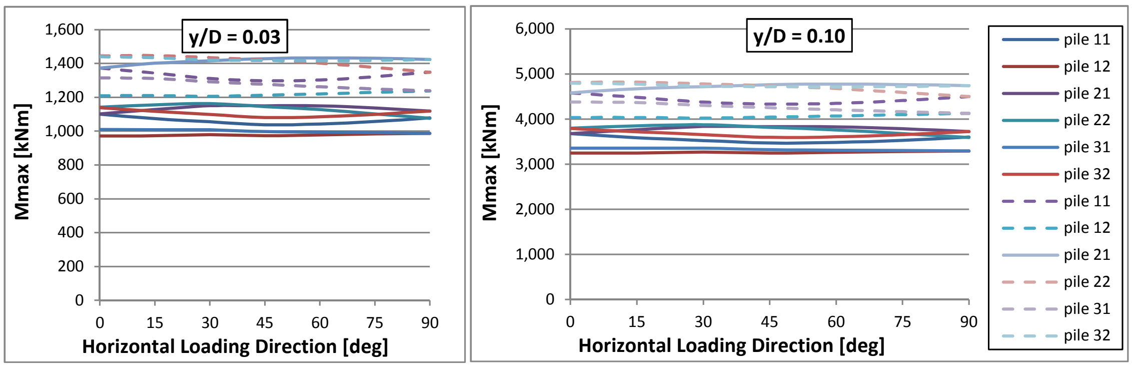


Figure 5.45: Maximum Bending Moments for 2x3 pile group (s_x=4D, s_y=2D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

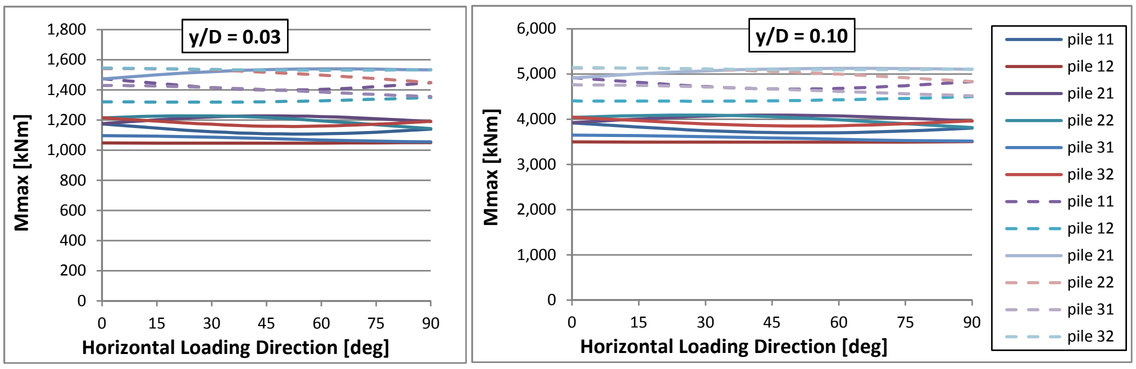


Figure 5.46: Maximum Bending Moments for 2x3 pile group (s_x=4D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

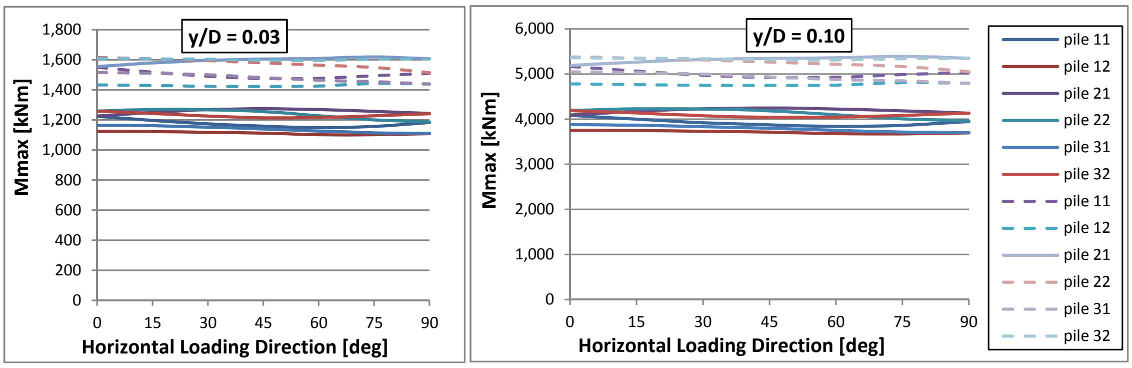


Figure 5.47: Maximum Bending Moments for 2x3 pile group (s_x=4D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

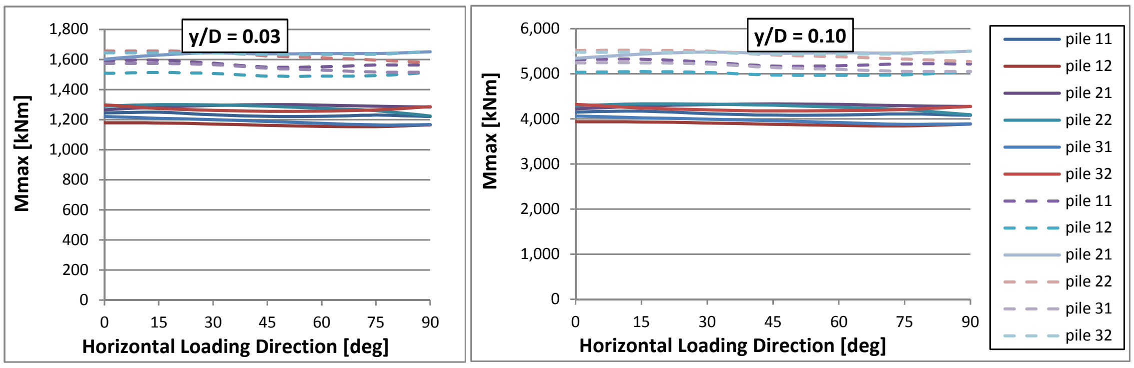


Figure 5.48: Maximum Bending Moments for 2x3 pile group (s_x=4D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

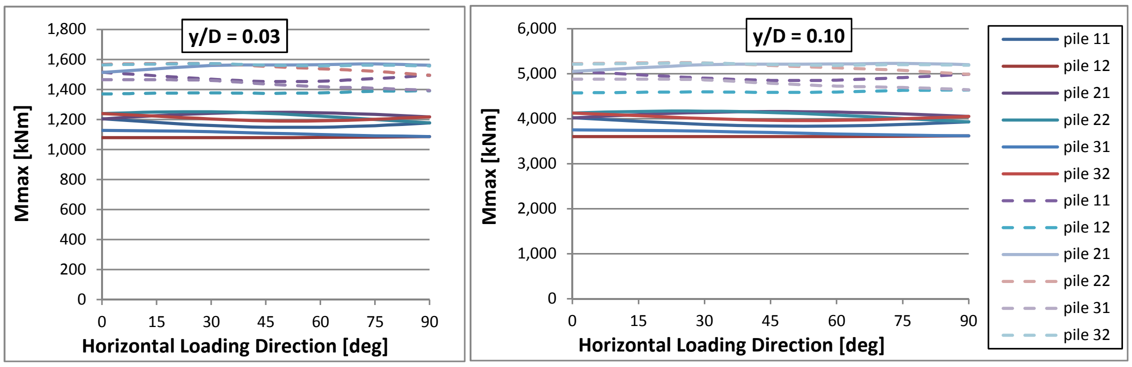


Figure 5.49: Maximum Bending Moments for 2x3 pile group (s_x=5D, s_y=3D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

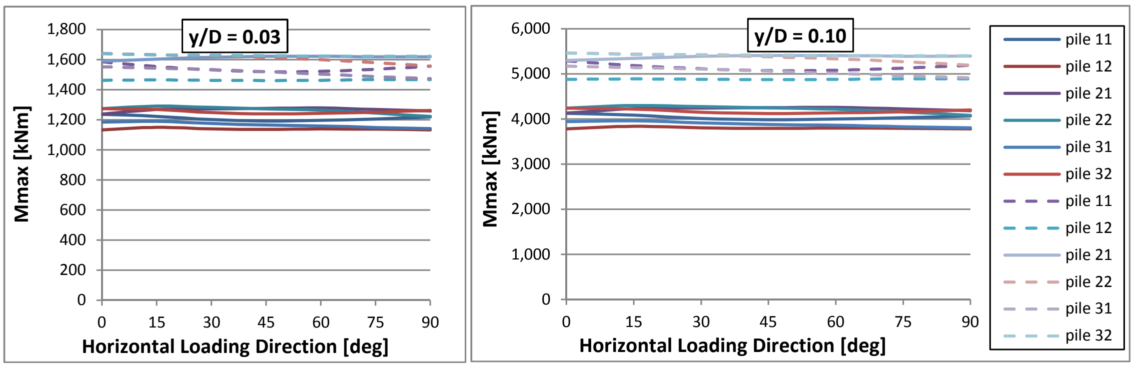


Figure 5.50: Maximum Bending Moments for 2x3 pile group (s_x=5D, s_y=4D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

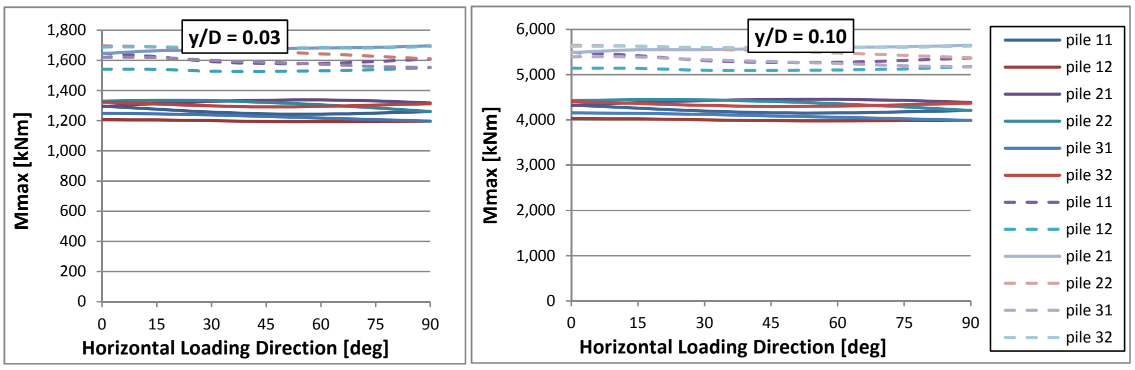


Figure 5.51: Maximum Bending Moments for 2x3 pile group (s_x=5D, s_y=5D), considering different normalized displacements y/D and different horizontal loading directions. Full lines - loose sand; dashed lines - dense sand.

6 Discussion

The conducted parametric study served as a basis for improvement of the design methodology of laterally loaded pile groups. This statement is supported by the fact that there exist considerable discrepancies between interaction factors and bending moments inside the pile group, which confirms the different behaviour of piles inside the group.

The level of interaction between the piles inside the group is higher at higher load levels, as expected. It was shown that interaction factor levels highly depend on the soil type - they are slightly lower for dense, in comparison with loose sands. These differences are smaller at lower load levels, where less plastic deformation occurs.

By means of quantification, interaction factors are between 0.6-0.8 for working load levels, and 0.4-0.8 for high load levels. Finally, the interaction factors increase with increasing pile spacings, as expected.

The differences in bending moments were observed for different piles inside the group. However, the influence of loading direction on the bending moment was relatively small.

The force in some individual piles changes significantly with the loading direction, characterized by skewed lines in the interaction factor plots. These piles are therefore more sensitive to the change (or unpredictable) horizontal loading direction. It was observed that the pile location within the group plays the more significant role than the considered load direction. This aspect of the pile behaviour should be analyzed in more detail, in order to identify the "critical" pile positions inside the pile group. The fact that some piles are more sensitive to the change of loading direction can be the governing design factor.

7 Conclusions

This dissertation deal with the extensive numerical study on the influence of the arbitrary lateral load direction on the response of piles inside the pile groups. Apriori sensitivity analysis of the considered problem has been done in the first phase of this research. The main problem parameters have been identified and served as a basis for numerical model generation and selection of parametric study parameters.

Within the thesis, two cutting-edge pile modelling techniques were compared. The full 3D pile model was adopted as a more reliable solution in comparison with the embedded beam (EB) model. The obtained results showed that the interface properties in the EB model formulation in PLAXIS 3D do not influence the lateral response of the pile group severely. However, when shear forces in the pile become large, plasticity will occur in the surrounding soil elements, outside the elastic zone. In other words, pile-soil interaction in lateral directions can only be controlled by the adjustments of surrounding soil stiffness parameters.

Numerical model of the laterally loaded pile group was proposed and validated against the measured results of laterally loaded single pile and pile row. Fitting techniques for the approximation of shear forces in the pile have been evaluated.

The capabilities of PLAXIS 3D software to perform multiple numerical simulations have been tested in an automated scripting environment, through the development of own computer programs. The performances of such a concept have shown to be acceptable and, in the case of this thesis, completely necessary for the solution.

The common approach to the analysis of horizontal loaded pile groups is expanded with the influence of arbitrary direction of horizontal loading, which will gain a better insight into the real behaviour of a pile groups. The presented results are improvement of methodology for analysis of pile groups under arbitrary horizontal loading. The results of this study provide clearer insight into the pile group mechanisms under lateral loading.

All presented results are to be used with caution when the significant difference from the research assumptions is present. This is especially important bearing in mind the different pile cap boundary conditions in the real pile groups. Finally, developed computer programs and scripts presented in this thesis will serve as a strong starting point for both academia and engineers for further studies of geotechnical problems.

8 Future Work

Beside the fact that this research provided significant insight into the pile group interactions under arbitrary loading, still many research topics remain open. Based upon the assumptions and restrictions which served as a basis for the presented study, as well as based on the main conclusions of this work, recommendations for further research topics are specified below:

- Analysis of the laterally loaded pile group under arbitrary loading, using the Strain Wedge model with arbitrary wedge orientation
- Experimental study of the pile group deformation patterns using modern technologies such as PIV and CT
- Study of the influence of concrete nonlinearity and cracking on the lateral response of pile group under working loading conditions
- Formulation and verification of the embedded beam model with advanced interface formulation, that can account for the lateral loading
- Analysis of the interaction effects under arbitrary lateral loading for the pile groups with irregular pile positions and lengths (e.g. piled raft foundations)
- Analysis of interaction inside the laterally loaded pile group of jacked-in piles
- Analysis of the influence of soil heterogeneity, especially the influence of soft top layer with variable thickness, on the lateral response of pile groups

Bibliography

[1] Poulos HG. *Tall building foundation design.* CRC Press, 2017.

[2] Rao SN, Ramakrishna V, Rao MB. "Influence of rigidity on laterally loaded pile groups in marine clay". In: *Journal of Geotechnical and Geoenvironmental Engineering* 124.6 (1998), pp. 542–549.

[3] Randolph M. "Design methods for pile group and piled rafts". In: *Proc. 13th Int. Conf. on SMFE.* Vol. 5. 1994, pp. 61–82.

[4] Burland J, Broms B, De Mello V. "B (1977). Behaviour of foundations and structures". In: *Proc. 9th Int. Conf. Soil Mech. Found. Engng, Tokyo.* Vol. 2, pp. 495–546.

[5] Mandolini A, Viggiani C. "Settlement of piled foundations". In: *Géotechnique* 47.4 (1997), pp. 791–816.

[6] Poulos HG, Davis EH. *Pile foundation analysis and design.* Monograph. 1980.

[7] Papadopoulou MC, Comodromos EM. "On the response prediction of horizontally loaded fixed-head pile groups in sands". In: *Computers and Geotechnics* 37.7-8 (2010), pp. 930–941.

[8] Lesny K. "Design of Laterally Loaded Piles-Limits of Limit State Design?" In: *Geo-Risk 2017: Reliability-Based Design and Code Developments (GSP 283)* (2017), pp. 267–276.

[9] Rudolph C, Bienen B, Grabe J. "Effect of variation of the loading direction on the displacement accumulation of large-diameter piles under cyclic lateral loading in sand". In: *Canadian geotechnical journal* 51.10 (2014), pp. 1196–1206.

[10] Comodromos EM, Pitilakis KD. "Response evaluation for horizontally loaded fixed-head pile groups using 3-D non-linear analysis". In: *International Journal for Numerical and Analytical Methods in Geomechanics* 29.6 (2005), pp. 597–625.

[11] *Eurocode 7: Geotechnical design - Part 1: General rules.* 2004.

[12] Ochoa M, W. O'Neill M. "Lateral pile interaction factors in submerged sand". In: *Journal of Geotechnical Engineering* 115.3 (1989), pp. 359–378.

[13] Randolph MF. "The response of flexible piles to lateral loading". In: *Geotechnique* 31.2 (1981), pp. 247–259.

[14] Fan CC, H. Long J. “A modulus-multiplier approach for non-linear analysis of laterally loaded pile groups”. In: *International journal for numerical and analytical methods in geomechanics* 31.9 (2007), pp. 1117–1145.

[15] Su D, Yan W. “A multidirectional p–y model for lateral sand–pile interactions”. In: *Soils and foundations* 53.2 (2013), pp. 199–214.

[16] Mayoral JM, Pestana JM, Seed RB. “Multi-directional cyclic p–y curves for soft clays”. In: *Ocean Engineering* 115 (2016), pp. 1–18.

[17] Gu M et al. “Response of 1 x 2 pile group under eccentric lateral loading”. In: *Computers and Geotechnics* 57 (2014), pp. 114–121.

[18] Chen S, Kong L, Zhang LM. “Analysis of pile groups subjected to torsional loading”. In: *Computers and Geotechnics* 71 (2016), pp. 115–123.

[19] Georgiadis K, Sloan S, Lyamin A. “Ultimate lateral pressure of two side-by-side piles in clay”. In: *Géotechnique* 63.9 (2013), pp. 733–745. DOI: 10.1680/geot.12.P.030.

[20] Su D, Zhou YG. “Effect of Loading Direction on the Response of Laterally Loaded Pile Groups in Sand”. In: *International Journal of Geomechanics* (2015). DOI: 10.1061/(ASCE)GM.1943-5622.0000544.

[21] Kotthaus M. “Zum Tragverhalten von horizontal belasteten Pfahlreihen aus langen Pfählen in Sand”. PhD thesis. Lehrstuhl für Grundbau und Bodenmechanik, Ruhr-Universität Bochum, 1992.

[22] VAN IMPE W. “Foreword: Belgian geotechnics’ experts research on screw piles”. In: *BELGIAN SCREW PILE TECHNOLOGY DESIGN AND RECENT DEVELOPMENTS* (2003).

[23] Broms BB. “Lateral resistance of piles in cohesive soils”. In: *Journal of the Soil Mechanics and Foundations Division* 90.2 (1964), pp. 27–64.

[24] Broms BB. “Lateral resistance of piles in cohesionless soils”. In: *Journal of the Soil Mechanics and Foundations Division* 90.3 (1964), pp. 123–158.

[25] Brinkgreve R et al. “PLAXIS 3D AE”. In: *User manual, Plaxis bv* (2015).

[26] Winkler E. *Die Lehre von der Elasticitaet und Festigkeit mit besonderer Rücksicht auf ihre Anwendung in der Technik.* 1867.

[27] Hetenyi M. “Beams on Elastic Foundation, University of Michigan”. In: *Ann Arbor* (1946).

[28] Terzaghi K. "Evalution of conefficients of subgrade reaction". In: *Geotechnique* 5.4 (1955), pp. 297–326.

[29] *Subsoil - Verification of the safety of earthworks and foundations - Supplementary rules to DIN EN 1997-1.* DIN, 2010.

[30] *Bored cast-in-place piles: Formation, design and bearing capacity.* DIN, 1990.

[31] Geotechnik APDG für. *EA-Pfähle: Empfehlungen des Arbeitskreises" Pfähle".* Ernst, 2012.

[32] *Tragverhalten von Pfahlgruppen unter Horizontal Belastung, author=Klüber, E, year=1988, publisher=Institut für Grundbau, Boden- und Felsmechanik, TH Darmstadt.*

[33] Norris G. "Theoretically based BEF laterally loaded pile analysis". In: *Proceedings of the 3rd International Conference on Numerical Methods in Offshore Piling.* Navtes. 1986, pp. 361–386.

[34] Ashour M, Pilling P, Norris G. "Documentation of the strain wedge model program for analyzing laterally loaded piles and pile groups". In: *Proc., 32nd Engineering Geology and Geotechnical Engineering Symposium.* 1997, pp. 26–28.

[35] Ashour M, Norris G, Pilling P. "Lateral loading of a pile in layered soil using the strain wedge model". In: *Journal of geotechnical and geoenvironmental engineering* 124.4 (1998), pp. 303–315.

[36] Ashour M, Pilling P, Norris G. "Assessment of pile group response under lateral load". In: (2001).

[37] Ashour M, Pilling P, Norris G. "Lateral behavior of pile groups in layered soils". In: *Journal of Geotechnical and Geoenvironmental Engineering* 130.6 (2004), pp. 580–592.

[38] McClelland B, Focht J. "Soil modulus for laterally loaded piles. J Soil Mech Found Div". In: *Proceedings of the American society of civil engineers.* Vol. 1. 1958, p. 22.

[39] Matlock H. "Correlations for design of laterally loaded piles in soft clay". In: *Offshore technology in civil engineering's hall of fame papers from the early years* (1970), pp. 77–94.

[40] Reese LC, Cox WR, Koop FD. "Analysis of laterally loaded piles in sand". In: *Offshore Technology in Civil Engineering Hall of Fame Papers from the Early Years* (1974), pp. 95–105.

[41] Dodds AM, Martin GR. *Modeling Pile Behavior in Large Pile Groups under Lateral Loading (Technical Report MCEER-07-0004).* Tech. rep. University of Southern California, 2007.

[42] Reese L et al. "LPILE Plus Version 4.0—A Program for the Analysis of Piles and Drilled Shafts under Lateral Loads; Ensoft". In: *Inc. Austin, Texas* (2000).

[43] Reese L et al. "Computer program GROUP version 8.0 technical manual, Ensoft". In: *Inc., Austin, Texas* (2010).

[44] Robertson PK, Davies MP, Campanella RG. "Design of laterally loaded driven piles using the flat dilatometer". In: *Geotechnical Testing Journal* 12.1 (1989), pp. 30–38.

[45] Briaud JL, Smith T, Meyer B. "Laterally loaded piles and the pressuremeter: comparison of existing methods". In: *Laterally loaded deep foundations: Analysis and performance.* ASTM International, 1984.

[46] Brown DA, Shie CF. "Three dimensional finite element model of laterally loaded piles". In: *Computers and Geotechnics* 10.1 (1990), pp. 59–79.

[47] Brown DA, Shie CF. "Some numerical experiments with a three dimensional finite element model of a laterally loaded pile". In: *Computers and Geotechnics* 12.2 (1991), pp. 149–162.

[48] Brown DA, Morrison C, Reese LC. "Lateral load behavior of pile group in sand". In: *Journal of Geotechnical Engineering* 114.11 (1988), pp. 1261–1276.

[49] Brown DA et al. *Static and dynamic lateral loading of pile groups.* TRB Washington, DC, USA, 2001.

[50] Fayyazi MS, Taiebat M, Finn WL. "Group reduction factors for analysis of laterally loaded pile groups". In: *Canadian geotechnical journal* 51.7 (2014), pp. 758–769.

[51] Basile F. "Analysis and design of pile groups". In: *Numerical analysis and modelling in geomechanics* (2003), pp. 278–315.

[52] Briaud JL. *Geotechnical Engineering: unsaturated and saturated soils.* John Wiley & Sons, 2013.

[53] Smith M. *ABAQUS/Standard User's Manual, Version 6.9.* English. United States: Dassault Systèmes Simulia Corp, 2009.

[54] CSI. *ETABS.* Version 18.

[55] Ansys. *Ansys.*

[56] SOFiSTiK. *SOFiSTiK*. Version 2020.

[57] Poulos HG. “Behavior of laterally loaded piles I. single piles”. In: *Journal of Soil Mechanics & Foundations Div* (1971).

[58] Poulos HG. “Behavior of laterally loaded piles II. Pile groups”. In: *Journal of Soil Mechanics & Foundations Div* (1971).

[59] Banerjee PK, Driscoll RM. “Three-dimensional analysis of vertical pile groups”. In: *ha Proceedings, Second International Conference on Numerical Methods in Geomechanics, held at Virginia Polytechnic, Blacksburg, Virginia in June*. 1976, pp. 438–450.

[60] Pecker A, Pender M, et al. “Earthquake resistant design of foundations: new construction”. In: *ISRM International Symposium*. International Society for Rock Mechanics and Rock Engineering. 2000.

[61] Brown DA, Shie CF. “Three dimensional finite element model of laterally loaded piles”. In: *Computers and Geotechnics* 10.1 (1990), pp. 59–79.

[62] Brown DA, Shie CF. “Numerical experiments into group effects on the response of piles to lateral loading”. In: *Computers and Geotechnics* 10.3 (1990), pp. 211–230.

[63] Mises Rv. “Mechanik der festen Körper im plastisch-deformablen Zustand”. In: *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse* 1913 (1913), pp. 582–592.

[64] Drucker DC, Prager W. “Soil mechanics and plastic analysis or limit design”. In: *Quarterly of Applied Mathematics* 10.2 (1952), pp. 157–165.

[65] Wakai A, Gose S, Ugai K. “3-D elasto-plastic finite element analyses of pile foundations subjected to lateral loading”. In: *Soils and Foundations* 39.1 (1999), pp. 97–111.

[66] Yang Z, Jeremić B. “Numerical study of group effects for pile groups in sands”. In: *International Journal for Numerical and Analytical Methods in Geomechanics* 27.15 (2003), pp. 1255–1276.

[67] Comodromos E, Papadopoulou M. “Response evaluation of horizontally loaded pile groups in clayey soils”. In: *Géotechnique* 62.4 (2012), p. 329.

[68] Papadopoulou MC, Comodromos EM. “Explicit extension of the p–y method to pile groups in sandy soils”. In: *Acta Geotechnica* 9.3 (2014), pp. 485–497.

[69] Comodromos EM, Papadopoulou MC. “Explicit extension of the p–y method to pile groups in cohesive soils”. In: *Computers and Geotechnics* 47 (2013), pp. 28–41.

[70] Brown DA, Reese LC, O'Neill MW. "Cyclic lateral loading of a large-scale pile group". In: *Journal of Geotechnical Engineering* 113.11 (1987), pp. 1326–1343.

[71] Christensen DS. "Full Scale Static Lateral Load Test of a 9 Pile Group in Sand". In: (2006).

[72] Huang AB et al. "Effects of construction on laterally loaded pile groups". In: *Journal of geotechnical and geoenvironmental engineering* 127.5 (2001), pp. 385–397.

[73] Ilyas T et al. "Centrifuge Model Study of Laterally Loaded Pile Groups in Clay". In: *Journal of Geotechnical and Geoenvironmental Engineering* 130.3 (2004), pp. 274–283. DOI: `10.1061/(ASCE)1090-0241(2004)130:3(274)`.

[74] Kotthaus M, Grundhoff T, Jessberger HL. "Single piles and pile rows subjected to static and dynamic lateral load". In: *International Conference Centrifuge 94*. 1994, pp. 497–502.

[75] McVay M, Casper R, Shang TI. "Lateral response of three-row groups in loose to dense sands at 3D and 5D pile spacing". In: *Journal of Geotechnical Engineering* 121.5 (1995), pp. 436–441.

[76] Morrison CS, Reese LC, et al. "A lateral-load test of a full-scale pile group in sand". In: (1988).

[77] Rollins KM, Peterson KT, Weaver TJ. "Lateral load behavior of full-scale pile group in clay". In: *Journal of geotechnical and geoenvironmental engineering* 124.6 (1998), pp. 468–478.

[78] Rollins KM, Sparks A. "Lateral resistance of full-scale pile cap with gravel backfill". In: *Journal of Geotechnical and Geoenvironmental Engineering* 128.9 (2002), pp. 711–723.

[79] Rollins KM, Lane JD, Gerber TM. "Measured and computed lateral response of a pile group in sand". In: *Journal of Geotechnical and Geoenvironmental Engineering* 131.1 (2005), pp. 103–114.

[80] Rollins KM et al. "Pile spacing effects on lateral pile group behavior: load tests". In: *Journal of geotechnical and geoenvironmental engineering* 132.10 (2006), pp. 1262–1271.

[81] Ruesta PF, Townsend FC. "Evaluation of laterally loaded pile group at Roosevelt Bridge". In: *Journal of Geotechnical and Geoenvironmental Engineering* 123.12 (1997), pp. 1153–1161.

[82] Snyder JL. “Full-Scale Lateral-Load Tests of a 3x5 Pile Group in Soft Clays and Silts”. In: (2004).

[83] Walsh JM. “Full-scale lateral load test of a 3x5 pile group in sand”. In: (2005).

[84] Ashour M, Ardalan H. “Employment of the p-multiplier in pile-group analysis”. In: *Journal of Bridge Engineering* 16.5 (2011), pp. 612–623.

[85] Otani J, Pham KD, Sano J. “Investigation of failure patterns in sand due to laterally loaded pile using X-ray CT”. In: *Soils and Foundations* 46.4 (2006), pp. 529–535.

[86] Hajialilue-Bonab M, Azarnya-Shahgoli H, Sojoudi Y. “Soil deformation pattern around laterally loaded piles”. In: *International Journal of Physical Modelling in Geotechnics* 11.3 (2011), pp. 116–125.

[87] Hajialilue-Bonab M, Sojoudi Y, Puppala AJ. “Study of strain wedge parameters for laterally loaded piles”. In: *International Journal of Geomechanics* 13.2 (2013), pp. 143–152.

[88] Iai S et al. “Soil-pile interaction under lateral load”. In: *Soil-Foundation-Structure Interaction.* CRC Press, 2010, pp. 117–124.

[89] Morita K et al. “Evaluation of vertical and lateral bearing capacity mechanisms of pile foundations using X-ray CT”. In: *Proceedings of international workshop on recent advances of deep foundations (IWDPF07), Yokosuka, Japan (eds Y. Kikuchi, M. Kimura, J. Otani and Y. Morikawa).* 2007, pp. 217–223.

[90] Cox WR, Dixon DA, Murphy BS. “Lateral-load tests on 25.4-mm (1-in.) diameter piles in very soft clay in side-by-side and in-line groups”. In: *Laterally loaded deep foundations: Analysis and performance.* ASTM International, 1984.

[91] Reese LC, Van Impe WF. *Single piles and pile groups under lateral loading.* CRC press, 2010.

[92] Chandrasekaran S, Boominathan A, Dodagoudar G. “Group interaction effects on laterally loaded piles in clay”. In: *Journal of geotechnical and geoenvironmental engineering* 136.4 (2010), pp. 573–582.

[93] McVay M et al. “Centrifuge testing of large laterally loaded pile groups in sands”. In: *Journal of Geotechnical and Geoenvironmental Engineering* 124.10 (1998), pp. 1016–1026.

[94] Mokwa RL, Duncan JM. “Laterally loaded pile group effects and py multipliers”. In: *Foundations and ground improvement.* ASCE. 2001, pp. 728–742.

[95] Pender M. “Aseismic pile foundation design analysis”. In: *Bulletin of the New Zealand Society for Earthquake Engineering* 26.1 (1993), pp. 49–160.

[96] Curras CJ et al. “Lateral loading & seismic response of CIDH pile supported bridge structures”. In: *Foundations and Ground Improvement.* ASCE. 2001, pp. 260–275.

[97] Bowles J. *Foundation Analysis and Design.* McGraw-hill, 1996.

[98] Comodromos EM, Papadopoulou MC, Rentzeperis IK. “Effect of cracking on the response of pile test under horizontal loading”. In: *Journal of geotechnical and geoenvironmental engineering* 135.9 (2009), pp. 1275–1284.

[99] Mokwa RL, Duncan JM. “Rotational Restraint of Pile Caps during Lateral Loading”. In: *Journal of Geotechnical and Geoenvironmental Engineering* 129.9 (2003), pp. 829–837. DOI: 10.1061/(ASCE)1090-0241(2003)129:9(829).

[100] Katzenbach R, Leppla S, Choudhury D. *Foundation systems for high-rise structures.* CRC press, 2016.

[101] Reese LC, Van Impe W. *F.(2001) Single Piles and Pile Groups Under Lateral Loading.*

[102] Das BM. *Principles of foundation engineering.* Cengage learning, 2015.

[103] Dunnavant TW, O’Neill MW. “Experimental p-y model for submerged, stiff clay”. In: *Journal of Geotechnical Engineering* 115.1 (1989), pp. 95–114.

[104] Brown DA, SHIE CF, KUMAR M. “Py curves for laterally loaded piles derived from three-dimensional finite element model”. In: *International symposium on numerical models in geomechanics. 3 (NUMOG III).* 1989, pp. 683–690.

[105] Potts DM et al. *Finite element analysis in geotechnical engineering: application.* Vol. 2. Thomas Telford London, 2001.

[106] Comodromos EM. “The contribution of numerical analysis to the response prediction of pile foundations”. In: *Linear and Non-linear Numerical Analysis of Foundations.* CRC Press, 2014, pp. 49–96.

[107] Sastry V, Meyerhof G. “Behaviour of flexible piles under inclined loads”. In: *Canadian Geotechnical Journal* 27.1 (1990), pp. 19–28.

[108] Anagnostopoulos C, Georgiadis M. “Interaction of axial and lateral pile responses”. In: *Journal of Geotechnical Engineering* 119.4 (1993), pp. 793–798.

[109] Karthigeyan S, Ramakrishna V, Rajagopal K. “Numerical investigation of the effect of vertical load on the lateral response of piles”. In: *Journal of Geotechnical and Geoenvironmental Engineering* 133.5 (2007), pp. 512–521.

[110] Hussien MN et al. “Vertical loads effect on the lateral pile group resistance in sand”. In: *Geomechanics and Geoengineering* 7.4 (2012), pp. 263–282.

[111] Hussien MN et al. “On the influence of vertical loads on the lateral response of pile foundation”. In: *Computers and Geotechnics* 55 (2014), pp. 392–403.

[112] Comodromos EM, Papadopoulou MC, Laloui L. “Contribution to the design methodologies of piled raft foundations under combined loadings”. In: *Canadian Geotechnical Journal* 53.4 (2015), pp. 559–577.

[113] Hazzar L, Hussien MN, Karray M. “Vertical load effects on the lateral response of piles in layered media”. In: *International Journal of Geomechanics* 17.9 (2017), p. 04017078.

[114] Zhao C. “A Contribution to Modelling of Mechanized Tunnel Excavation”. PhD. Thesis. Ruhr University Bochum, 2018.

[115] Schanz T, Vermeer PA. “Formulation and verification of the Hardening Soil Model”. In: *Beyond 2000 in Computation Geotechnics - 10 Years of PLAXIS*. Rotterdam: Balkema, 1999, pp. 1–16.

[116] Sadek M, Shahrour I. “A three dimensional embedded beam element for reinforced geomaterials”. In: *International Journal for Numerical and Analytical Methods in Geomechanics* 28.9 (2004), pp. 931–946.

[117] Engin H, Septanika E, Brinkgreve R. “Improved embedded beam elements for the modelling of piles”. In: (2007).

[118] Marjanović M et al. “Modeling of laterally loaded piles using embedded beam elements”. English. In: *Proceedings of the International Conference ”Contemporary achievements in Civil Engineering 2016”*. Ed. by M Bešević et al. Subotica, Serbia: University of Novi Sad - Faculty of Civi Enginering Subotica, 2016, pp. 349–358. DOI: 10.14415/konferencijaGFS2016.035.

[119] Dao T. “Validation of PLAXIS Embedded Piles For Lateral Loading Validation of PLAXIS Embedded Piles For Lateral Loading”. MSc. Thesis. TU Delft, 2011.

[120] Turello DF, Pinto F, Sánchez PJ. "Three dimensional elasto-plastic interface for embedded beam elements with interaction surface for the analysis of lateral loading of piles". In: *International Journal for Numerical and Analytical Methods in Geomechanics* 41.6 (2017), pp. 859–879.

[121] Turello DF, Pinto F, Sanchez PJ. "Embedded beam element with interaction surface for lateral loading of piles". In: *International Journal for Numerical and Analytical Methods in Geomechanics* 40.March 2007 (2016), pp. 568–582. DOI: `10.1002/nag`. arXiv: `nag.2347 [10.1002]`.

[122] Ninić J, Ninic J. "Computational strategies for predictions of the soil-structure interaction during mechanized tunneling". In: June (2015).

[123] Tedesco G. "Offshore tower or platform foundations : numerical analysis of a laterally loaded single pile or pile group in soft clay and analysis of actions on a jacket structure". MSc. Thesis. University of Bologna, 2013.

[124] Zhang Y, Andersen KH, Tedesco G. "Ultimate bearing capacity of laterally loaded piles in clay – Some practical considerations". In: *Marine Structures* 50 (2016), pp. 260–275. DOI: `10.1016/j.marstruc.2016.09.002`.

[125] Jaky J. "The coefficient of earth pressure at rest". In: *Journal of the Society of Hungarian Architects and Engineers* (1944), pp. 355–358.

[126] Duncan JM, Evans Jr LT, Ooi PS. "Lateral load analysis of single piles and drilled shafts". In: *Journal of geotechnical engineering* 120.6 (1994), pp. 1018–1033.

[127] Van Rossum G, Drake Jr FL. *Python Reference Manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[128] Hussien MN et al. "Soil–pile separation effect on the performance of a pile group under static and dynamic lateral loads". In: *Canadian Geotechnical Journal* 47.11 (2010), pp. 1234–1246.

[129] Haiderali AE, Madabhushi G. "Evaluation of Curve Fitting Techniques in Deriving p–y Curves for Laterally Loaded Piles". In: *Geotechnical and Geological Engineering* 34.5 (2016). DOI: `10.1007/s10706-016-0054-2`.

[130] Saltelli A et al. *Global sensitivity analysis: the primer.* John Wiley & Sons, 2008.

[131] Khaledi K et al. "Robust and reliable metamodels for mechanized tunnel simulations". In: *Computers and Geotechnics* 61 (2014), pp. 1–12. DOI: `10.1016/j.compgeo.2014.04.005`.

[132] Kennedy J, Eberhart R. “Particle swarm optimization”. In: *Neural Networks, 1995. Proceedings., IEEE International Conference on* 4 (1995), 1942–1948 vol.4. DOI: `10.1109/ICNN.1995.488968`.

[133] Garnier J et al. “Catalogue of scaling laws and similitude questions in geotechnical centrifuge modelling”. In: *International Journal of Physical Modelling in Geotechnics* 7.3 (2007), pp. 01–23.

[134] Kayalar A. “Bettung von horizontal belasteten Pfahlreihen und Bohrpfahlwänden”. In: (2012).

[135] Hannigan P et al. *Design and Construction of Driven Pile Foundations, Federal Highway Administration, FHWA, Reference Manual-Volume I and II, National Highway Institute.* Tech. rep. NHI-05-042, Courses, 2006.

[136] State Highway AA of, Officials T. *AASHTO LRFD Bridge Design Specifications, Customary US Units: 2006 Interim Revisions.* American Association of State Highway and Transportation Officials, 2006.

Biography

Miloš Marjanović was born on January 8th 1986 in Užice, Republic of Serbia, where he finished elementary school and Gymnasium. He received the "*Vuk Karadžić*" Award during the elementary school, and the Gymnasium Award for the best graduation thesis. He completed both BSc. (2005-2009) and MSc. studies (2009-2010) at the Faculty of Civil Engineering in Belgrade, Module Structural Engineering. Miloš Marjanović won the Prize of the Faculty of Civil Engineering in Belgrade (*Academician Prof. Djordje Lazarević Fund*) for the best MSc. thesis in the field of concrete structures. He won the Prize of the Regional Chamber of Commerce Užice for the best students in 2009. He received several scholarships for achieved success during regular studies.

Miloš Marjanović enrolled the PhD studies at the Faculty of Civil Engineering in Belgrade in 2010. Since December 2010 he has been working at the Faculty of Civil Engineering as a Teaching Assistant in scientific fields: Soil Mechanics and Geotechnics in Transportation Engineering. Since 2015, Miloš Marjanović participates as the Scholarship Holder in the international PhD program *SEEFORM*, in which he stayed for 13 months at Ruhr University Bochum, Chair of Foundation Engineering, Soil and Rock Mechanics. He also participated in several seminars for PhD students in Serbia and abroad. He is also engaged as a researcher on the Project of the Ministry of Education, Science and Technological Development of Serbia N^0 TR-36046.

Close field of his scientific research is related to the static and dynamic analysis of soil-structure interaction using Finite Element Method. He authored and co-authored 4 papers in peer-reviewed journals and 24 papers in national and international conference proceedings.

He is a member of the Serbian Society for Soil Mechanics and Geotechnical Engineering and the International Society for Soil Mechanics and Geotechnical Engineering (ISSMGE). He speaks and writes English and German, and uses Russian with basic knowledge. He is a long-term member of the Artistic Association of University of Belgrade "Branko Krsmanović". He is married and lives in Belgrade.

Appendix - Computer Programs

Main Program - Multiple Simulations Processor

Configuration Constants `Main_ConfigConstants.py`

```
import os

SimExtension = '.p3d'
RecordExtension = '.sav'
DefaultSimDir = os.getcwd() + '\\Simulations'
DSExtension = '.dat'
DefaultBlueprint = 'BLUEPRINT'+SimExtension

IconTrue = os.getcwd() + '\\Config\\Icon_True.png'
IconFalse = os.getcwd() + '\\Config\\Icon_False.png'
```

Basic Functions `Main_Functions.py`

```
import numpy as np
import pickle
import openpyxl as xl
from openpyxl.chart import Series, Reference, ScatterChart
import os

from PyQt5.QtWidgets import QFileDialog, QMessageBox, QDialog, QGridLayout


# FILE/OBJECT OPERATORS  ############################################################

def folder_size(path):

    total_size = 0

    for dirpath, dirnames, filenames in os.walk(path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            total_size += os.path.getsize(fp)

    return total_size


def save_object(obj, path):
    obj_file = open(path, 'wb')
    pickle.dump(obj, obj_file)
    obj_file.close()


def load_object(path):
    obj_file = open(path, 'rb')
    obj = pickle.load(obj_file)
    obj_file.close()
    return obj


# ND ARRAY ##########################################################################


def unique_ndarray(numpy_ndarray):

    row_num, col_num = np.shape(numpy_ndarray)

    unique_rows = []
    duplicate_rows = []

    for r_1 in range(0, row_num):
    # Search for duplicated rows (all values in both rows are the same!)

        if (r_1 in duplicate_rows) is False:

            unique_rows.append(r_1)

            for r_2 in range(r_1 + 1, row_num):
                if list(numpy_ndarray[r_1, :]) == list(numpy_ndarray[r_2, :]):
                    duplicate_rows.append(r_2)

    # If duplicated rows found, reshape ndarray by inserting unique rows from original
     table

    if len(duplicate_rows) > 0:         # Some duplicates are found
        new_table = np.ndarray(shape=(len(unique_rows), col_num))

        for j, old_row in enumerate(unique_rows):
            new_table[j, :] = numpy_ndarray[old_row, :]
    else:
        new_table = numpy_ndarray

    return new_table, len(duplicate_rows)
    # Return unique ndarray and number of deleted rows

def link_np_tables(table1, table2):
    # Connects two tables, one below the other

    len1 = len(table1)
    len3 = len1 + len(table2)
    new_table = np.ndarray(shape=(len3, len(table1[0])))

    for r in range(0, len1):
        new_table[r, :] = table1[r, :]

    for r in range(len1, len3):
        new_table[r, :] = table2[r - len1, :]

    return new_table            # Return linked table


def np_slicer(r_c, filter_list, input_table):
    """ This function extracts desired rows or columns from initial numpy array (2D)
     and creates new (sliced) array """

    rows0, columns0 = np.shape(input_table)

    if r_c == 'C':
        new_table = np.zeros((rows0, len(filter_list)))

        for c1, c0 in enumerate(filter_list):
            new_table[:, c1] = input_table[:, c0]

    else:
        new_table = np.zeros((len(filter_list), columns0))

        for r1, r0 in enumerate(filter_list):
            new_table[r1, :] = input_table[r0, :]

    return new_table


def vector_nd(x):

    new_vector = np.zeros((len(x), 1))
    for j, x1 in enumerate(x):
        new_vector[j, 0] = x1

    return new_vector


# OPENPYXL  #########################################################################


def parse_xl_ws_nd(ws, rows, cols):

    data_table = np.ndarray(shape=(rows, cols))

    for r in range(1, rows + 1):
        for c in range(1, cols + 1):

            cell_value = ws.cell(row=r, column=c).value
            if type(cell_value) == int or type(cell_value) == float:
```

```
            # Accept only integers and floats

                data_table[r-1, c-1] = cell_value
            else:
                return None

    return data_table


def parse_xl_nd(filename, rows, cols):

    wb = xl.load_workbook(filename)
    ws = wb.worksheets[0]
    return parse_xl_ws_nd(ws, rows, cols)


def parse_xl_nd_full(filename, rows, columns, sheets):
    wb = xl.load_workbook(filename)

    Parsed = []
    final_nd = None

    for sheet in range(0, sheets):
        ws_data_parsed = parse_xl_ws_nd(wb.worksheets[sheet], rows, columns)

        if ws_data_parsed is not None:
            Parsed.append(ws_data_parsed)
        else:
            break

    if len(Parsed) > 0:

        sheets_2 = len(Parsed)

        if sheets_2 > 1:
            final_nd = np.ndarray((rows, columns, sheets_2))
            for sheet in range(0, sheets_2):
                final_nd[:, :, sheet] = Parsed[sheet][:, :]

        elif sheets_2 == 1:
            final_nd = Parsed[0]

    return final_nd


def fill_xl(worksheet, values, start=(1, 1)):

    startr = start[0]
    startc = start[1]

    for r in range(startr, startr + len(values)):
        for c in range(startc, startc + len(values[0])):
            worksheet.cell(row=r, column=c).value = values[r - startr][c - startc]
```

```
def fill_xl_row_col(worksheet, rowcol, vector, start=(1, 1)):

    if rowcol == 'R':
        startc = start[1]
        for c in range(startc, startc + len(vector)):
            worksheet.cell(row=start[0], column=c).value = vector[c - startc]

    elif rowcol == 'C':
        startr = start[0]
        for r in range(startr, startr + len(vector)):
            worksheet.cell(row=r, column=start[1]).value = vector[r - startr]


def fill_xl_with_headers(ws, captions, data, start=(1, 1)):

    fill_xl_row_col(ws, 'R', captions, start)       # Fill Header Row
    fill_xl(ws, data, (start[0] + 1, start[1]))     # Fill the Rest Data


def xl_headers(worksheet, row_title, column_title):

    col_he, row_he = (False, False)

    if row_title is not None:
        row_he = True
    if column_title is not None:
        col_he = True

    if col_he is True and row_he is True:
        fill_xl_row_col(worksheet, 'R', column_title, (1, 2))
        fill_xl_row_col(worksheet, 'C', row_title, (2, 1))
    elif col_he is True:
        fill_xl_row_col(worksheet, 'R', column_title, (1, 1))
    elif row_he is True:
        fill_xl_row_col(worksheet, 'C', row_title, (1, 1))


def xl_dimensions(shape_tuple):

    shape_list = list(shape_tuple)
    rows, columns, tabs = (shape_list[0], 1, 1)

    if len(shape_list) == 3:
        columns = shape_list[1]
        tabs = shape_list[2]

    elif len(shape_list) == 2:
        columns = shape_list[1]

    return rows, columns, tabs
```

```
def measure_xl_3D(fn):
    # Calculate Number of Rows, Columns and Sheets based on First Row and Column size
     on first sheet

    wb = xl.load_workbook(fn)
    ws = wb.worksheets[0]
    total_sheets = len(wb.worksheets)

    r, c = (1, 1)

    # Measure Row
    search = True
    while search is True:
        read = ws.cell(row=r, column=1).value
        if read is not None:
            r += 1
        else:
            search = False

    # Measure Column
    search = True
    while search is True:
        read = ws.cell(row=1, column=c).value
        if read is not None:
            c += 1
        else:
            search = False

    return r - 1, c - 1, total_sheets


def get_xl_row(ws, start, length):

    r_i = start[0]
    c_i = start[1]

    get_tuple = (ws.cell(row=r_i, column=c_i).value,)

    if length > 1:
        for c in range(c_i+1, c_i+length):
            get_tuple += (ws.cell(row=r_i, column=c).value,)

    return get_tuple


def export_nd_xl(row_titles, column_titles, tab_titles, nd_input, row_start=1,
 column_start=1):

    if column_titles is not None:
        row_start = 2

    if row_titles is not None:
        column_start = 2

    dimensions = list(np.shape(nd_input))
    tabs = 1

    if len(dimensions) == 3:
        tabs = dimensions[2]

    wb = xl.Workbook()

    if tabs > 1:
        for i in range(1, tabs):
            wb.create_sheet(str(i))

    for tab in range(0, tabs):
        ws = wb.worksheets[tab]
        ws.title = tab_titles[tab]

        xl_headers(ws, row_titles, column_titles)

        # Slice Results to 1D or 2D table
        if len(dimensions) == 3:
            fill_rows = nd_input[:, :, tab]
        else:
            fill_rows = nd_input

        if len(dimensions) > 1:
            fill_xl(ws, fill_rows, (row_start, column_start))
        else:
            fill_xl_row_col(ws, 'C', fill_rows, (row_start, column_start))

    return wb


def nd_vector(x):

    return_list = []

    for item in x:
        for element in item:
            return_list.append(element)

    return return_list


# POPUP WINDOWS #####################################################################


def popup_open_save(action, folder, window_title, filter_string):
    dlg = QFileDialog()

    dlg.setDirectory(folder)
    if action == 'Save':
        dlg.setFileMode(QFileDialog.AnyFile)
        return dlg.getSaveFileName(dlg, window_title, folder, filter_string)[0]

    elif action == 'Open':
        dlg.setFileMode(QFileDialog.ExistingFile)
        return dlg.getOpenFileName(dlg, window_title, folder, filter_string)[0]
        # Selected filename


def popup_open_multiple(folder, window_title, filter_string):

    dlg = QFileDialog()
    dlg.setDirectory(folder)
    dlg.setFileMode(QFileDialog.ExistingFiles)

    return dlg.getOpenFileNames(dlg, window_title, folder, filter_string)[0]


def popup_message(message_text, icon, title, buttons):
    msg = QMessageBox()

    msg.setStandardButtons(buttons)
    msg.setIcon(icon)
    msg.setText(message_text)
    msg.setWindowTitle(title)

    return msg.exec_()


def popup_simple_dialog(window_title, widgets_table):
    dlg = QDialog()

    dlg.setWindowTitle(window_title)
    layout = QGridLayout()
    dlg.setLayout(layout)

    for r in range(0, len(widgets_table)):
        for c in range(0, len(widgets_table[0])):
            layout.addWidget(widgets_table[r][c], r, c)

    dlg.exec_()


def popup_yes_no_dialog(message_text, icon, window_title):
    dlg = QMessageBox()

    dlg.setIcon(icon)
    dlg.setText(message_text)
    dlg.setWindowTitle(window_title)
    dlg.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)

    return dlg.exec_()
```

```
def popup_select_folder(folder):
    dlg = QFileDialog()
    return dlg.getExistingDirectory(dlg, 'Select Folder', folder,
     QFileDialog.ShowDirsOnly)


# SPECIAL FUNCTIONS (MODULES 2, 3) ##################################################


def all_processed(result_tables):
    for table in result_tables:
        if table is None:
            return False

    return True


def set_shapes(np_tables_list):

    return_list = []

    for res_table in np_tables_list:
        if res_table is not None:
            return_list.append(np.shape(res_table))
        else:
            return_list.append(None)

    return return_list


def global_shape(shapes_list):

    return_value = None

    real_shapes = [shape for shape in shapes_list if shape is not None]
    # Get shape tuples if they exist

    if len(real_shapes) > 0:  # There are some real shapes. Check if all are the same
        unique_shapes = list(set(real_shapes))
        # Remove duplicated shapes. Only one should stay if Consistent

        if len(unique_shapes) == 1:
            return_value = unique_shapes[0]

    return return_value


# MATH/LIST OPERATIONS  ############################################################


def norm_extrap_2D(input_matrix, x_range, n_e, limits=(0.1, 0.9)):
    """ Normalizes or extrapolates 2D numpy array within desired limits and using real
         data ranges. Default interpolation range is 0.1-0.9, because we don’t want
          normalized values to be close to 0 and 1. Input matrix must be 2D numpy
           array """

    m, n = np.shape(input_matrix)
    x_out = np.zeros((m, n))

    for i in range(0, m):
        xmin_r = x_range[i, 0]
        xmax_r = x_range[i, 1]
        delta_x = xmax_r - xmin_r
        delta_y = limits[1] - limits[0]

        for j in range(0, n):

            if n_e == ’N’:
                x_out[i, j] = limits[0] + delta_y / delta_x * (input_matrix[i, j]
                 - xmin_r)  # Normalization

            else:
                x_out[i, j] = xmin_r + delta_x / delta_y * (input_matrix[i, j]
                 - limits[0])  # Extrapolation

    return x_out


def is_number(x):

    try:
        float(x)
        return True

    except ValueError:
        return False


def remove_duplicates(test_list):

    unique_list = []
    search_list = test_list

    while len(search_list) > 0:
        unique_list.append(search_list[0])

        new_search_list = []
        for j in range(1, len(search_list)):

            if search_list[0] != search_list[j]:
                new_search_list.append(search_list[j])

        search_list = new_search_list[:]
    return unique_list


def show_notice(label_object, notice_text):
    label_object.setText(notice_text)
    label_object.repaint()


def xy_series_xl(ws, row_start, col_start, length, series_title, mark_sym,
 title_from_data=False):

    # Creates excel series from {length x 2} excel block of XY data

    x_values = Reference(ws, min_col=col_start, min_row=row_start, max_row=row_start
     + length)

    y_values = Reference(ws, min_col=col_start+1, min_row=row_start, max_row=row_start
     + length)

    new_series = Series(y_values, x_values, title=series_title,
     title_from_data=title_from_data)

    new_series.marker.symbol = mark_sym
    new_series.graphicalProperties.line.noFill = True

    return new_series


def xl_scatter_chart(title, style, x_title, y_title):

    chart = ScatterChart()

    chart.title = title
    chart.style = style
    chart.x_axis.title = x_title
    chart.y_axis.title = y_title

    return chart
```

Main Classes `MyClasses1.py`

```
from pyDOE import lhs
import time
import numpy as np
import os
import openpyxl as xl

from Main_Functions import unique_ndarray, save_object, link_np_tables, folder_size,
 load_object, fill_xl_row_col

from Main_Functions import fill_xl_with_headers, norm_extrap_2D
from Main_ConfigConstants import DSExtension, RecordExtension, DefaultBlueprint


class ModelParameter:

    def __init__(self, mp_id, caption, low, high, step):

        self.MPID = mp_id
        self.Caption = caption
        self.Range = (low, high, step)
        if low == high:
            self.Constant = True
            self.StepValues = [low]
            self.StepCount = 1
        else:
            self.Constant = False
            self.StepValues = [self.Range[0] + j * self.Range[2]
                               for j in range(0, int(round((self.Range[1] -
                                self.Range[0]) / self.Range[2], 0) + 1))]
            self.StepCount = len(self.StepValues)


class ModelParameterList(list):

    @property
    def __NoOfVariables(self):
        return len(self.split_varied())

    @property
    def __InputRanges(self):

        input_ranges_matrix = np.zeros((self.__NoOfVariables, 2))

        for j, mp_range in enumerate([mp.Range for mp in self.split_varied()]):
            input_ranges_matrix[j, 0] = mp_range[0]
            input_ranges_matrix[j, 1] = mp_range[1]

        return input_ranges_matrix

    def uniform_table(self):

        combinations = 1

        for i in [mp.StepCount for mp in self]:
            combinations *= i

        varied_list = self.split_varied()
        # Split between varied and constant columns

        size1 = combinations
        size_blocks = []
        for mp in varied_list:
            size1 = int(size1 / mp.StepCount)
            size_blocks.append(size1)

        variations = []
        for c, mp in enumerate(varied_list):
            block2 = []
            for j in range(0, mp.StepCount):
                for item in [mp.StepValues[j]] * size_blocks[c]:
                    block2.append(item)

            column = []
            for j in range(0, int(combinations / len(block2))):
                for item in block2:
                    column.append(item)
            variations.append(column)

        full_table = np.zeros(shape=(combinations, len(self)))

        for c, mp in enumerate(varied_list):
            full_table[:, mp.MPID] = variations[c]

        for c, mp in enumerate([mp for mp in self if mp.Constant]):
            for row1 in range(0, combinations):
                full_table[:, mp.MPID] = mp.StepValues

        return full_table

    def split_varied(self):
        return [mp for mp in self if not mp.Constant]

    def lhs_table(self, number_of_points, integers_list):

        # Create latin hypercube set using existing data - values are between 0-1
        lhs_01 = lhs(number_of_points, samples=self.__NoOfVariables)
        # {Variables x number_of_points}

        # Extrapolate input parameter matrix within defined input ranges
        data_table_1 = norm_extrap_2D(lhs_01, self.__InputRanges, 'E', limits=(0, 1))

        # Initialize final results table
        result_table = np.zeros(shape=[len(self), number_of_points])

        # Check if parameter datatype is integer -- round data in the table to zero
         decimals (it is still float64!)

        for j, mp_index in enumerate(self.get_var_id()):
            if mp_index in integers_list:
                data_table_1[j, :] = np.round(data_table_1[j, :])

        # Expand data_table_1 to full size using constant values of other parameters.
        constant_mp_list = [mp for mp in self if mp.Constant]
        for id_value in zip([mp.MPID for mp in constant_mp_list], [mp.StepValues for
         mp in constant_mp_list]):
            result_table[id_value[0], :] = id_value[1]*number_of_points
            # Assign constant values

        # Add varied values to result_table from data_table_1
        for r, row in enumerate(data_table_1):
            result_table[self.get_var_id()[r], :] = row[:]

        # Transpose result_table and return it - final table is shaped
        {num_of_pts x num_of_parameters}

        return np.transpose(result_table)

    def get_captions(self):
        return [mp.Caption for mp in self]

    def get_var_id(self):
        return [mp.MPID for mp in self.split_varied()]

    def get_var_captions(self):
        return [mp.Caption for mp in self.split_varied()]


class Simulation:

    def __init__(self, input_vector, sim_index, meta_status, fn_prefix):

        self.InputParameters = list(input_vector)
        self.SimID = sim_index

        self.MetaStatus = meta_status
        # Training or Test

        self.Filename = fn_prefix + '_' + str(self.SimID)

        self.ComputationTime = 0.0
        # Updated in ***update_sim_from_record***

        self.DataUsage = 0.0
        self.Nodes = 0
        self.Elements = 0

        self.Calculated = False
        self.Processed = False
        # Updated in ***update_sim_from_record***

        self.SimResultsTable = None

        self.CalculationLog = RecordsList()

    def update_sim_records(self, records):

        self.CalculationLog += records
        # Update sim with found records
        last_results = self.CalculationLog.last_calc_data()[2]

        last_proc_results = self.CalculationLog.last_proc_res()

        if last_results is not None:
            self.Calculated, self.Nodes, self.Elements, self.ComputationTime,
             self.DataUsage = last_results

        if last_proc_results is not None:
            self.Processed, self.SimResultsTable = last_proc_results

    def datatable_2(self):

        last_calculation_data = self.CalculationLog.last_calc_data()

        return [self.Filename, self.Calculated, self.Processed,
         last_calculation_data[0], last_calculation_data[1]]


class Dataset:

    def __init__(self, mp_list, scenario_type, description, blueprint):

        # Upon initialization, only model parameter list is provided. Other properties
         are set to initial zero values

        self.__Parameters = mp_list
        self.__ScenarioType = scenario_type
        self.__Description = description
        self.__ModellingSequence = blueprint
        self.__DataTable = None
        self.__MyName = ''

        self.__Simulations = []

        self.CurrentPath = ''
        # This is temporary attribute, which changes each time when DS is loaded

        self.SnapMatrices = []

    @property
    def __Captions(self):
        return self.__Parameters.get_captions()

    @property
    def __NotCalculatedSims(self):
        return [sim for sim in self.__Simulations if not sim.Calculated]

    @property
    def __CalculatedSims(self):
        return [sim for sim in self.__Simulations if sim.Calculated]
```

```
@property
def __ProcessedSims(self):
    return [sim for sim in self.__Simulations if sim.Processed]

@property
def __NotProcessedSims(self):
    return [sim for sim in self.__CalculatedSims if not sim.Processed]

@property
def __LocalSubfolder(self):
    return self.CurrentPath[0:(len(self.CurrentPath)-len(DSExtension))]

@property
def SummaryTable(self):

    # Calculate number of simulations for display (for example: 12+5 in MM case)
    if self.__ScenarioType == 'Metamodel':

        train = len([sim for sim in self.__Simulations if sim.MetaStatus ==
         'Training'])

        test = len([sim for sim in self.__Simulations if sim.MetaStatus ==
         'Test'])

        sims_no = str(train) + '+' + str(test)

    else:

        sims_no = str(len(self.__Simulations))

    block1 = [self.__ScenarioType,
              self.__Description,
              self.__ModellingSequence,
              sims_no]

    table2 = [self.__Captions,
              self.__DataTable]

    SimTableHeaders = ['Simulation', 'Calculated', 'Processed', 'Computation Time
     [min]', 'Size on Disk [MB]', 'Metamodel Status', 'Mesh nodes', 'Mesh
      elements']

    sim_data = [[sim.Filename,
                 sim.Calculated,
                 sim.Processed,
                 sim.ComputationTime,
                 sim.DataUsage,
                 sim.MetaStatus,
                 sim.Nodes,
                 sim.Elements] for sim in self.__Simulations]

    table1 = [SimTableHeaders, sim_data]
```

```
        return [block1, table1, table2]

    @property
    def SummaryTable_2(self):

        block1 = [self.__ScenarioType,
                  self.__Description,
                  self.__ModellingSequence]

        table1 = [['Simulation', 'Calculated', 'Processed', 'Last Calculation PC',
         'Last Calculation Path'], [sim.datatable_2() for sim in self.__Simulations]]

        return [block1, table1]

    def SummaryTable_2_1(self, list_indices):

        captions = ['Simulation', 'Action', 'Computer', 'Path']
        data = []
        for sim in self.__get_sims_by_id(list_indices):
            sim_block = [sim.Filename, 'Awaiting start...', 'None', 'None']
            data.append(sim_block)

        return [data, captions]

    @property
    def SummaryTable_2_2(self):

        # Calculate number of simulations for display (for example: 12+5 in MM case)
        SimTableHeaders = ['Simulation', 'Computation Time [min]', 'Size on Disk
         [MB]', 'Mesh nodes', 'Mesh elements', 'Processed']

        sim_data = [[sim.Filename,
                     sim.ComputationTime,
                     sim.DataUsage,
                     sim.Nodes,
                     sim.Elements,
                     sim.Processed] for sim in self.__Simulations]

        sim_results = [sim.SimResultsTable for sim in self.__Simulations]

        return SimTableHeaders, sim_data, sim_results

    @property
    def SummaryTable_3(self):

        block1 = [self.__ScenarioType, self.__Description,
         str(len(self.__Simulations))]

        return block1, [[sim.SimResultsTable, sim.MetaStatus] for sim in
         self.__Simulations], self.__varied_table()

    @property
    def SummaryTable_4(self):
        return [[sim.InputParameters, sim.SimResultsTable] for sim in
         self.__Simulations]

    def __varied_table(self):

        varied_indices = self.__Parameters.get_var_id()
        varied_captions = self.__Parameters.get_var_captions()

        if len(varied_indices) == 0:
            return None
        else:
            rows = len(varied_indices)
            columns = len(self.__Simulations)

            table = np.zeros((rows, columns))

            for r in range(0, rows):
                table[r, :] = self.__DataTable[:, varied_indices[r]]

            return table, varied_captions

    def __save(self):
        save_object(self, self.CurrentPath)          # Save into File

    def __save_task_record(self, record_and_sim):

        rec_list = [file for file in os.listdir(self.__LocalSubfolder)
        if file.endswith(RecordExtension)]

        start = 0

        if len(rec_list) > 0:
        # Search for first available index if there are other records

            indices = [int(fname[0:(len(fname) - len(RecordExtension))].split('_')[1])
             for fname in rec_list]

            indices.sort()

            found = False
            while not found:
                if start not in indices:
                # Find first free index, start from 0, and assign it to the record
                 file name

                    found = True
                else:
                    start += 1

        # Record index determined. Save record to dataset subfolder
        save_object(record_and_sim, self.__LocalSubfolder + '\\' + 'Record_' +
         str(start) + RecordExtension)
```

```
def __get_sims_by_id(self, id_list):
    return [sim for sim in self.__Simulations if sim.SimID in id_list]

def get_sim_by_index(self, i):
    return self.__Simulations[i]

def __remote_path(self, remote_folder_path):
    return remote_folder_path + '\\' + self.__MyName + '----'

def save_first(self, full_path):

    just_name_ext = os.path.basename(full_path)

    self.__MyName = just_name_ext[0:(len(just_name_ext) - len(DSExtension))]
    self.CurrentPath = full_path

    self.__save()

def assign_datatable(self, arguments, sim_fn_prefix):

    training_table = None
    test_table = None
    meta_status = None

    if self.__ScenarioType == 'Simple':
        self.__DataTable = self.__Parameters.uniform_table()
        meta_status = [None] * len(self.__DataTable)

    elif self.__ScenarioType == 'Custom':
        self.__DataTable = arguments
        meta_status = [None] * len(self.__DataTable)

    elif self.__ScenarioType == 'Metamodel':

        if len(self.__Parameters.split_varied()) > 0:
        # Check number of variable parameters

            training_pts, test_pts, training_alg, test_alg = arguments[0]
            integers_list = arguments[1]

            if training_alg == 'Latin Hypercube Sampling':
                training_table = self.__Parameters.lhs_table(training_pts,
                 integers_list)     # Calculate LHS Sample

            training_unique = unique_ndarray(training_table)[0]
             # Get unique values

            if test_pts > 0:
                if test_alg == 'Latin Hypercube Sampling':
                    test_table = self.__Parameters.lhs_table(test_pts,
                     integers_list)         # Calculate LHS Sample

                test_unique = unique_ndarray(test_table)[0]
                final_table = link_np_tables(training_unique, test_unique)
                # Link two tables into single dataset

                meta_status = ['Training']*len(training_unique) +
                 ['Test']*len(test_unique)

            else:
                final_table = training_unique
                meta_status = ['Training'] * len(training_unique)

            self.__DataTable = final_table
        else:
            return False
            # All parameters are constants - dataset cannot be created.
            Exit function

    self.__Simulations = [Simulation(vector, sim_id, meta_status[sim_id],
     sim_fn_prefix)

                          for sim_id, vector in enumerate(self.__DataTable)]
    return True

def export_SummaryTable_1(self, export_filename):

    wb = xl.Workbook()

    wb.create_sheet('Basics')
    fill_xl_row_col(wb.worksheets[1], 'R', ['Dataset Type', 'Dataset Description',
                                            'Modelling Sequence', 'Total Number
                                            of Simulations'])

    fill_xl_row_col(wb.worksheets[1], 'R', self.SummaryTable[0], start=(2, 1))

    wb.create_sheet('Simulations Info')
    fill_xl_with_headers(wb.worksheets[2], self.SummaryTable[1][0],
     self.SummaryTable[1][1])

    wb.create_sheet('Input Values')
    fill_xl_with_headers(wb.worksheets[3], self.SummaryTable[2][0],
     self.SummaryTable[2][1], start=(1, 2))

    fill_xl_row_col(wb.worksheets[3], 'C',
                    ['Simulation/Parameter'] + [item[0]
                    for item in self.SummaryTable[1][1]])

    wb.remove_sheet(wb.worksheets[0])

    wb.save(export_filename)

def filter(self, selected_indexes, io):

    if io == 'Input':
        sims_to_run = [sim.SimID for sim in self.__NotCalculatedSims
        if sim.SimID in selected_indexes]

        sims_to_exclude = [sim.SimID for sim in self.__CalculatedSims
        if sim.SimID in selected_indexes]

    else:
        sims_to_run = [sim.SimID for sim in self.__NotProcessedSims
        if sim.SimID in selected_indexes]

        sims_to_exclude = [sim.SimID for sim in self.__ProcessedSims
        if sim.SimID in selected_indexes]

    return sims_to_run, sims_to_exclude

def start_local(self, sim_index, pc_id, sim_file_path, job_type):

    sim = self.__Simulations[sim_index]

    blueprint_path = self.__LocalSubfolder + '\\' + DefaultBlueprint

    if job_type == 'Calculation':

        # Define path to save calculation file
        sim_file_path = self.__LocalSubfolder + '\\' + sim.Filename

        # Initialize timer and data space
        sim.ComputationTime = float(time.time())
        sim.DataUsage = float(folder_size(self.__LocalSubfolder))

    self.__save_task_record((sim_index, Record(pc_id, sim_file_path, None,
     job_type, 'Begin', 'Local')))

    return sim.InputParameters, sim_file_path, blueprint_path

def end_local(self, sim_index, pc_id, results, sim_file_path, job_type):

    sim = self.__Simulations[sim_index]

    if job_type == 'Calculation':

        # Define path to save calculation file
        sim_file_path = self.__LocalSubfolder + '\\' + sim.Filename

        # Calculate data usage and computation time
        comp_time = (time.time() - sim.ComputationTime)/60
        data_usage = (folder_size(self.__LocalSubfolder) - sim.DataUsage)/1048576

        results = (results[0], results[1], results[2], comp_time, data_usage)

    self.__save_task_record((sim_index, Record(pc_id, sim_file_path, results,
     job_type, 'End', 'Local')))
```

```
def start_remote(self, sim_index, pc_id, job_type, remote_folder_path):

    sim = self.__Simulations[sim_index]

    blueprint_path = self.__remote_path(remote_folder_path) + DefaultBlueprint
    sim_file_path = self.__remote_path(remote_folder_path) + sim.Filename

    if job_type == 'Calculation':
        sim.ComputationTime = float(time.time())
        sim.DataUsage = 0.0

    self.__save_task_record((sim_index, Record(pc_id, sim_file_path, None,
     job_type, 'Begin', 'Remote')))

    return sim.InputParameters, sim_file_path, blueprint_path

def end_remote(self, sim_index, pc_id, results, remote_folder_path, job_type):

    sim = self.__Simulations[sim_index]

    sim_file_path = self.__remote_path(remote_folder_path) + sim.Filename

    if job_type == 'Calculation':
        results = (results[0], results[1], results[2], (time.time() -
         sim.ComputationTime)/60, 0.0)

    self.__save_task_record((sim_index, Record(pc_id, sim_file_path, results,
     job_type, 'End', 'Remote')))

def update_records(self):

    # Get list of files containing previous records. Continue if found

    saved_records_list = [file for file in os.listdir(self.__LocalSubfolder)
    if file.endswith(RecordExtension)]

    if len(saved_records_list) > 0:

        # Load simulation instances from records
        records_sims = [load_object(self.__LocalSubfolder + '\\' + record_fn)
        for record_fn in saved_records_list]

        for sim in self.__Simulations:
            sim.update_sim_records([rec_set[1] for rec_set in records_sims
            if sim.SimID == rec_set[0]])

        # Erase record files, because we don't need them anymore
        for file in [(self.__LocalSubfolder + '\\' + record_fn)
        for record_fn in saved_records_list]:

            os.remove(file)

        self.__save()     # Save dataset after changes
    def change_results(self, changed_set):
        self.__Simulations[changed_set[0]].Processed = True
        self.__Simulations[changed_set[0]].SimResultsTable = changed_set[1]
        self.__save()

    def add_snapshot_matrices(self, added_matrices):
        for matrix in added_matrices:
            self.SnapMatrices.append(matrix)
        self.__save()

    def update_from_child(self, child_ds, update_list):

        for sim_id in update_list:
            child_sim = child_ds.get_sim_by_index(sim_id)
            self.__Simulations[sim_id] = child_sim

        self.__save()

    def get_name(self):
        return self.__MyName


class Record:

    def __init__(self, pc_id, sim_path, results_tuple, analysis_type, start_stop,
     session):

        self.ComputerID = pc_id
        self.SimPath = sim_path
        self.Results = results_tuple

        self.AnalysisType = analysis_type
        self.StartStop = start_stop

        self.__Session = session
        self.__Time = time.time()


class RecordsList(list):

    def __last_ended(self, calc_proc):

        records = [rec for rec in self if rec.AnalysisType == calc_proc and
         rec.StartStop == 'End']

        if len(records) > 0:
        # Return last record (if there are any records)

            return records[-1]
        else:
            return Record(None, None, None, None, None, None)
    def last_calc_data(self):

        last_rec = self.__last_ended('Calculation')
        return last_rec.ComputerID, last_rec.SimPath, last_rec.Results

    def last_proc_res(self):
        return self.__last_ended('Processing').Results


class SnapshotMatrix:

    def __init__(self, data_table, captions):
        self.Data = data_table
        self.Captions = captions
```

User Interface Classes `Main_WidgetClasses.py`

```
import os
from Main_ConfigConstants import DefaultSimDir, SimExtension, DefaultServerName

from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIcon

from PyQt5.QtWidgets import QTableWidgetItem, QTableWidget, QAbstractScrollArea,
 QPushButton, QLineEdit, QMessageBox

from PyQt5.QtWidgets import QLabel, QGridLayout, QDialog, QWidget

from Main_ConfigConstants import IconTrue, IconFalse

from Main_Functions import popup_message, load_object, popup_open_save


def cell_label_button(value_cell, button_widget, color=None):
    cell_widget = QWidget()
    cell_layout = QGridLayout()
    cell_widget.setLayout(cell_layout)

    label = QLabel(str(value_cell))
    if color is not None:
        label.setStyleSheet('color: ' + color)

    cell_layout.addWidget(label, 0, 0)
    cell_layout.addWidget(button_widget, 0, 1)

    cell_layout.setAlignment(Qt.AlignCenter)
    cell_layout.setContentsMargins(0, 0, 0, 0)

    return cell_widget
```

```
def fill_table_items(table_widget, values, start=(0, 0)):

    # Fills the table widget with values from the list, starting from desired cell.
    # Booleans are filled with icons

    startr = start[0]
    startc = start[1]

    for r in range(startr, startr + len(values)):
        for c in range(startc, startc + len(values[0])):
            if type(values[r - startr][c - startc]) == bool:
                table_item = IconCell(values[r - startr][c - startc])
            else:
                table_item = StringCell(str(values[r - startr][c - startc]))

            table_widget.setItem(r, c, table_item)


def create_simple_table(data_table, captions):

    # Creates QTableWidgets with captions

    table_widget = QTableWidget(len(data_table), len(captions))
    table_widget.setHorizontalHeaderLabels(captions)
    table_widget.horizontalHeader().setVisible(True)
    table_widget.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

    fill_table_items(table_widget, data_table)

    table_widget.resizeColumnsToContents()
    table_widget.clearSelection()

    return table_widget


def load_dataset_sim_table(layout, path):

    ds = load_object(path)  # Reloading dataset and widgets
    ds.CurrentPath = path

    table = ActiveTable_1(ds.SummaryTable_2[1][1], ds.SummaryTable_2[1][0])
    layout.addWidget(table, 0, 0)

    return ds, table


class IconCell(QTableWidgetItem):

    def __init__(self, result):
        super().__init__()

        if result is True:
            icon = QIcon(IconTrue)
        else:
            icon = QIcon(IconFalse)

        self.setIcon(icon)

        self.setTextAlignment(Qt.AlignCenter)
        self.setFlags(Qt.ItemIsEnabled | Qt.ItemIsSelectable)


class StringCell(QTableWidgetItem):

    def __init__(self, string):
        super().__init__(string)

        self.setTextAlignment(Qt.AlignCenter)
        self.setFlags(Qt.ItemIsEnabled | Qt.ItemIsSelectable)


class Input_ExcelCaptions(QDialog):

    def __init__(self, parameters_number, title, caption1):

        super(Input_ExcelCaptions, self).__init__()

        self.setWindowTitle(title)

        layout = QGridLayout()
        self.setLayout(layout)

        cmdOK = QPushButton('OK')
        cmdOK.clicked.connect(self.cmdOK_clicked)

        layout.addWidget(QLabel(caption1), 0, 0)
        layout.addWidget(QLabel('Unit'), 0, 1)
        layout.addWidget(cmdOK, parameters_number+2, 1)

        self.TextFields = []
        for r in range(0, parameters_number):
            p_i = QLineEdit(caption1 + '_' + str(r+1))
            u_i = QLineEdit('Unit_' + str(r+1))

            layout.addWidget(p_i, r+1, 0)
            layout.addWidget(u_i, r+1, 1)

            self.TextFields.append((p_i, u_i))

        self.ReturnValues = None

        self.exec_()

    def cmdOK_clicked(self):

        data_empty = False
        for param_set in self.TextFields:       # Check if some fields are empty
            for input_widget in param_set:
                if input_widget.text() == '':
                    data_empty = True
                    popup_message('Please fill all fields!', QMessageBox.Critical,
                     'Input Error', QMessageBox.Ok)

                    break

        if not data_empty:
            self.ReturnValues = [(p[0].text(), p[1].text()) for p in self.TextFields]
            self.close()


class ActiveTable_1(QTableWidget):

    def __init__(self, table_data, table_captions):
        super().__init__(len(table_data), len(table_captions))

        # Observer Variables that monitor the content of table columns
        self.LinkedPaths = [item[4] for item in table_data]
        self.LinkedComputers = [item[3] for item in table_data]
        self.ChangedPaths = []
        self.LinkButtons = []
        self.CalculatedList = [item[1] for item in table_data]
        self.ProcessedList = [item[2] for item in table_data]

        self.setHorizontalHeaderLabels(table_captions)
        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        fill_table_items(self, table_data)

        # Check if all local paths exist on local pc. For remote computer this is not
         possible at this moment.

        self.InvalidLocalPaths = []
        for j, calc_path in enumerate(self.LinkedPaths):
            if calc_path is not None:
                if not os.path.isfile(calc_path + SimExtension) and
                 self.LinkedComputers[j] == DefaultServerName:

                    self.InvalidLocalPaths.append(j)

        # Create link buttons
        for r in range(0, len(self.LinkedPaths)):
            button = QPushButton('Link calculated')
            button.clicked.connect(self.link_button_clicked)
            self.LinkButtons.append(button)

            self.setItem(r, 4, StringCell(''))
            # Erase previous cell content and add new widget
            if r in self.InvalidLocalPaths:
                color = 'red'
            else:
                color = None

            self.setCellWidget(r, 4, cell_label_button(self.LinkedPaths[r], button,
             color))

        self.clearSelection()
        self.resizeColumnsToContents()

    def link_button_clicked(self):

        # Find which button was clicked
        i = 0
        for i, button in enumerate(self.LinkButtons):
            if self.sender() == button:
                break  # Button found

        link_fn = popup_open_save('Open', DefaultSimDir, 'Select Simulation',
         'Simulations (*' + SimExtension + ')')

        if link_fn != '':
            self.__add_linked(i, link_fn)

    def link_multiple(self, files_list):
        for i, file in zip(self.SelectedRows, files_list):
            self.__add_linked(i, file)

    def __add_linked(self, i, link_filename):

        # Update Table Cell
        self.setItem(i, 3, StringCell(DefaultServerName))
        # Set computer cell to local Computer

        self.setItem(i, 4, StringCell(''))
        # Erase previous cell content and add new widget

        self.setCellWidget(i, 4, cell_label_button(link_filename, self.LinkButtons[i],
         'blue'))

        self.resizeColumnsToContents()

        # Update observer variables
        self.LinkedPaths[i] = link_filename
        self.LinkedComputers[i] = DefaultServerName
        self.ChangedPaths.append(i)
        self.CalculatedList[i] = True

        if i in self.InvalidLocalPaths:
            self.InvalidLocalPaths.remove(i)
    @property
    def SelectedRows(self):
        return sorted(set(i.row() for i in self.selectedIndexes()))

    @property
    def SelectedLinked(self):
        return [i for i in list(set(self.ChangedPaths)) if i in self.SelectedRows]

    @property
    def NeededComputers(self):
        return list(set([self.LinkedComputers[j] for j in self.SelectedRows]))

    @property
    def SelectedInvalidPaths(self):
        return [j for j in self.SelectedRows if j in self.InvalidLocalPaths]

    @property
    def ProcessingQueue(self):
        # Here we sort simulations according to linked computer. Each computer gets
         its own processing sim Queue

        # We need to exclude NOT CALCULATED SIMS (except if they are linked outside)
        return [(computer, [j for j in self.SelectedRows if self.LinkedComputers[j]
         == computer])
                for computer in self.NeededComputers]

    def get_paths(self, sim_indices):
        return [self.LinkedPaths[j] for j in sim_indices]

    def selected_calculated(self):
        ok = True
        for j in self.SelectedRows:
            if self.CalculatedList[j] is False:
                ok = False
                break
        return ok
```

User Interface Classes - 2 Main_WidgetClasses_small.py

```
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIcon

from PyQt5.QtWidgets import QTableWidgetItem, QTableWidget, QAbstractScrollArea,
 QPushButton, QMdiSubWindow, QWidget

from PyQt5.QtWidgets import QLabel, QGridLayout, QDialog, QLineEdit, QMessageBox
from Main_ConfigConstants import IconTrue, IconFalse
from Main_Functions import popup_message


def fill_table_items(table_widget, values, start=(0, 0)):
    # Fills the QTableWidget with values from the list, starting from desired cell.
    # Booleans are filled with icons

    startr = start[0]
    startc = start[1]

    for r in range(startr, startr + len(values)):
        for c in range(startc, startc + len(values[0])):
            if type(values[r - startr][c - startc]) == bool:
                table_item = IconCell(values[r - startr][c - startc])
            else:
                table_item = StringCell(str(values[r - startr][c - startc]))

            table_widget.setItem(r, c, table_item)


def create_simple_table(data_table, captions):

    # Creates QTableWidgets with captions

    table_widget = QTableWidget(len(data_table), len(captions))
    table_widget.setHorizontalHeaderLabels(captions)
    table_widget.horizontalHeader().setVisible(True)
    table_widget.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

    fill_table_items(table_widget, data_table)

    table_widget.resizeColumnsToContents()
    table_widget.clearSelection()

    return table_widget


def new_subwindow(title, dimensions):
    new_window = QMdiSubWindow()
    top_widget = QWidget()
    new_layout = QGridLayout(top_widget)
    top_widget.setLayout(new_layout)
    new_window.setWindowTitle(title)
    new_window.setFixedSize(dimensions[0], dimensions[1])
    new_window.setWindowFlags(Qt.CustomizeWindowHint)
    new_window.setWidget(top_widget)
    return new_window, new_layout


class IconCell(QTableWidgetItem):

    def __init__(self, result):
        super().__init__()

        if result is True:
            icon = QIcon(IconTrue)
        else:
            icon = QIcon(IconFalse)

        self.setIcon(icon)

        self.setTextAlignment(Qt.AlignCenter)
        self.setFlags(Qt.ItemIsEnabled | Qt.ItemIsSelectable)


class StringCell(QTableWidgetItem):

    def __init__(self, string):
        super().__init__(string)

        self.setTextAlignment(Qt.AlignCenter)
        self.setFlags(Qt.ItemIsEnabled | Qt.ItemIsSelectable)


class Input_ExcelCaptions(QDialog):

    def __init__(self, parameters_number, title, caption1):

        super(Input_ExcelCaptions, self).__init__()

        self.setWindowTitle(title)

        layout = QGridLayout()
        self.setLayout(layout)

        cmdOK = QPushButton('OK')
        cmdOK.clicked.connect(self.cmdOK_clicked)

        layout.addWidget(QLabel(caption1), 0, 0)
        layout.addWidget(QLabel('Unit'), 0, 1)
        layout.addWidget(cmdOK, parameters_number+2, 1)

        self.TextFields = []
        for r in range(0, parameters_number):
            p_i = QLineEdit(caption1 + '_' + str(r+1))
            u_i = QLineEdit('Unit_' + str(r+1))

            layout.addWidget(p_i, r+1, 0)
            layout.addWidget(u_i, r+1, 1)

            self.TextFields.append((p_i, u_i))

        self.ReturnValues = None

        self.exec_()

    def cmdOK_clicked(self):

        data_empty = False
        for param_set in self.TextFields:       # Check if some fields are empty
            for input_widget in param_set:
                if input_widget.text() == '':
                    data_empty = True
                    popup_message('Please fill all fields!', QMessageBox.Critical,
                     'Input Error', QMessageBox.Ok)
                    break

        if not data_empty:
            self.ReturnValues = [(p[0].text(), p[1].text()) for p in self.TextFields]
            self.close()
```

Module 1 - Calculation Scenario Maker `M1.py`

```
import sys
import openpyxl as xl
import numpy as np
import shutil
import os

from PyQt5 import uic
from PyQt5.QtWidgets import QMessageBox, QLabel, QPushButton, QLineEdit, QDialog,
 QGridLayout, QComboBox, QTableWidget

from PyQt5.QtWidgets import QWidget, QTableWidgetItem, QMainWindow, QVBoxLayout,
 QTabWidget, QApplication

from MyClasses1 import ModelParameter, ModelParameterList, Dataset
from Main_WidgetClasses import create_simple_table, Input_ExcelCaptions

from M1_WidgetClasses import SummaryTableDS

from Main_ConfigConstants import GUI_file1, DSExtension, DefaultSimDir,
 ParametersConfig, MinimumMMTrain
from Main_Functions import unique_ndarray, load_object, popup_open_save,
 popup_simple_dialog
from Main_Functions import popup_message, get_xl_row, parse_xl_nd, is_number

Ui_MainWindow, QtBaseClass = uic.loadUiType(GUI_file1)


class Input_Form_MM(QDialog):

    def __init__(self):
        super(Input_Form_MM, self).__init__()

        self.setWindowTitle('Metamodel Data Input')

        self.txtTrainP = QLineEdit(str(MinimumMMTrain))
        self.txtTestP = QLineEdit('1')
        self.cmbTrainP = QComboBox()
        self.cmbTestP = QComboBox()
        self.cmbTrainP.addItems(['Latin Hypercube Sampling'])
        self.cmbTestP.addItems(['Latin Hypercube Sampling'])
        cmdOK = QPushButton('OK')
        cmdOK.clicked.connect(self.cmdOK_clicked)

        layout = QGridLayout()
        layout.addWidget(QLabel('Sampling Algorithm'), 0, 2)
        layout.addWidget(QLabel('Number of Training Points: '), 1, 0)
        layout.addWidget(self.txtTrainP, 1, 1)
        layout.addWidget(self.cmbTrainP, 1, 2)
        layout.addWidget(QLabel('Number of Test Points: '), 2, 0)
        layout.addWidget(self.txtTestP, 2, 1)
        layout.addWidget(self.cmbTestP, 2, 2)
        layout.addWidget(cmdOK, 3, 2)

        self.setLayout(layout)
        self.ReturnValues = None        # Return values container variable
        self.exec_()

    def cmdOK_clicked(self):
        training_pts = int(self.txtTrainP.text())
        # Obtain data from input fields

        test_pts = int(self.txtTestP.text())

        # Check number of Training and Test Points.
        #If TestPoints is 0, notify and use training points as test points

        if training_pts < MinimumMMTrain:
            popup_message('Invalid number of Training points. Minimum is ' +
             str(MinimumMMTrain) + '!',
                          QMessageBox.Critical, 'Input Error', QMessageBox.Ok)
        else:
            if test_pts == 0:
                popup_message('Test points not selected! Training points will be
                 used for Metamodel test!',
                              QMessageBox.Information, 'Information',
                               QMessageBox.Ok)
            self.ReturnValues = (training_pts, test_pts,
                                 str(self.cmbTrainP.currentText()),
                                  str(self.cmbTestP.currentText()))
                                  # Return Values
            self.close()


class Input_Form_CustomDS(QDialog):

    def __init__(self):
        super(Input_Form_CustomDS, self).__init__()

        self.setWindowTitle('Custom Dataset Input')
        layout = QGridLayout()

        self.txtRows = QLineEdit('2')
        self.txtColumns = QLineEdit('2')
        cmdBrowseTable = QPushButton('Select Excel File')
        self.cmdOK = QPushButton('OK')
        self.cmdOK.setEnabled(False)
        self.lblFilename = QLabel('*** Data Table not loaded ***')
        self.lblFilename.setEnabled(False)

        layout.addWidget(QLabel('Number of Rows (Simulations): '), 0, 0)
        layout.addWidget(self.txtRows, 0, 1)
        layout.addWidget(QLabel('Number of Columns
        (Input Parameter Values): '),1, 0)
        layout.addWidget(self.txtColumns, 1, 1)
        layout.addWidget(self.lblFilename, 2, 0)
        layout.addWidget(cmdBrowseTable, 2, 1)
        layout.addWidget(self.cmdOK, 3, 1)

        self.cmdOK.clicked.connect(self.cmdOK_clicked)
        cmdBrowseTable.clicked.connect(self.cmdBrowseTable_clicked)

        self.setLayout(layout)
        self.ReturnValues = None
        self.FileName = ''
        self.exec_()

    def cmdOK_clicked(self):

        def __message_duplicated_row(num_r):
            sentence1 = 'Data Table succesfully loaded. '
            string2 = ''

            if num_r == 1:
                string2 = 'One duplicated row was removed.'
            elif num_r > 1:
                string2 = str(num_r) + ' duplicated rows were removed.'

            return sentence1 + string2

        no_errors = True        # Error catcher
        try:
            rows = int(self.txtRows.text())
            cols = int(self.txtColumns.text())

            # No ValueError found.
            if rows <= 0 or cols <= 0:      # Rows or columns equal to 0
                popup_message('Invalid number of rows or columns',
                 QMessageBox.Critical, 'Input Error', QMessageBox.Ok)
                no_errors = False

            if no_errors:
            # Rows and Columns ok. Open the excel file.
```

```
                parsed_xl = parse_xl_nd(self.FileName, rows, cols)
                # FileName exists, because button is Enabled

                if parsed_xl is not None:
                    unique_data_table, erased = unique_ndarray(parsed_xl)
                    # Erase duplicated rows and notify user.
                    popup_message(__message_duplicated_row(erased),
                     QMessageBox.Information,
                                  'Information', QMessageBox.Ok)

                    captions_dialog = Input_ExcelCaptions(cols,
                    'Input Parameter Descriptions',
                    'Parameter').ReturnValues
                    if captions_dialog is not None:
                        self.ReturnValues = unique_data_table, captions_dialog
                        self.close()
                else:
                    popup_message('Error in the Excel file. Please check the
                     data!',
                     QMessageBox.Critical, 'Excel Table Error', QMessageBox.Ok)

        except ValueError:
            # No integers entered. Error catched during text to int conversion
            popup_message('Invalid number of rows or columns!',
             QMessageBox.Critical, 'Input Error', QMessageBox.Ok)

    def cmdBrowseTable_clicked(self):

        filename = popup_open_save('Open', DefaultSimDir, 'Open Excel Table',
         'Excel files (*.xls *.xlsx)')

        if filename != '':
            self.lblFilename.setText(filename)
            self.lblFilename.repaint()
            self.lblFilename.setEnabled(True)
            self.FileName = filename
            self.cmdOK.setEnabled(True)


class Input_Form_DatasetBasic(QDialog):

    def __init__(self):
        super(Input_Form_DatasetBasic, self).__init__()

        self.setWindowTitle('Dataset Description')
        layout = QGridLayout()

        self.cmbDSType = QComboBox()
        self.cmbDSType.addItems(['Simple', 'Metamodel', 'Custom'])
        self.txtFilename = QLineEdit('SIMULATION')
        self.cmbBlueprint = QComboBox()
        self.cmbBlueprint.addItems(['Full Model Generation',
        'Use Model Blueprint'])
        self.txtDescription = QLineEdit('Dataset Description...')
        cmdOK = QPushButton('OK')

        layout.addWidget(QLabel('Dataset Type'), 0, 0)
        layout.addWidget(QLabel('Modelling Sequence'), 1, 0)
        layout.addWidget(QLabel('Simulation File Prefix'), 2, 0)
        layout.addWidget(QLabel('Dataset Description'), 3, 0)
        layout.addWidget(self.cmbDSType, 0, 1)
        layout.addWidget(self.cmbBlueprint, 1, 1)
        layout.addWidget(self.txtFilename, 2, 1)
        layout.addWidget(self.txtDescription, 3, 1)
        layout.addWidget(cmdOK, 4, 1)

        cmdOK.clicked.connect(self.cmdOK_clicked)

        self.setLayout(layout)
        self.ReturnValues = None
        self.exec_()

    def cmdOK_clicked(self):
        filename_string = self.txtFilename.text()

        if len(filename_string.split()) != 1:
        # Check number of words for simulation prefix
            popup_message('Use single word as filename prefix!',
             QMessageBox.Critical, 'Input Error', QMessageBox.Ok)

        else:
            self.ReturnValues = (self.cmbDSType.currentText(),
             self.cmbBlueprint.currentText(), filename_string,
                                 self.txtDescription.text())
            self.close()


class Summary_Form_Dataset(QDialog):

    def __init__(self, all_data, state):
        super(Summary_Form_Dataset, self).__init__()

        buttonok = 'Accept and Save'
        buttonno = 'Decline'

        self.TableData = (all_data[1], all_data[2])
        # Data Tables
        self.PopupArgs = ['Save', DefaultSimDir, 'Save Dataset As',
        'Dataset files (*.dat)']

        self.setWindowTitle('Dataset Summary - NOT SAVED!')

        layout = QGridLayout()

        if state[0] == 'Loaded':
```

```
        buttonok = 'Export to Excel'
        buttonno = 'Unload'
        self.setWindowTitle('LOADED DATASET - ' + state[1])
        self.PopupArgs = ['Save', DefaultSimDir, 'Save Excel Table As',
        'Excel files (*.xlsx)']

    self.cmdDataTable = QPushButton('Input Values Table')
    self.cmdSimsTable = QPushButton('Simulations Data Table')

    layout.addWidget(QLabel('Dataset Type'), 0, 0)
    layout.addWidget(QLabel('Dataset Description'), 1, 0)
    layout.addWidget(QLabel('Modelling Sequence'), 2, 0)
    layout.addWidget(QLabel('Total Number of Simulations'), 3, 0)

    for j in range(0, 4):
    # Fill label values
        layout.addWidget(QLabel(all_data[0][j]), j, 1)

    self.cmdOK = QPushButton(buttonok)
    self.cmdCancel = QPushButton(buttonno)

    layout.addWidget(self.cmdSimsTable, 4, 0)
    layout.addWidget(self.cmdDataTable, 4, 1)
    layout.addWidget(self.cmdOK, 5, 0)
    layout.addWidget(self.cmdCancel, 5, 1)

    self.cmdOK.clicked.connect(self.cmdOK_clicked)
    self.cmdCancel.clicked.connect(self.cmdCancel_clicked)
    self.cmdDataTable.clicked.connect(self.cmdDataTable_clicked)
    self.cmdSimsTable.clicked.connect(self.cmdSimsTable_clicked)

    self.setLayout(layout)
    self.ReturnValues = None
    # Return values container variable
    self.exec_()

def cmdOK_clicked(self):
    self.ReturnValues = popup_open_save(self.PopupArgs[0], self.PopupArgs[1],
     self.PopupArgs[2], self.PopupArgs[3])
    self.close()

def cmdCancel_clicked(self):
    self.ReturnValues = None
    self.close()

def cmdDataTable_clicked(self):
    popup_simple_dialog('Data Table', [[SummaryTableDS(self.TableData[1][1],
                                                   self.TableData[1][0],
                                                   [item[0] for item in
                                                    self.TableData[0[1]])]])

def cmdSimsTable_clicked(self):
    popup_simple_dialog('Simulations Table',
        [[create_simple_table(self.TableData[0][1], self.TableData[0][0])]])


class MyApp(QMainWindow, Ui_MainWindow):

    def __init__(self):

        def __load_parameters():
            wb = xl.load_workbook(ParametersConfig)
            ws = wb['Parameters']

            data_types = []
            tab_titles = []
            r = 2
            row_index = ws.cell(row=2, column=2).value

            while row_index is not None:
                data_type = ws.cell(row=r, column=9).value
                # Check data type
                if data_type is not None and data_type == 'Integer':
                    data_types.append(row_index)

                tab_title = ws.cell(row=r, column=1).value
                # Check tab limits
                if tab_title is not None:
                # Found tab title
                    tab_titles.append((tab_title, row_index))
                r += 1
                row_index = ws.cell(row=r, column=2).value

            tab_titles.append((None, r-2))
            tab_lengths = [(tab_titles[i+1][1]-tab_titles[i][1])
            for i in range(0, len(tab_titles)-1)]
            tabs_all = []

            for i in range(0, len(tab_titles)-1):
                tab_title = tab_titles[i][0]
                tab_start = tab_titles[i][1]
                tab_length = tab_lengths[i]

                tab_set = []
                for j in range(0, tab_length):
                    tab_set.append(get_xl_row(ws, (j+tab_start+2, 2), 7))
                tabs_all.append((tab_title, tab_set))

            Tables_Widgets = []         # Generate GUI widgets
            tab_widget = QTabWidget()

            start_row_label = 0
            for tab in tabs_all:
                rows = len(tab[1])
                columns = len(tab[1][0])-1
                table = QTableWidget(rows, columns)
                table.setHorizontalHeaderLabels(('Parameter', 'Unit', 'Min',
                 'Max', 'Step', 'Description'))
                table.setVerticalHeaderLabels((str(index) for index in
                 range(start_row_label, start_row_label+rows)))
                table.horizontalHeader().setStretchLastSection(True)
                for r in range(0, rows):
                    for c in range(0, columns):
                        table.setItem(r, c, QTableWidgetItem(str(tab[1][r][c+1])))

                tab_page = QWidget()
                tab_widget.addTab(tab_page, tab[0])

                tab_page_layout = QVBoxLayout()
                tab_page_layout.addWidget(table)
                tab_page.setLayout(tab_page_layout)

                Tables_Widgets.append((table, start_row_label,
                start_row_label+rows))
                start_row_label += rows

            return data_types, Tables_Widgets, tab_widget

        QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)

        integersList, par_t, Tab_widget = __load_parameters()
        # Initialize MP from config file
        self.DataIntegers = integersList
        # List model parameters with integer data type

        self.ParameterTableWidgets = par_t
        self.ParametersLayout.addWidget(Tab_widget)
        self.cmdGenerateData.clicked.connect(self.cmdGenerateData_clicked)
        self.cmdLoadDataset.clicked.connect(self.cmdLoadDataset_clicked)

    def cmdGenerateData_clicked(self):

        def __get_input_table_data():
            data_input_error_list = []
            mp_list = []
            for table_set in self.ParameterTableWidgets:
                tbl_widget = table_set[0]
                start = table_set[1]

                for j in range(start, table_set[2]):

                    # Read Table Data as text. All cells are editable
                    cell_name = tbl_widget.item(j - start, 0).text()

                    cell_min = tbl_widget.item(j - start, 2).text()
                    cell_max = tbl_widget.item(j - start, 3).text()
                    cell_step = tbl_widget.item(j - start, 4).text()

                    # Check if input is correct
                    founderrors = (cell_name == '')
                    # User must enter something as parameter name

                    if not (is_number(cell_min) and is_number(cell_max)
                    and is_number(cell_step)):  # Must be numbers
                        founderrors = True
                    else:
                        # Check for errors Min>Max or Step > Interval

                        mp_min = float(cell_min)                # Min
                        mp_max = float(cell_max)                # Max
                        mp_step = float(cell_step)              # Step

                        if (mp_min > mp_max) or (mp_min < mp_max
                        and mp_max - mp_min < mp_step):
                            founderrors = True

                    # Error check finished. Create Model_Parameter
                    if no errors have been found.
                    if not founderrors:
                        mp_list.append(ModelParameter(j,
                                                      cell_name + ' [' +
                                                       tbl_widget.item(j - start,
                                                        1).text() + ']',
                                                      float(cell_min),
                                                      float(cell_max),
                                                      float(cell_step)))
                    else:
                        data_input_error_list.append(j)

            # Inform user on errors
            if len(data_input_error_list) == 0:
                return mp_list
            else:
                popup_message('Please fix errors in rows: ' +
                 str(data_input_error_list)[1:-1] + ' and repeat!',
                              QMessageBox.Critical, 'Data Table Error',
                               QMessageBox.Ok)

        def __parse_to_mpl(data_table, captions):
            mp_list = []
            for mp_col in range(0, len(data_table[0])):
                p_min = np.min(data_table[:, mp_col])
                p_max = np.max(data_table[:, mp_col])
                caption = captions[mp_col][0] + ' [' + captions[mp_col][1] + ']'
                mp_list.append(ModelParameter(mp_col, caption,
                p_min, p_max, p_max - p_min))

            return mp_list
```

```
def __prepare_dataset_subfolder(fn, extension):
    proceed = True
    subfolder_path = fn[0:len(fn) - len(extension)]

    if not os.path.exists(subfolder_path):
    # Subfolder does not exist. Create without notification!
        os.makedirs(subfolder_path)

    elif len(os.listdir(subfolder_path)) > 0:
    # Subfolder exists with data. Prompt for data erase.

        confirm_erase = popup_message('Dataset Subdirectory contains
        files and folders! '
                                      'Do you want to erase ALL DATA data
                                       in Subdirectory?',
                                      QMessageBox.Warning, 'Confirm
                                       Delete',
                                      QMessageBox.Ok | QMessageBox.Cancel)

        if confirm_erase == QMessageBox.Ok:
        # Erase allowed by user
            shutil.rmtree(subfolder_path, ignore_errors=True)
            os.makedirs(subfolder_path)
        else:
            proceed = False
            # User chooses to keep subfolder. Do not proceed
    return proceed

basic_dialog = Input_Form_DatasetBasic().ReturnValues
# Get basic dataset input from Dialog Window

if basic_dialog is not None:
    ds_type, blueprint, sim_fn_prefix, ds_desc = basic_dialog
    # Basic dataset data
    no_input_errors = True
    # Error catcher
    ds_creator = None

    if ds_type != 'Custom':
        mpr_list = __get_input_table_data()
        # Get input from table widget, Handle errors

        if mpr_list is not None:
        # No input errors found in datatable

            ds_creator = Dataset(ModelParameterList(mpr_list), ds_type,
             ds_desc, blueprint)

            if ds_type == 'Simple':
                ds_creator.assign_datatable(None, sim_fn_prefix)
                # No additional input needed. assign table

            elif ds_type == 'Metamodel':
                # Additional input needed. prompt for input
                    mm_dialog = Input_Form_MM().ReturnValues

                    if mm_dialog is not None:
                    # Input values ok. assign table.

                        if not ds_creator.assign_datatable((mm_dialog,
                         self.DataIntegers), sim_fn_prefix):
                            no_input_errors = False
                            popup_message('At least one input parameter
                            should have range!', QMessageBox.Critical,
                                          'Input Error', QMessageBox.Ok)
                    else:
                        no_input_errors = False
            else:
                no_input_errors = False
        else:
            custom_dialog = Input_Form_CustomDS().ReturnValues

            if custom_dialog is not None:
                table, titles = custom_dialog
                ds_creator = Dataset(ModelParameterList(__parse_to_mpl(table,
                 titles)), ds_type, ds_desc, blueprint)
                ds_creator.assign_datatable(table, sim_fn_prefix)

            else:
                no_input_errors = False
                # Input canceled. Program stops

        # dataset with simulations is prepared at this point. Display summary.
         Prompt user for final confirmation.
        if no_input_errors:
            choice1 = Summary_Form_Dataset(ds_creator.SummaryTable,
             ('Summary', None)).ReturnValues

            if choice1 is not None and choice1 != '' and
             __prepare_dataset_subfolder(choice1, DSExtension):
                ds_creator.save_first(choice1)
                # Save dataset for the first time
                popup_message('Dataset saved', QMessageBox.Information,
                 'Information', QMessageBox.Ok)

@staticmethod
def cmdLoadDataset_clicked():

    ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
     'Dataset files (*.dat)')

    if ds_filename != '':

        try:
            ds_loaded = load_object(ds_filename)
```

```
                choice2 = Summary_Form_Dataset(ds_loaded.SummaryTable,
                ('Loaded', ds_filename)).ReturnValues
                if choice2 is not None and choice2 != '':

                    ds_loaded.export_SummaryTable_1(choice2)

                    popup_message('Excel file successfuly created.',
                     QMessageBox.Information, 'File Export',
                                  QMessageBox.Ok)

            except ImportError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened.
                Close it and repeat.',
                              QMessageBox.Critical, 'File Error', QMessageBox.Ok)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MyApp()
    window.show()
    sys.exit(app.exec_())
```

Module 2 - Simulation Runner `M2.py`

```
import sys
import zipfile
import math
import multiprocessing
from multiprocessing import Process
import time
import datetime
import openpyxl as xl

from PyQt5.QtCore import Qt, pyqtSlot, QRunnable, QThreadPool, pyqtSignal, QObject
from PyQt5.QtWidgets import QMainWindow, QApplication, QMdiArea, QMdiSubWindow,
 QMessageBox, QLabel, QGridLayout
from PyQt5.QtWidgets import QWidget, QPushButton, QLineEdit, QProgressBar,
 QDialog, QComboBox

from Main_Functions import popup_open_save, popup_message, get_xl_row,
 popup_yes_no_dialog, popup_open_multiple

from Main_ConfigConstants import DefaultSimDir, Machines_List, DefaultServerName,
 PreProcessing, PostProcessing

from Main_ConfigConstants import SimExtension, MaxComputers
from M2_Classes import PLAXIS_Machine

from M2_WidgetClasses import DataTable, Console, ActiveTable_2

from Main_WidgetClasses import load_dataset_sim_table

from Main_WidgetClasses import create_simple_table, StringCell

import M2_Solvers


class DialerSignals(QObject):

    answer = pyqtSignal(str, str, str)


class SimRunnerSignals(QObject):

    started = pyqtSignal(int, object)
    # Returns simulation ID, machine ID

    ended = pyqtSignal(int, object, bool, str)
    # Returns simulation ID, machine ID and result (T/F)


class Dialer(QRunnable):

    def __init__(self, machine, io, timeout):
        super(Dialer, self).__init__()

        self.__Machine = machine
        self.__IO = io
        self.__Timeout = timeout

        self.DSignals = DialerSignals()

    @pyqtSlot()
    def run(self):

        # Run connection check on local/remote machine
        if self.__IO == 'Input':
            function_run = self.__Machine.check_input_connection
        else:
            function_run = self.__Machine.check_output_connection

        # Try to join Pool after timeout period. If False, return timeout event.
        Return if ok. Return on error.

        try:
            p = multiprocessing.Process(target=function_run)
            p.start()
            p.join(self.__Timeout)
            if p.is_alive():
                p.terminate()
                p.join()
                self.DSignals.answer.emit(self.__Machine.MachineID,
                self.__IO, 'Timeout')        # connection timeout

            else:
                self.DSignals.answer.emit(self.__Machine.MachineID,
                self.__IO, 'Connected')      # connected

        except:
            self.DSignals.answer.emit(self.__Machine.MachineID, self.__IO,
             'Failed')             # connection failed


class SimRunner(QRunnable):

    def __init__(self, simulation_id, machine, dataset, analysis_type,
    sim_path, script_to_run):

        super(SimRunner, self).__init__()

        self.__SimID = simulation_id
        self.__PC = machine
        self.__DS = dataset
        self.__Job = analysis_type
        self.__SimPath = sim_path
        self.__Script = script_to_run

        self.RSignals = SimRunnerSignals()

    @pyqtSlot()
    def run(self):

        self.RSignals.started.emit(self.__SimID, self.__PC)
        # Emit START signal to main program

        # Initialize simulation start and write record
        if self.__PC.NetworkPosition == 'Local':
            i1, i2, i3 = self.__DS.start_local(self.__SimID,
            self.__PC.MachineID, self.__SimPath, self.__Job)
        else:
            i1, i2, i3 = self.__DS.start_remote(self.__SimID, self.__PC.MachineID,
             self.__Job, self.__PC.SimsPath)

        # Run simulation Pre/Post processing Script
        if self.__Job == 'Calculation':
            results = self.__Script(i1, i2, i3, self.__PC.HostName,
             self.__PC.PortInput)
            # Run Calc Script
        else:
            results = self.__Script(self.__SimPath, self.__PC.HostName,
             self.__PC.PortOutput)
            # Run PostScript

        # End simulation and write record
        if self.__PC.NetworkPosition == 'Local':
            self.__DS.end_local(self.__SimID, self.__PC.MachineID, results,
             self.__SimPath,
             self.__Job)
        else:
            self.__DS.end_remote(self.__SimID, self.__PC.MachineID, results,
             self.__PC.SimsPath, self.__Job)

        # Emit appropriate ENDED signal to main program
        if self.__Job == 'Calculation':
            self.RSignals.ended.emit(self.__SimID, self.__PC, results[0], i2)

        else:
            self.RSignals.ended.emit(self.__SimID, self.__PC, results[0],
             self.__SimPath)


class Data_Explorer(QDialog):

    def __init__(self, table_data):
        super(Data_Explorer, self).__init__()

        captions, data_table, result_tables = table_data
        # Unpack input tuple

        self.setFixedSize(1000, 500)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Data Explorer')

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdExport = QPushButton('Export All Results')
        self.cmdClose = QPushButton('Close')
        self.tblDataTable = DataTable(data_table, captions, result_tables)

        self.cmdExport.clicked.connect(self.cmdExport_clicked)
        self.cmdClose.clicked.connect(self.cmdClose_clicked)
        self.tblDataTable.table_added.connect(self.Data_Added)

        layout.addWidget(self.tblDataTable, 0, 0)
        layout.addWidget(self.cmdExport, 1, 0)
        layout.addWidget(self.cmdClose, 2, 0)

        self.Changes = []

        self.exec_()

    def cmdExport_clicked(self):
        if self.tblDataTable.can_export():
        # We have some results and we export them to excel or numpy
            if self.tblDataTable.GlobalShape is not None:
            # All data tables have the same shape
                if len(self.tblDataTable.GlobalShape) > 3:
                # Export 4D+ tables to numpy
                        self.tblDataTable.export_numpy_tables()
                else:
                # Export 1D/2D/3D tables to Excel
                    self.tblDataTable.export_excel_consistent()
            else:
                self.tblDataTable.export_all()

    @pyqtSlot(int, object)
    def Data_Added(self, sim_index, res_table):
        self.Changes.append((sim_index, res_table))

    def cmdClose_clicked(self):
        self.close()


class Calculation_Window(QDialog):

    def __init__(self, computers, loaded_ds, sims_list, calc_or_proc,
     pc_controller, calc_paths, script):

        super(Calculation_Window, self).__init__()

        self.OnlineComputers = computers
        self.Dataset = loaded_ds
        self.Job = calc_or_proc
        self.MachinesController = pc_controller
        self.CalculatedPaths = calc_paths
        self.RunnerScript = script
        self.SimulationIDList = sims_list

        self.Sims_Pool = sims_list[:]

        self.Calculated = 0
        self.Remaining = len(sims_list)
        self.TotalSims = len(sims_list)

        self.Timers = []
        self.ReturnValue = None

        self.RunThreads = QThreadPool()
        self.RunThreads.setMaxThreadCount(MaxComputers)

        # Setting up GUI ########################################################
        layout = QGridLayout()
        self.setLayout(layout)

        self.setFixedSize(1000, 500)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Job Runner')

        data_table = self.Dataset.SummaryTable_2_1(sims_list)[0]
        captions = self.Dataset.SummaryTable_2_1(sims_list)[1]

        self.tblTable = create_simple_table(data_table, captions)

        self.cnConsole1 = Console(app)

        self.pbProgress1 = QProgressBar()
        self.pbProgress1.setValue(0)

        self.lblTimer = QLabel('Remaining Time: ***')

        self.lblRemainingSims = QLabel('Remaining Simulations: ' +
         str(len(sims_list)) + '/' + str(len(sims_list)))

        self.cmdStart = QPushButton('START JOB')
        self.cmdClose = QPushButton('CLOSE WINDOW')

        self.cmdStart.clicked.connect(self.cmdStart_clicked)
        self.cmdClose.clicked.connect(self.cmdClose_clicked)

        # Add widgets to layout
        layout.addWidget(self.tblTable, 0, 0)
        layout.addWidget(self.cnConsole1, 1, 0)
        layout.addWidget(self.pbProgress1, 2, 0)
        layout.addWidget(self.lblRemainingSims, 3, 0)
        layout.addWidget(self.lblTimer, 4, 0)
        layout.addWidget(self.cmdStart, 5, 0)
        layout.addWidget(self.cmdClose, 6, 0)

        self.__widgets_init()

        self.exec_()

    def __start_worker(self, sim, machine, path):

        self.Sims_Pool.remove(sim)
        # Simulation is Started. Remove it from the list

        worker = SimRunner(sim, machine, self.Dataset, self.Job, path,
        self.RunnerScript)

        worker.RSignals.started.connect(self.Sim_Started)
        worker.RSignals.ended.connect(self.Sim_Ended)
        self.RunThreads.start(worker)

    def cmdStart_clicked(self):

        if popup_yes_no_dialog('Job Cannot be stopped! Proceed?',
         QMessageBox.Question, 'Confirm Start') == QMessageBox.Ok:
```

```
        self.Timers.append(time.time())
        self.__widgets_job_start()
        # Initialize Observer GUI Widgets

        # Start first job. Then wait for simulation to finish and assign new
        simulation to freed computer

        first_job_count = min(len(self.Sims_Pool), len(self.OnlineComputers))
        # Min of Sims/Machines num

        first_machines = [self.OnlineComputers[i] for i in range(0,
         first_job_count)]

        if self.Job == 'Processing':
            first_sims = [self.MachinesController[i][1][0] for i in range(0,
             len(first_machines))]

            first_paths = [self.CalculatedPaths[self.__sim_row(sim_index)]
             for sim_index in first_sims]

        else:
            first_sims = [self.Sims_Pool[i] for i in range(0,
             first_job_count)]

            first_paths = [None]*first_job_count

        for i, machine in enumerate(first_machines):
        # Assign simulation to each machine - Initial Job

            self.__start_worker(first_sims[i], machine, first_paths[i])

def cmdClose_clicked(self):

    if self.Calculated == 0:
        self.ReturnValue = 'Nothing Happened'

    else:
        self.ReturnValue = 'Closed'

    self.close()

@pyqtSlot(int, str)
def Sim_Started(self, sim_id, machine):
    self.__widgets_sim_started(sim_id, machine.MachineID)

@pyqtSlot(int, str, bool, str)
def Sim_Ended(self, sim_id, machine, success, path):

    self.Remaining -= 1
    self.Calculated += 1

    self.Timers.append(time.time())
    self.__widgets_sim_ended(sim_id, machine, success, path)
    # Update widgets after simulation is Ended

    # Create new job if there are simulations in the queue

    if len(self.Sims_Pool) > 0:
    # There are non-calculated sims. Find next sim to run on freed machine

        sim_to_run = self.Sims_Pool[0]
        run_next, next_path = (True, None)
        # Set for Calculation Case.

        if self.Job == 'Processing':

            # Determine next simulation for postprocessing using Controller,
            which determines next sim on freed PC!
            next_sim = None

            for machine_set in self.MachinesController:

                if machine.MachineID == machine_set[0]:
                # Found set of simulations on freed PC

                    # Search for next simulation among freed PC's simulations
                    sims_on_pc = machine_set[1]

                    for j, sim in enumerate(sims_on_pc):
                    # Search for next simulation

                        if sim_id == sim and j + 1 < len(sims_on_pc):
                        # Simulation Found. Check if next exists

                            next_sim = sims_on_pc[j + 1]

            if next_sim is not None:
                sim_to_run = next_sim
                next_path = self.CalculatedPaths[self.__sim_row(sim_to_run)]
            else:
                run_next = False

        if run_next:
            self.__start_worker(sim_to_run, machine, next_path)
            # Start new sim in queue

    else:
        # At this point, there are no NEW sims to be started. However,
        we must check if ALL simulations

        # are finished, and then call <<__widgets_job_ended>> method

        if self.Remaining == 0:
        # It will be ZERO when the LAST simulation is ENDED
```

```
            self.__widgets_job_ended()  # Reset Widgets on job End

# WIDGET UPDATE METHODS ##################################################

def __sim_row(self, sim_id):
    return self.SimulationIDList.index(sim_id)

def __t_rem(self):

    t_average_per_sim = (self.Timers[-1] - self.Timers[0]) / self.Calculated

    return int(t_average_per_sim * self.Remaining)

def __table_row_fill(self, row, data, cstart):

    for c in range(cstart, cstart+len(data)):
        self.tblTable.setItem(row, c, StringCell(data[c-cstart]))

    self.tblTable.resizeColumnsToContents()

def __table_sim_ended(self, row, success, pc_id, path):

    if success:
        action_table = 'Successfully Finished'
    else:
        action_table = 'Not Successfully Finished'

    self.__table_row_fill(row, [action_table, pc_id, path], 1)

def __widgets_sim_started(self, sim_id, pc_id):
    self.__table_row_fill(self.__sim_row(sim_id),
    ['In Progress...', pc_id], 1)  # Update simulation table

    self.cnConsole1.twrite('*** Simulation ' + str(sim_id) + ' started on
     Computer ' + str(pc_id) + ' ***')

def __widgets_init(self):
    self.cnConsole1.twrite('*** Job Initialized ***')
    self.cnConsole1.write('')
    self.cnConsole1.write('Number of Online Computers: ' +
     str(len(self.OnlineComputers)))

    self.cnConsole1.write('Number of Simulations: ' + str(self.TotalSims))
    self.cnConsole1.write('')

def __widgets_job_start(self):

    self.cnConsole1.twrite('*** Job Started ***')
    # Reset progress Widgets Data

    self.cnConsole1.write('')

    self.cmdStart.setEnabled(False)
```

```
        self.cmdClose.setEnabled(False)

    def __widgets_job_ended(self):

        self.lblTimer.setText('Remaining Time: ***')

        self.cnConsole1.write('')
        self.cnConsole1.twrite('*** Job Ended ***')

        self.cmdClose.setEnabled(True)

    def __widgets_sim_ended(self, sim_id, machine, success, path):

        self.__table_sim_ended(self.__sim_row(sim_id), success, machine.MachineID,
         path)

        self.lblRemainingSims.setText('Remaining Simulations: ' +
         str(self.Remaining) + '/' + str(self.TotalSims))

        self.lblTimer.setText('Remaining Time: ' +
         str(datetime.timedelta(seconds=self.__t_rem())))

        progress = math.ceil((self.Calculated / self.TotalSims) * 100)
        if progress > 100:
            progress = 100

        self.pbProgress1.setValue(progress)

        self.cnConsole1.twrite('*** Simulation ' + str(sim_id) + ' ended on
         Computer ' + str(machine.MachineID) + ' ***')


class MainWindow(QMainWindow):

    def __init__(self, parent=None):

        super(MainWindow, self).__init__(parent)

        self.LOADED_DATASET = None
        self.SimsTable = None
        self.DialerPool = QThreadPool()

        self.Worker_1 = None
        self.Worker_2 = None

        self.setWindowTitle('SIMULATION RUNNER')
        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)         # Sub Windowed GUI

        # Commands Window
        top_widget_1 = QWidget()
        layout1 = QGridLayout(top_widget_1)
        top_widget_1.setLayout(layout1)
```

```
self.commands_window = QMdiSubWindow()
self.mdi.addSubWindow(self.commands_window)

self.commands_window.setWindowTitle('Dataset Not Loaded')
self.commands_window.setFixedSize(400, 300)
self.commands_window.setWindowFlags(Qt.CustomizeWindowHint)
self.commands_window.setWidget(top_widget_1)

cmdOpenDataset = QPushButton('LOAD DATASET')
self.cmdInspectResults = QPushButton('VIEW RESULTS')
self.cmdRunAll = QPushButton('RUN CALCULATIONS')
self.cmdOutput = QPushButton('RUN POST PROCESSING')

self.cmdRunAll.setEnabled(False)
self.cmdOutput.setEnabled(False)
self.cmdInspectResults.setEnabled(False)

cmdOpenDataset.clicked.connect(self.cmdOpenDataset_clicked)
self.cmdRunAll.clicked.connect(self.cmdRunAll_clicked)
self.cmdOutput.clicked.connect(self.cmdOutput_clicked)
self.cmdInspectResults.clicked.connect(self.cmdInspectResults_clicked)

# Create List of Pre/Post processing functions
self.cmbPreProcess = QComboBox()
self.cmbPreProcess.addItems([item[1] for item in PreProcessing])
self.cmbPreProcess.setCurrentIndex(0)

self.cmbPostProcess = QComboBox()
self.cmbPostProcess.addItems([item[1] for item in PostProcessing])
self.cmbPostProcess.setCurrentIndex(0)

layout1.addWidget(QLabel('Dataset Type'), 0, 0)
layout1.addWidget(QLabel('Dataset Description'), 1, 0)
layout1.addWidget(QLabel('Modelling Sequence'), 2, 0)
layout1.addWidget(cmdOpenDataset, 3, 0)
layout1.addWidget(self.cmdInspectResults, 3, 1)
layout1.addWidget(self.cmdRunAll, 4, 0)
layout1.addWidget(self.cmdOutput, 4, 1)

layout1.addWidget(QLabel('PreProcessing Script: '), 5, 0)
layout1.addWidget(self.cmbPreProcess, 5, 1)
layout1.addWidget(QLabel('PostProcessing Script: '), 6, 0)
layout1.addWidget(self.cmbPostProcess, 6, 1)

self.Labels = []
for j in range(0, 3):
# Fill label values
    label = QLabel('***DATASET NOT LOADED***')
    self.Labels.append(label)
    layout1.addWidget(label, j, 1)

# Simulations Table Window
```

```
        top_widget_2 = QWidget()
        self.Layout2 = QGridLayout(top_widget_2)
        top_widget_2.setLayout(self.Layout2)

        sim_window = QMdiSubWindow()
        self.mdi.addSubWindow(sim_window)

        sim_window.setWindowTitle('Simulations')
        sim_window.setFixedSize(1000, 600)
        sim_window.setWindowFlags(Qt.CustomizeWindowHint)
        sim_window.setWidget(top_widget_2)

        # Machines List Window
        self.LOADED_MACHINES = []

        top_widget_3 = QWidget()
        self.Layout3 = QGridLayout(top_widget_3)
        top_widget_3.setLayout(self.Layout3)

        machines_window = QMdiSubWindow()
        self.mdi.addSubWindow(machines_window)

        machines_window.setWindowTitle('Computers')
        machines_window.setFixedSize(600, 600)
        machines_window.setWindowFlags(Qt.CustomizeWindowHint)
        machines_window.setWidget(top_widget_3)

        self.cnConsole = Console(app)
        # Initialize custom Console (inherits QTextEdit)

        self.cnConsole.write('*** Computers Loaded ***')
        self.txtConnectionTimeout = QLineEdit('3')
        self.pbConnect = QProgressBar()
        self.__initialize_machines()

        # Show sub windows
        self.commands_window.show()
        sim_window.show()
        machines_window.show()

    def __return_online(self, io):

        # This is block code function that checks if there are ANY machine
        and simulation selected in the list.

        # Then it checks if selected computers are online for the
         desired purpose (calculation or postprocessing).

        # Returns state and online computers - True + ComputersList / False
        + empty list

        computers_online = []
        proceed = False
        not_executed = None
        executed = None

        if len(self.MachinesTable.SelectedRows) == 0 or
         len(self.SimsTable.SelectedRows)
        == 0:
            popup_message('Select at least one Computer and Simulation to
             proceed!',
                          QMessageBox.Critical, 'Error', QMessageBox.Ok)

        else:
            computers_online = [self.LOADED_MACHINES[i]
                                for i in self.MachinesTable.SelectedRows if
                                 self.LOADED_MACHINES[i].online(io)]

            if len(computers_online) == 0:
                popup_message('All Selected Computers are Offline!',
                 QMessageBox.Critical, 'Error', QMessageBox.Ok)

            else:
                not_executed, executed =
                 self.LOADED_DATASET.filter(self.SimsTable.SelectedRows, io)

                proceed = True

        return computers_online, not_executed, executed, proceed

    def __link_all(self):
        self.cmdLinkAll = QPushButton('Link Multiple Calculated Simulations')
        self.cmdLinkAll.clicked.connect(self.cmdLinkAll_clicked)
        self.Layout2.addWidget(self.cmdLinkAll, 1, 0)

    def __reload_dataset(self):
        reload_path = self.LOADED_DATASET.CurrentPath
        del self.LOADED_DATASET                             # DS Unloaded

        self.LOADED_DATASET, self.SimsTable = load_dataset_sim_table(self.Layout2,
         reload_path)
        self.__link_all()

    def cmdLinkAll_clicked(self):

        to_link = len(self.SimsTable.SelectedRows)

        if to_link > 0:     # At least one simulation is selected for linking

            link_filenames = popup_open_multiple(DefaultSimDir,
            'Select Simulations', 'Simulations (*' + SimExtension + ')')

            selected = len(link_filenames)

            if selected > 0:
                if to_link != selected:
                    popup_message('Number of selected simulations and number of
                     files must match!', QMessageBox.Warning, 'Error',
                      QMessageBox.Ok)

                else:
                    self.SimsTable.link_multiple(link_filenames)

        else:
            popup_message('Select at least one simulation and repeat.',
             QMessageBox.Warning, 'Error', QMessageBox.Ok)

    def __initialize_machines(self):

        def __load_plaxis_machines_xl():

            Parsed_Data = []

            try:
                wb = xl.load_workbook(Machines_List)
                ws = wb['MACHINES']
                r1 = 2

                check = ws.cell(row=r1, column=1).value
                while check is not None:
                    Parsed_Data.append(get_xl_row(ws, (r1, 1), 7))
                    r1 += 1
                    check = ws.cell(row=r1, column=2).value

            except FileNotFoundError:
                popup_message('Computers File Not Found', QMessageBox.Critical,
                'File Error', QMessageBox.Ok)

            except zipfile.BadZipFile:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            return Parsed_Data

        def __create_machine_instances():

            loaded_machines = []

            for machine_data_xl in __load_plaxis_machines_xl():

                machine_name = machine_data_xl[6]
                sim_path = machine_data_xl[5]
                ip = machine_data_xl[0]
                position = 'Remote'

                if machine_name == DefaultServerName:
                # Override excel data if computer is SERVER by default name
```

```
                ip = 'localhost'
                sim_path = DefaultSimDir
                position = 'Local'

            loaded_machines.append(PLAXIS_Machine(ip, machine_name, position,
             sim_path,
                                                  machine_data_xl[3],
                                                  machine_data_xl[4],
                                                  machine_data_xl[1],
                                                  machine_data_xl[2]))
        return loaded_machines

    self.LOADED_MACHINES = __create_machine_instances()

    if len(self.LOADED_MACHINES) == 0:
        popup_message('No Computers!', QMessageBox.Information, 'Information',
         QMessageBox.Ok)

    else:
        self.MachinesTable = ActiveTable_2([item.SummaryData for item in
         self.LOADED_MACHINES],
                                           ['Computer Name', 'IP',
                                            'PreProcessor Online',
                                            'PostProcessor Online'],
                                           2, ['Check Connection', 'Check
                                            Connection'], 2)

        self.cmdCheckAll = QPushButton('CHECK ALL CONNECTIONS')

        self.MachinesTable.cell_button_clicked.connect
        (self.check_single_connect)

        self.cmdCheckAll.clicked.connect(self.cmdCheckAllConnections_clicked)

        self.Layout3.addWidget(self.MachinesTable, 0, 0)
        self.Layout3.addWidget(self.cnConsole, 1, 0)
        self.Layout3.addWidget(QLabel('Connection Timeout [s]: '), 2, 0)
        self.Layout3.addWidget(self.txtConnectionTimeout, 3, 0)
        self.Layout3.addWidget(self.cmdCheckAll, 4, 0)
        self.Layout3.addWidget(self.pbConnect, 5, 0)

def __prepare_final_queue(self, final_queue, comp_online):

    sim_indices = []

    for item in final_queue:
        for index in item[1]:
            sim_indices.append(index)

    sim_indices.sort()

    job_machines = []
    for item in final_queue:
        for m in comp_online:
            if m.MachineID == item[0]:
                job_machines.append(m)

    return job_machines, sim_indices, self.SimsTable.get_paths(sim_indices)

@pyqtSlot(int, int)
def check_single_connect(self, i, j):

    if j == 2:
        i_o = 'Input'
        pc_str = 'PreProcessor'
    else:
        i_o = 'Output'
        pc_str = 'PostProcessor'

    M = self.LOADED_MACHINES[i]
    self.cnConsole.write('*** Connecting with ' + pc_str + ' Computer ' +
    M.MachineID + ' ...')

    worker = Dialer(M, i_o, float(self.txtConnectionTimeout.text()))
    worker.DSignals.answer.connect(self.connection_answer)

    self.DialerPool.start(worker)

@pyqtSlot(str, str, str)
def connection_answer(self, machine_id, i_o, answer):

    # Search for machine using ***UNIQUE*** machineID
    row = 0
    M = self.LOADED_MACHINES[0]
    for row, M in enumerate(self.LOADED_MACHINES):
        if M.MachineID == machine_id:
            break                       # Machine found

    if i_o == 'Input':
        pc_string = 'PreProcessor'
        column = 2
    else:
        pc_string = 'PostProcessor'
        column = 3

    if answer == 'Connected':
        update_status = True
        say = '*** ' + pc_string + ' Computer ' + M.MachineID + ' Connected!'

    elif answer == 'Timeout':
        update_status = False
        say = '*** Connection Timeout on Computer ' + M.MachineID + ' ***'

    else:
        update_status = False
        say = '*** Connection with ' + pc_string + ' Computer ' +
        M.MachineID + ' Failed!'

    M.ConnectionStatus[column-2] = update_status

    self.cnConsole.write(say)
    self.MachinesTable.reset_cell(update_status, row, column)

def update_progress_bar(self):

    new_value = self.pbConnect.value() + math.ceil(50
    /len(self.LOADED_MACHINES))

    if new_value > 100:
        new_value = 100

    self.pbConnect.setValue(new_value)

def cmdRunAll_clicked(self):

    script = getattr(M2_Solvers,
     PreProcessing[self.cmbPreProcess.currentIndex()][0])

    pc_online, not_calculated, calculated, proceed =
     self.__return_online('Input')

    if proceed:

        # Check if calculated (including linked) simulations are selected for
        re-calculation

        # This is additional conditional question, it doesn't happen in all
         cases. Ask user to repeat

        proceed1 = True
        if len(calculated) + len(self.SimsTable.SelectedLinked) > 0:

            if popup_yes_no_dialog('Some selected Simulations already have
             results. 'Previous results might be overwritten! Calculate
              Again?',
                                   QMessageBox.Information, 'Confirm
                                    Continue') ==
                                    QMessageBox.Cancel:

                proceed1 = False
                # User chooses not to proceed with repeated execution

        if proceed1:
        # Start Calculation Window. If properly closed, update dataset

            sim_list = list(set(not_calculated + calculated))
            sim_list.sort()

            if Calculation_Window(pc_online, self.LOADED_DATASET, sim_list,
            'Calculation', None, None,
                                script).ReturnValue == 'Closed':

                self.LOADED_DATASET.update_records()
                # Dataset is saved again after this line. Reload.

                self.__reload_dataset()

def cmdOutput_clicked(self):

    Computers_Online, not_processed, processed,
    proceed = self.__return_online('Output')

    script = getattr(M2_Solvers,
     PostProcessing[self.cmbPostProcess.currentIndex()][0])

    if proceed:
    # We have computers online. Check further.

        # Check if some results already exists and Ask user to repeat
         processing.

        proceed1 = True
        if len(processed) > 0:
            if popup_yes_no_dialog('Some selected Simulations are already
             processed.
             Process Again?',
                                  QMessageBox.Information, 'Confirm
                                   Continue') == QMessageBox.Cancel:

                proceed1 = False

        if proceed1:
            # Here we must check if some simulations selected for processing
             are not calculated. From that list we exclude the simulations
              that are linked externally

            if self.SimsTable.selected_calculated() is False:
                popup_message('Some selected simulations are not calculated
                or externally linked!', QMessageBox.Critical, 'Error',
                 QMessageBox.Ok)

            else:
                # Check if needed computers are available. Notify if so.
                Recreate Queue for next step

                queue = []
                online_computers = [m.MachineID for m in Computers_Online]
                # Just names of Online Computers
                proceed2 = True
                notice2 = False

                for j1, needed_computer in
```

```
 enumerate(self.SimsTable.NeededComputers):

    if needed_computer in online_computers:
        queue.append(self.SimsTable.ProcessingQueue[j1])

    else:
        notice2 = True
        # Needed computer not online. Notify user in next
         block

# Final queue created. If some needed computers are not
 online, we know that through *notice2*

if notice2 is True and popup_yes_no_dialog('Needed computers
 not Online. Proceed?', QMessageBox.Warning, 'Confirm
  Continue') == QMessageBox.Cancel:

    proceed2 = False    # Further execution stopped

if proceed2 is True:

    if len(queue) == 0:   # Program will not continue if we
     don't have needed computers online

        popup_message('Needed Computers are not available!',
         QMessageBox.Critical,
                      'Computers Offline', QMessageBox.Ok)

    else:

        # Finally, we check if all paths in the table are Ok,
         and notify user to link missing sim!

        if len(self.SimsTable.SelectedInvalidPaths) > 0:
            popup_message('Some Selected Simulations are
             Missing! Link them first!',7

                          QMessageBox.Critical, 'Computers
                           Offline',
                           QMessageBox.Ok)

        else:

            # Now everything is Ok. We have Final Calculation
             queue and we start Processing

            job_pc, sim_indices, sim_paths =
            self.__prepare_final_queue(queue,
             Computers_Online)

            outcome = Calculation_Window(job_pc,
             self.LOADED_DATASET, sim_indices, 'Processing',
                                        queue, sim_paths,
                                                     script)

                            if outcome.ReturnValue == 'Closed':
                                self.LOADED_DATASET.update_records()
                                # DS is saved again after this line. Reload.

                                self.__reload_dataset()

def cmdOpenDataset_clicked(self):

    ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
    'Dataset files (*.dat)')

    if ds_filename != '':
        try:
            self.commands_window.setWindowTitle(ds_filename)

            self.LOADED_DATASET, self.SimsTable =
             load_dataset_sim_table(self.Layout2, ds_filename)

            self.LOADED_DATASET.update_records()

            self.__link_all()

            self.cmdInspectResults.setEnabled(True)

            if len(self.LOADED_MACHINES) > 0:
                self.cmdRunAll.setEnabled(True)
                self.cmdOutput.setEnabled(True)

            # Fill data in Window 1
            for lbl, value in zip(self.Labels,
             self.LOADED_DATASET.SummaryTable_2[0]):

                lbl.setText(value)

        except ImportError:
            popup_message('Invalid File', QMessageBox.Critical, 'File Error',
             QMessageBox.Ok)

        except PermissionError:
            popup_message('Permission Error. File already opened. Close it
            and repeat.',

             QMessageBox.Critical, 'File Error', QMessageBox.Ok)

        except AttributeError:
            popup_message('Invalid File', QMessageBox.Critical, 'File Error',
             QMessageBox.Ok)

def cmdCheckAllConnections_clicked(self):

    if self.DialerPool.activeThreadCount() == 0:
        # If searching is not in progress, check all connections

            # Update Widgets
            self.cnConsole.write('')
            self.cnConsole.write('*** Checking all Connections ***')
            self.MachinesTable.disable_buttons()
            self.pbConnect.setValue(0)

            to = float(self.txtConnectionTimeout.text())

            for M in self.LOADED_MACHINES:

                for i_o in ['Input', 'Output']:

                    worker = Dialer(M, i_o, to)
                    # Create workers for all machines/cases

                    worker.DSignals.answer.connect(self.connection_answer)
                    # Two functions added to the same event

                    worker.DSignals.answer.connect(self.update_progress_bar)
                    # Update progress bar

                    self.DialerPool.start(worker)

    def cmdInspectResults_clicked(self):
        changes = Data_Explorer(self.LOADED_DATASET.SummaryTable_2_2).Changes

        if len(changes) > 0:

            if popup_yes_no_dialog('Save changes to Dataset?',
             QMessageBox.Question,
                                   'Confirm changes') == QMessageBox.Ok:
                                   # Discard changes and close

                for changed_set in changes:
                    self.LOADED_DATASET.change_results(changed_set)

                self.__reload_dataset()


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Main Classes `M2_Classes.py`

```
import imp
import subprocess
```

```
from Main_ConfigConstants import PLAXIS_Path_MAIN_SERVER_PC

# PLAXIS Boilerplate Code #
found_module = imp.find_module('plxscripting', [PLAXIS_Path_MAIN_SERVER_PC])
plxscripting = imp.load_module('plxscripting', *found_module)
from plxscripting.easy import *


class Machine:

    def __init__(self, ip, name, position, sims_path, i_path, o_path):

        self.MachineID = name
        # Initialize Basic Computer+Software machine Data

        self.HostName = ip

        self.NetworkPosition = position
        self.SimsPath = sims_path
        self.I_Path = i_path
        self.O_Path = o_path

    def __eq__(self, other):
        return self.MachineID == other.MachineID


class PLAXIS_Machine(Machine):

    def __init__(self, ip, name, position, sims_path, i_path, o_path, i_port, o_port):
        super().__init__(ip, name, position, sims_path, i_path, o_path)

        self.PortInput = i_port
        # Input/Output ports for Plaxis Remote scripting

        self.PortOutput = o_port
        self.ConnectionStatus = [False, False]
        # Preprocessor/Postprocessor are Offline

    @property
    def SummaryData(self):
        return [self.MachineID, self.HostName, self.ConnectionStatus[0],
         self.ConnectionStatus[1]]

    def check_input_connection(self):
        # If computer is remote, then we cannot run subprocess commands at it.
         Instead, user should manually start batch

        # file on the remote PC upon turning on. Checking of input and output
         connection at remote pc is done just using

        # the simple new_server(host, port) commands
        if self.NetworkPosition == 'Local':
        # Open PLAXIS using Batch commands. Only possible for Local PC

            subprocess.Popen(self.I_Path + ' --AppServerPort=' + str(self.PortInput))

        S, G = new_server(self.HostName, self.PortInput)

    def check_output_connection(self):
        if self.NetworkPosition == 'Local':
        # Open PLAXIS using Batch commands. Only possible for Local PC
            subprocess.Popen(self.O_Path + ' --AppServerPort=' + str(self.PortOutput))

        S, G = new_server(self.HostName, self.PortOutput)

    def online(self, io):
        if io == 'Input':
            return self.ConnectionStatus[0]
        elif io == 'Output':
            return self.ConnectionStatus[1]
```

PLAXIS Pre-/Postprocessing Scripts `M2_Solvers.py`

```
import imp
import time
import random
import numpy as np

from Main_ConfigConstants import PLAXIS_Path_MAIN_SERVER_PC
from M2_Solvers_PLAXIS_Functions import *

found_module = imp.find_module('plxscripting', [PLAXIS_Path_MAIN_SERVER_PC])
plxscripting = imp.load_module('plxscripting', *found_module)
from plxscripting.easy import *

# Pre/Postprocessing Scripts ########################################################


class Point:

    def __init__(self, x, y, z):

        self.X = x
        self.Y = y
        self.Z = z

    def move(self, direction_vector):

        self.X += direction_vector[0]
        self.Y += direction_vector[1]
        self.Z += direction_vector[2]
    def distance(self, pt2):

        dx = self.X - pt2.X
        dy = self.Y - pt2.Y
        dz = self.Z - pt2.Z

        return (dx ** 2 + dy ** 2 + dz ** 2) ** 0.5

    @property
    def Coordinates(self):
        return self.X, self.Y, self.Z


def rect_around_group(n, m, sx, sy, d, width, z):
    # z in [m], other dimensions in diameters

    x1 = -width*d
    y1 = -width*d

    x2 = ((n-1)*sx + width)*d
    y2 = ((m-1)*sy + width)*d

    return (x1, y1, z), (x2, y1, z), (x2, y2, z), (x1, y2, z)


def pile_group_preprocess(input_vector, save_filename, bp_path, host, port):

    """

    Custom pile group 3D FEM model (PLAXIS 3D) v0.2

    Model description:

    3D pile is generated by replacing soil material in pile zone with pile material
     (Concrete) - Wished-in-place pile

    Pile installation effects are neglected

    Soil-pile interface is simulated using 2D thin line elements and MC law.
    Rint is the governing interface property

    Structural forces in the pile are obtained using 'dummy' beam through the pile
     center. This beam has low stiffness and zero weight, in order to minimize its
      impact on the global stiffness matrix. This also means that the pile behaviour
       is assumed to be like Bernoulli-Euler beam (no deformation in the pile cross
        section plane)

    Pile material is LINEAR ELASTIC

    Hardening Soil and Mohr-Coulomb constitutive models are supported through <<soil
     model>> parameter.

    Analysis is DRAINED
```

```
Optional Concrete pile cap can be inserted in the model in order to simulate fixed
 head pile group. In that case prescribed displacements are set on the TOP surface
  of the pile cap. Dummy beams are set through whole pile cap. As an alternative,
   parallel prescribed displacements can be added to the pile tops in order to
    simulate head fixity.

Only one prescribed displacement is set to final calculation value. Custom number
 of intermediate calculation steps are stored for further evaluation

Finite element mesh is Diameter-Scaled, containing the elements of the same global
 size. Inside the refined zone (prismatic, around WHOLE pile group), smaller
  elements are introduced through refinements coefficient. Size of small and main
   finite elements is introduced as input parameters (in Diameters).

Eoed and Eur are constrained with E50 through appropriate input multipliers

# UNPACKING INPUT PARAMETERS VECTOR INTO SCRIPT VARIABLES ######################

N = int(input_vector[0])
M = int(input_vector[1])

sx, sy, D, length1, length2_0, tol, gok_top_ratio, load_angle, bound, bottom_z,
 max_z_x = input_vector[2:13]

ref_bottom, fe2, fe1 = input_vector[13:16]

soil_model = int(input_vector[16])

uns_w, coh, phi, e50, e_ur_m, nu_soil, p_m, rint, E_p, nu_p,
w_p = input_vector[17:28]

virt_thickness, cap_thickness = input_vector[28:30]
cap_type = int(input_vector[30])
cap_extent = input_vector[31]

length2 = length2_0 + cap_thickness*D

# Constraints - psi with phi. E50 with Eoed. Eur. nu  .........................
if phi > 30:
    psi = phi - 30
else:
    psi = 0.0

e_oed = e50*1
e_ur = e50*e_ur_m

if N == 1 and M == 1:  # Override loading angle for single pile case
    load_angle = 0

# ............................................................................
# User defined model settings
dummy_mult = 1e6
```

```
split_vector = [0.015, 0.03, 0.06, 0.1]     # Vector 1
steps = 20

# START ###########################################################################

s_i, g_i = new_server(host, port)
s_i.new()
g_i.save(save_filename)
# Save at the beginning. If something is wrong, we know which simulation was
 running

# MATERIAL SETS ###################################################################
Soil_M = None

if soil_model == 1:
    Soil_M = MC_soil_drained(g_i, 'MC-Soil', (uns_w, uns_w, e50, nu_soil, coh,
     phi, psi, rint))

if soil_model == 2:
    Soil_M = HS_soil_drained(g_i, 'HS-Soil', (uns_w, uns_w, e50, e_oed, e_ur, p_m,
     nu_soil, coh, phi, psi, rint))

Pile_Material = LE_soil(g_i, 'Concrete-Soil', (w_p, E_p, nu_p))

Dummy_Material = LE_Beam(g_i, 'Dummy-Beam', (0, E_p/dummy_mult, nu_p, A_Circle(D),
 I_Circle(D), I_Circle(D), 0))

# BOREHOLES AND DOMAIN ############################################################
# Setup model boundaries and mesh coefficients using scaled model parameters

min_Z_bound = length1 + bottom_z*D

# Calculate model domain size
X1 = ((N-1)*sx + 2*bound)*D
Y1 = ((M-1)*sy + 2*bound)*D
Z1 = length2 + min_Z_bound

zone_refinements = fe2/fe1
global_mesh = (fe1*D) / ((X1 ** 2 + Y1 ** 2 + Z1 ** 2) ** 0.5)

# BOREHOLES AND DOMAIN

g_i.SoilContour.initializerectangular(-bound*D,
                                      -bound*D,
                                      ((N-1)*sx + bound)*D,
                                      ((M-1)*sy + bound)*D)

borehole = g_i.borehole(0, 0)
# Create Borehole and ground water level

borehole.Head = -min_Z_bound
soil_layer = g_i.soillayer(min_Z_bound)
# Create Soil Layer and assign material
```

```
soil_layer.Soil.Material = Soil_M

# STRUCTURES ##################################################################
g_i.gotostructures()

BottomPileVolumes = []
Surfaces_to_Create = []

for row in range(0, N):
    for col in range(0, M):

        Xp = row*sx*D
        Yp = col*sy*D

        # Setup points - coordinates
        TopPoint = Point(Xp, Yp, length2)
        BottomPoint = Point(Xp, Yp, -length1)
        GroundPoint = Point(Xp, Yp, 0)
        CapPoint = Point(Xp, Yp, length2-virt_thickness*D)

        # Create 3D volume pile
        ground_surf = surf_1(g_i, circle_polyline(g_i, D/2, GroundPoint), True)

        bottom_volume, bottom_soil = g_i.extrude(ground_surf, (0, 0, -length1))
        bottom_soil.Material = Soil_M
        BottomPileVolumes.append(bottom_volume)

        upper_volume, upper_soil = g_i.extrude(ground_surf, (0, 0, length2))
        upper_soil.Material = Pile_Material

        g_i.delete(ground_surf)

        # Create Dummy Beam
        Dummy_Beam = g_i.beam(TopPoint.Coordinates, BottomPoint.Coordinates)[3]
        Dummy_Beam.Material = Dummy_Material

        # Create Interface
        surf1, surf2, surf3, group_1 = g_i.decomposesrf(bottom_volume)
        g_i.delete(surf1, surf3)
        # Remove not needed surface

        g_i.posinterface(surf2)
        # Pile shaft

        # Create Pile Cap. Choose between real 3D (CASE 1) or parallel prescribed
         displacements (CASE 0)

        if cap_type == 0:
            Surfaces_to_Create.append(surf_1(g_i, circle_polyline(g_i, D/2,
             TopPoint), True))       # Top circle

        if cap_type == 1:
            Surfaces_to_Create.append(surf_1(g_i, circle_polyline(g_i, D/2,
             TopPoint), True))       # Top circle

            Surfaces_to_Create.append(surf_1(g_i, circle_polyline(g_i, D/2,
             CapPoint), True))       # Bottom circle

if cap_type == 2:
    pts_rect_cap_1 = rect_around_group(N, M, sx, sy, D, cap_extent, length2)
    pts_rect_cap_2 = rect_around_group(N, M, sx, sy, D, cap_extent,
    length2-virt_thickness*D)

    Surfaces_to_Create.append(g_i.surface(pts_rect_cap_1[0], pts_rect_cap_1[1],
                                          pts_rect_cap_1[2], pts_rect_cap_1[3]))
                                                      # Top rect

    Surfaces_to_Create.append(g_i.surface(pts_rect_cap_2[0], pts_rect_cap_2[1],
                                          pts_rect_cap_2[2], pts_rect_cap_2[3]))

    cap_vol, cap_soil = g_i.extrude(Surfaces_to_Create[0],
    (0, 0, -cap_thickness*D))

    cap_soil.Material = Pile_Material

for surface in Surfaces_to_Create:
    surf_displacement(g_i, surface, 'Prescribed', 'Prescribed', 'Free')

# Create refinements zone around whole pile group

pts_rect_top = rect_around_group(N, M, sx, sy, D, max_z_x, -fe2*D)
rectangle_top = g_i.surface(pts_rect_top[0], pts_rect_top[1], pts_rect_top[2],
 pts_rect_top[3])

extruded_vol, extruded_soil = g_i.extrude(rectangle_top,
(0, 0, -(min_Z_bound-(ref_bottom+fe2)*D)))

extruded_soil.Material = Soil_M
g_i.delete(rectangle_top)

# MESH ########################################################################
g_i.gotomesh()

# Setup mesh refinements
for item in (g_i.Points[:] + g_i.Lines[:] + g_i.Surfaces[:]):
# Reset all coarseness to 1

    item.CoarsenessFactor = 1

for volume in g_i.Volumes[:]:
# Setup all created volumes to refinement factor

    volume.CoarsenessFactor = zone_refinements
    # Scaled model geometry parametrization
    echo = g_i.mesh(global_mesh)
    elements = int(echo.split()[1])
    nodes = int(echo.split()[3])

    # PHASES ######################################################################
    g_i.gotostages()

    # [InitialPhase] - already created (default)

    # [Construction] - Change Pile material to Concrete, activate Upper pile Volume,
     DummyBeams and Interfaces

    phase_2 = create_phase(g_i, 'Construction', None)

    for vol in BottomPileVolumes:
        g_i.setmaterial(vol, phase_2, Pile_Material)

    for item in g_i.Beams[:] + g_i.Soils[:] + g_i.SurfaceDisplacements[:] +
     g_i.Interfaces[:]:

        item.Active[phase_2] = True

    # [Prescribed Displacements] .....................................................

    fx_list = [round(item*D*Cosinus(load_angle)/gok_top_ratio, 4)
    for item in split_vector]

    fy_list = [round(item*D*Sinus(load_angle)/gok_top_ratio, 4)
    for item in split_vector]

    for fx, fy in zip(fx_list, fy_list):

        load_phase = create_phase(g_i,
        'Ux=' + str(fx) + ' m ' + 'Uy=' + str(fy) + ' m', (tol, steps))

        for s_d in g_i.SurfaceDisplacements[:]:
            s_d.ux[load_phase] = fx
            s_d.uy[load_phase] = fy

    # CALCULATION ##################################################################

    g_i.calculate()

    calc_success = not any(phase.CalculationResult != 1 for phase in g_i.Phases)

    g_i.save(save_filename)
    s_i.close()

    return calc_success, nodes, elements


def pile_group_check_mat(input_vector, save_filename, bp_path, host, port):
    # This short PLAXIS script creates the material set with given input parameters.
     If error inside the PLAXIS occurs, this script returns FALSE, and continues to
      check the next simulation. Results are written into the dataset, and displayed
       on Console. After datasets are checked, we can run real simulations
       with new script.

    soil_model = int(input_vector[16])

    uns_w, coh, phi, e50, e_ur_m, nu_soil, p_m, rint = input_vector[17:25]

    # Constraints - psi with phi. E50 with Eoed. Eur. nu

    if phi > 30:
        psi = phi - 30
    else:
        psi = 0.0

    e_oed = e50*1
    e_ur = e50*e_ur_m

    s_i, g_i = new_server(host, port)
    s_i.new()

    try:        # Error can occur in this block of code.
        if soil_model == 1:
            MC_soil_drained(g_i, 'MC-Soil', (uns_w, uns_w, e50, nu_soil, coh, phi,
             psi, rint))

        if soil_model == 2:
            HS_soil_drained(g_i, 'HS-Soil', (uns_w, uns_w, e50, e_oed, e_ur, p_m,
             nu_soil, coh, phi, psi, rint))

    except:
        s_i.close()
        print(False, save_filename)
        return False, 0, 0

    s_i.close()
    print(True, save_filename)
    return True, 0, 0


def pile_group_postprocess(filename, host, port):
    """
    Custom pile group 3D FEM model (PLAXIS 3D) v0.1 - POSTPROCESSING

    Model description:

    At this stage, only results from Dummy Beams are extracted for postprocessing.
     Soil and interface results remain in the raw simulation results.

    Loading is applied in following phases:
    1. K0 - InitialPhase
```

```
2. Construction (Horizontal Displacements are set to zero, small vertical
 displacements occur due to gravity load)
3+ Horizontal displacement increments

Results are extracted phase by phase, starting from <<Construction>> phase where
 Horizontal Displacements are 0

Results are extracted from MESH NODES and using results SMOOTHING in PLAXIS

Result Types: Displacements (x, y, z, total), Structural Forces (N, Q12, Q13, M2,
 M3), Node Coordinates (x, y, z)

In postprocessing, raw data is splitted pile by pile using pile coordinates filter
"""

dummy_mult = 1e6

# START

try:
    s_o, g_o = new_server(host, port)
    s_o.open(filename)

    ResultTypes_Beam = ['X', 'Y', 'Z', 'Ux', 'Uy', 'Uz', 'Utot',
     'LinearStructureN', 'LinearStructureQ12', 'LinearStructureQ13',
      'LinearStructureM2', 'LinearStructureM3']

    # Determine number of PLAXIS Results Table rows
    rows = len(g_o.getresults(g_o.Phases[1], g_o.ResultTypes.Beam.X,
    'node', True)[:])

    # Determine number of stored loading steps. K0 calculation is ignored
    saved_steps = [step for step in g_o.Steps[1:] if hasattr(step.Info,
     'GlobalError')]

    layers = len(saved_steps)

    # Extract results for Beams, step by step
    results = np.zeros((rows, len(ResultTypes_Beam), layers))

    for i, step in enumerate(saved_steps):
        for j, res_type in enumerate(ResultTypes_Beam):
            results[:, j, i] = g_o.getresults(step, getattr(g_o.ResultTypes.Beam,
             res_type), 'node', True)[:]

    # Multiply Q12, Q13, M2, M3 with dummy multiplier to get real results for
     Dummy Beam structural forces

    for r in range(0, rows):
        for p in range(0, layers):
            for c in range(8, 12):
                results[r, c, p] *= dummy_mult
        s_o.close()

        return True, results

    except:
    # If any error happens during this stage, we return False as outcome

        return False, None

def blueprint_demo_1(input_vector, save_filename, bp_path, host, port):

    # UNPACKING INPUT PARAMETERS VECTOR INTO SCRIPT VARIABLES #######################
    D = input_vector[4]
    max_disp, load_angle = input_vector[32:34]

    # START #########################################################################
    s_i, g_i = new_server(host, port)
    s_i.open(bp_path)

    elements, nodes = (0, 0)

    # PHASES ########################################################################
    g_i.gotostages()

    # Prescribed Displacements

    for load_phase in g_i.Phases[2:]:
        fx = round(max_disp*Cosinus(load_angle)*D, 4)
        fy = round(max_disp*Sinus(load_angle)*D, 4)

        g_i.setcurrentphase(load_phase)
        load_phase.Identification = 'Ux=' + str(fx) + ' m ' + 'Uy=' + str(fy) + ' m'

        for sd in g_i.SurfaceDisplacements[:]:
            sd.ux[load_phase] = fx
            sd.uy[load_phase] = fy

        for phase in g_i.Phases[:]:
            phase.ShouldCalculate = True

    # CALCULATION ###################################################################

    g_i.calculate()

    calc_success = not any(phase.CalculationResult != 1 for phase in g_i.Phases)

    g_i.save(save_filename)
    s_i.close()

    return calc_success, nodes, elements
```

PLAXIS Pre-/Postprocessing Scripts - Additional Functions

M2_Solvers_PLAXIS_Functions.py

```
import math


# BASIC FORMULAS ####################################################################

def K0_Jaky(phi):
    return 1-Sinus(phi)

def A_Circle(D):
    return D**2*math.pi/4

def Sinus(angle):
    return math.sin(math.radians(angle))

def Cosinus(angle):
    return math.cos(math.radians(angle))

def G_Modulus(E, nu):
    return E/(2*(1+nu))

def I_Circle(D):
    return D**4*math.pi/64

# Material models ###################################################################


def LE_soil(G, title, input_list):
    # Non-porous analysis is assumed - this material is used for simulation of
     structural volume elements

    w, E, nu = input_list

    material = G.soilmat()

    material.setproperties(
        'MaterialName', title,
        'SoilModel', 1,
        'DrainageType', 4,

        'gammaUnsat', w,

        'Eref', E,
        'nu', nu,
        'Gref', E / (2 * (1 + nu)))

    return material

def MC_soil_drained(G, title, input_list):
    # Drained analysis is assumed
    unsat_w, sat_w, E, nu, c, phi, psi, rint = input_list

    material = G.soilmat()

    material.setproperties(
        'MaterialName', title,
        'SoilModel', 2,
        'DrainageType', 0,

        'gammaUnsat', unsat_w,
        'gammaSat', sat_w,

        'Gref', G_Modulus(E, nu),
        'nu', nu,

        'cref', c,
        'phi', phi,
        'psi', psi,

        'K0Primary', K0_Jaky(phi),
        'K0Secondary', K0_Jaky(phi),

        'DefaultValuesAdvanced', True,
        'Rinter', rint)

    return material


def HS_soil_drained(G, title, input_list):
    # p=100 kPa is assumed. It is also a default value (default)
    # Rf = 0.9 is assumed (default value)
    # Drained Analysis is assumed

    unsat_w, sat_w, e50, e_oed, e_ur, power_m, nu, c, phi, psi, rint = input_list

    material = G.soilmat()

    material.setproperties(
        'MaterialName', title,
        'SoilModel', 3,
        'DrainageType', 0,

        'gammaUnsat', unsat_w,
        'gammaSat', sat_w,

        'E50ref', e50,
        'EoedRef', e_oed,
        'Gref', G_Modulus(e_ur, nu),
        'nu', nu,

        'powerm', power_m,
        'cref', c,
        'phi', phi,
        'psi', psi,

        'K0nc', K0_Jaky(phi),
        'K0Primary', K0_Jaky(phi),
        'K0Secondary', K0_Jaky(phi),

        'DefaultValuesAdvanced', True,
        'Rinter', rint)

    return material

def LE_Beam(G, title, input_list):

    w, E, nu, area, I2, I3, I23 = input_list

    material = G.beammat()

    material.setproperties(
        'MaterialName', title,
        'IsLinear', True,

        'w', w,
        'E', E,
        'nu', nu,

        'A', area,
        'Iyy', I3,
        'Izz', I2,
        'Iyz', I23)

    return material


# Geometry ..........................................................................

def circle_polyline(G, radius, center):
    # Create horizontal circle line in PLAXIS

    xs, ys, zs = center.Coordinates
    line_1, point_1 = G.polycurve((xs, ys - radius, zs), (1, 0, 0), (0, 1, 0), 'Arc',
     (0, 360, radius))

    return line_1


def surf_1(G, line, delete_line=True):

    surf = G.surface(line)

    if delete_line:
        G.delete(line)
    return surf


def surf_displacement(G, surface, x, y, z):

    surf_disp = G.surfdispl(surface)
    surf_disp.Displacement_x = x
    surf_disp.Displacement_y = y
    surf_disp.Displacement_z = z

    return surf_disp


# Phases .............................................................................

def create_phase(G, title, tol_steps):

    new_phase = G.phase(G.Phases[-1])
    G.setcurrentphase(new_phase)
    new_phase.Identification = title

    if tol_steps is not None:
        new_phase.Deform.UseDefaultIterationParams = False
        new_phase.Deform.ToleratedError = tol_steps[0]/100
        new_phase.MaxStepsStored = tol_steps[1]

    return new_phase
```

User Interface Classes `M2_WidgetClasses.py`

```
import numpy as np
import time

from PyQt5.QtCore import pyqtSignal, Qt
from PyQt5.QtWidgets import QMessageBox, QAbstractScrollArea, QPushButton,
 QTableWidget, QDialog, QLineEdit, QLabel

from PyQt5.QtWidgets import QComboBox, QCheckBox, QGridLayout, QTextEdit, QTabWidget,
 QWidget

from Main_Functions import popup_open_save, popup_message, popup_yes_no_dialog,
 popup_select_folder, export_nd_xl

from Main_Functions import set_shapes, all_processed, global_shape, parse_xl_nd_full,
 measure_xl_3D, xl_dimensions

from Main_ConfigConstants import DefaultSimDir, Max_Widgets

from Main_WidgetClasses import IconCell, fill_table_items, StringCell,
 cell_label_button
```

```
class DataTable(QTableWidget):

    table_added = pyqtSignal(int, object)

    def __init__(self, table_data, table_headers, result_tables_list):
        super().__init__(len(table_data), len(table_headers))

        self.__ResultTables = result_tables_list
        self.__Data_Buttons = []
        self.__SimNames = [item[0] for item in table_data]
        self.__Shapes = set_shapes(self.__ResultTables)
        # Read numpy table shapes

        self.setHorizontalHeaderLabels(table_headers)
        # Fill Table items

        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        fill_table_items(self, table_data)

        for r, shape in enumerate(self.__Shapes):
        # Create buttons

            button = DataTable.__table_button(shape)
            button.clicked.connect(self.__emit_button_clicked)
            self.setCellWidget(r, 5, cell_label_button('', button))
            self.__Data_Buttons.append(button)

        self.clearSelection()
        self.resizeColumnsToContents()

    @property
    def GlobalShape(self):
        return global_shape(self.__Shapes)

    @staticmethod
    def __table_button(shape):

        enabled = True

        if shape is not None:
            if len(shape) <= 3:
                caption = 'Show Data Table ' + str(shape)
            else:
                caption = '4D+ Table'
                enabled = False
        else:
            caption = 'Add Data Table'

        new_button = QPushButton(caption)
        new_button.setEnabled(enabled)

        return new_button

    def __emit_button_clicked(self):

        r = 0
        # Determine which button was clicked, r is a button index

        for r in range(0, len(self.__Data_Buttons)):
            if self.sender() == self.__Data_Buttons[r]:
                break

        button_caption = self.__Data_Buttons[r].text()
        # Button Found. Define what to do depending on button caption

        if button_caption != 'Add Data Table':  # Showing Data
            Numpy_Table(self.__ResultTables[r])

        elif button_caption == 'Add Data Table':  # Adding Data Table

            filename = popup_open_save('Open', DefaultSimDir, 'Import Data',
            'Data files (*.xls *.xlsx *.npy)')

            if filename != '':  # Something Chosen

                try:
                # file load errors catcher

                    data_to_add = None

                    if filename.endswith('.npy'):
                    # Determine file type

                        data_to_add = np.load(filename)
                        # Numpy array

                    else:
                    # Parse Excel File
                        rows, columns, sheets = measure_xl_3D(filename)
                        if rows > 0 and columns > 0:
                            data_to_add = parse_xl_nd_full(filename, rows, columns
                            , sheets)

                    if data_to_add is None:
                        popup_message('Error in the Excel file. Nothing to Load!
                         Please check the data!', QMessageBox.Critical,
                         'Excel Table Error', QMessageBox.Ok)

                    else:
                        self.__add_Data_Table(data_to_add, r)

                except:
```

```
                # Catched: [InvalidFileException], [BadZipFile], [OSError]
                popup_message('Data Import Error! ', QMessageBox.Critical,
                'Import Error', QMessageBox.Ok)

def __add_Data_Table(self, data_to_add, r):

    if popup_yes_no_dialog('Add Data Table with shape ' +
     str(np.shape(data_to_add)) + ' to Dataset?', QMessageBox.Information,
      'Confirm Continue') == QMessageBox.Ok:

        self.__ResultTables[r] = data_to_add
        self.__Shapes[r] = np.shape(data_to_add)

        new_button = DataTable.__table_button(np.shape(data_to_add))
        new_button.clicked.connect(self.__emit_button_clicked)
        self.__Data_Buttons[r] = new_button

        self.setItem(r, 5, IconCell(True))
        self.setCellWidget(r, 5, cell_label_button('', new_button))

        self.table_added.emit(r, data_to_add)

def __all_processed(self):
    return all_processed(self.__ResultTables)

def __nothing_processed(self):
    for table in self.__ResultTables:
        if table is not None:
            return False
    return True

def __default_captions(self):

    if self.GlobalShape is not None and len(self.GlobalShape) <= 3:
    # Consistent/max 3D tables are results

        rows, columns, tabs = xl_dimensions(self.GlobalShape)

        row_captions = ['Row_' + str(r) for r in range(1, rows + 1)]
        columns_captions = ['Column_' + str(c) for c in range(1, columns + 1)]
        tab_captions = ['Tab_' + str(t) for t in range(1, tabs + 1)]

        return row_captions, columns_captions, tab_captions

    else:
        return None

def can_export(self):
    # Check if there are any results for export. Also, if some results are
     missing, ask user for continue

    # Return True if there is something for export. False - program will not
     continue
    proceed = True

    if self.__nothing_processed() is True:
        popup_message('Nothing for Export!', QMessageBox.Warning, 'No Results',
         QMessageBox.Ok)

        proceed = False

    else:
        if not self.__all_processed():
            if popup_yes_no_dialog('Some results are missing. Proceed?',
             QMessageBox.Information,
                                   'Confirm Continue') == QMessageBox.Cancel:
                proceed = False

    return proceed

def export_numpy_tables(self):

    saved = False
    folder_to_save = popup_select_folder(DefaultSimDir)

    if folder_to_save != '':
        for j, table in enumerate(self.__ResultTables):
            if table is not None:
                np.save(folder_to_save + '\\' + self.__SimNames[j], table)

        popup_message('Results Succesfully Exported as Numpy Tables.',
         QMessageBox.Information, 'Export Succesfull', QMessageBox.Ok)

        saved = True

    return saved

def __export_joined(self, r_c, c_c):

    tab_names = []
    data_tables = []
    rows, columns, tabs = xl_dimensions(self.GlobalShape)

    for j, table in enumerate(self.__ResultTables):
        if table is not None:
            tab_names.append(self.__SimNames[j])
            data_tables.append(table)

    data_table_2 = np.ndarray((rows, columns, len(data_tables)))

    for j in range(0, len(data_tables)):

        if len(self.GlobalShape) == 2:
            data_table_2[:, :, j] = data_tables[j]
        elif len(self.GlobalShape) == 1:
            data_table_2[:, 0, j] = data_tables[j]

    exported = False
    wb = export_nd_xl(r_c, c_c, tab_names, data_table_2)
    filename = popup_open_save('Save', DefaultSimDir, 'Save As', 'Excel files
     (*.xlsx)')

    if filename != '':

        try:
            wb.save(filename)
            exported = True

        except PermissionError:
            popup_message('Permission Error. File already opened. Close it and
             repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

    return exported

def __export_sim_sim_excel(self, r_c, c_c, t_c):

    folder_to_save = popup_select_folder(DefaultSimDir)
    successfull = False

    if folder_to_save != '':
        for j, table in enumerate(self.__ResultTables):
            if table is not None:

                wb = export_nd_xl(r_c, c_c, t_c, table)

                try:
                    wb.save(folder_to_save + '\\' + self.__SimNames[j] + '.xlsx')
                    successfull = True

                except PermissionError:
                    popup_message('Permission Error. File already opened.
                    Close it and repeat.', QMessageBox.Critical, 'File Error',
                     QMessageBox.Ok)

    return successfull

def export_excel_consistent(self):

    input_data = Input_Form_Excel_Captions(self.__default_captions(),
     self.GlobalShape).ReturnValues

    if input_data is not None:
        (joined, sim_by_sim), (r_c, c_c, t_c) = input_data

        successfull = False

        if joined:
                successfull = self.__export_joined(r_c, c_c)

            if sim_by_sim:
                successfull = self.__export_sim_sim_excel(r_c, c_c, t_c)

            if successfull:
                popup_message('Results Succesfully Exported to Excel format.',
                 QMessageBox.Information, 'Export Succesfull', QMessageBox.Ok)

    def export_all(self):

        folder_to_save = popup_select_folder(DefaultSimDir)

        if folder_to_save != '':
            successfull = True

            for j, shape in enumerate(self.__Shapes):
                if shape is not None:

                    if len(shape) > 3:
                        np.save(folder_to_save + '\\' + self.__SimNames[j],
                         self.__ResultTables[j])

                    else:
                        r, c, t = xl_dimensions(shape)
                        wb = export_nd_xl(None, None,
                        ['Tab_'+str(i) for i in range(0, t)], self.__ResultTables[j])

                        try:
                            wb.save(folder_to_save + '\\' + self.__SimNames[j] +
                             '.xlsx')

                        except PermissionError:
                            popup_message('Permission Error. File already opened.
                             Close it and repeat.', QMessageBox.Critical, 'File
                              Error', QMessageBox.Ok)

                            successfull = False

            if successfull:
                popup_message('Results Succesfully Exported!',
                 QMessageBox.Information, 'Export Succesfull', QMessageBox.Ok)


class ActiveTable_2(QTableWidget):

    cell_button_clicked = pyqtSignal(int, int)

    def __init__(self, table_data, table_captions, columns, button_captions, start_c):
        super().__init__(len(table_data), len(table_captions))

        # Create simple table widget
        self.setHorizontalHeaderLabels(table_captions)
        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        fill_table_items(self, table_data)

        # Create Custom Widgets inside Cells (2 columns). Custom widgets consist of
         Button and Label.

        self.Buttons = []
        self.StartingColumn = start_c

        for c in range(self.StartingColumn, self.StartingColumn+columns):
            button_col = []
            for r in range(0, len(table_data)):
                button = QPushButton(button_captions[c-self.StartingColumn])
                button.clicked.connect(self.emit_clicked)
                self.setCellWidget(r, c, cell_label_button('', button))
                button_col.append(button)

            self.Buttons.append(button_col)

        self.clearSelection()
        self.resizeColumnsToContents()

    @property
    def SelectedRows(self):
        return sorted(set(i.row() for i in self.selectedIndexes()))

    def emit_clicked(self):

        for c in range(0, len(self.Buttons)):
            for r in range(0, len(self.Buttons[0])):
                if self.sender() == self.Buttons[c][r]:
                    self.Buttons[c][r].setEnabled(False)
                     # Disable button on click

                    self.cell_button_clicked.emit(r, c+self.StartingColumn)
                    break

    def reset_cell(self, value, row, col):

        button = self.Buttons[col-self.StartingColumn][row]
        button.setEnabled(True)

        self.setItem(row, col, IconCell(value))
        self.setCellWidget(row, col, cell_label_button('', button))

        self.resizeColumnsToContents()
        self.clearSelection()

    def disable_buttons(self):
        for column_buttons in self.Buttons:
            for button in column_buttons:
                button.setEnabled(False)


class Console(QTextEdit):

    def __init__(self, app):
        super().__init__()
        self.TopApp = app

    def write(self, text_string):
        self.append(text_string)
        self.verticalScrollBar().setValue(self.verticalScrollBar().maximum())
        self.TopApp.processEvents()

    def twrite(self, text_string):
        self.write(time.strftime("%b %d %Y %H:%M:%S") + '    ' + text_string)


class Numpy_Table(QDialog):

    def __init__(self, results_table):

        super().__init__()

        tab_widget = QTabWidget()
        top_layout = QGridLayout()
        self.setLayout(top_layout)
        top_layout.addWidget(tab_widget)

        dimensions = list(np.shape(results_table))

        # Set rows, columns, tabs
        tabs = 1
        columns = 1
        if len(dimensions) == 3:
            rows, columns, tabs = (dimensions[0], dimensions[1], dimensions[2])

        elif len(dimensions) == 2:
            rows, columns = (dimensions[0], dimensions[1])
        elif len(dimensions) == 1:
            rows = dimensions[0]

        for tab in range(0, tabs):
            # Create Tab widget page
            tab_page = QWidget()
            tab_page_layout = QGridLayout()
            tab_page.setLayout(tab_page_layout)
            tab_widget.addTab(tab_page, str(tab+1))
            tab_table = QTableWidget(rows, columns)
            tab_page_layout.addWidget(tab_table)

            # Slice Results to 1D or 2D table - create fill_table
            if len(dimensions) == 3:
```

```
            fill_table = results_table[:, :, tab]
            fill_rows = fill_table[:]
        else:
            fill_rows = results_table[:]

        if len(dimensions) > 1:
            for r in range(0, rows):            # Fill Table
                for c in range(0, columns):
                    tab_table.setItem(r, c, StringCell(str(fill_rows[r][c])))
        else:
            for r in range(0, rows):
                tab_table.setItem(r, 0, StringCell(str(fill_rows[r])))

    self.exec_()


class Input_Form_Excel_Captions(QDialog):

    def __init__(self, captions_tuple, glob_shape):

        super(Input_Form_Excel_Captions, self).__init__()

        self.setWindowTitle('Setup Export Table')
        self.setWindowFlags(Qt.CustomizeWindowHint)

        self.layout = QGridLayout()
        self.setLayout(self.layout)

        cmdOK = QPushButton('Accept')
        cmdOK.clicked.connect(self.cmdOK_clicked)

        self.cbNoCaptions = QCheckBox('No Captions')
        self.cbNoCaptions.setChecked(False)
        self.cbNoCaptions.stateChanged.connect(self.cbNoCaptions_set)

        row_labels = captions_tuple[0]
        col_labels = captions_tuple[1]
        tab_labels = captions_tuple[2]

        rows = len(row_labels)
        cols = len(col_labels)
        tabs = len(tab_labels)
        last_text_field_row = max(rows, cols, tabs)+1

        self.RowFields = [QLineEdit(row_labels[r]) for r in range(0, rows)]
        self.ColFields = [QLineEdit(col_labels[c]) for c in range(0, cols)]
        self.TabFields = [QLineEdit(tab_labels[t]) for t in range(0, tabs)]

        self.layout.addWidget(QLabel('Rows'), 0, 0)
        self.layout.addWidget(QLabel('Columns'), 0, 1)
        self.layout.addWidget(QLabel('Sheets'), 0, 2)

        if rows > Max_Widgets or cols > Max_Widgets or tabs > Max_Widgets:
            last_text_field_row = Max_Widgets + 1
            popup_message('Maximum number of fields reached!',
             QMessageBox.Information, 'Input Fields', QMessageBox.Ok)

        self.__add_widgets(rows, self.RowFields, 0)
        self.__add_widgets(cols, self.ColFields, 1)
        self.__add_widgets(tabs, self.TabFields, 2)

        self.cmbJoinSplit = QComboBox()

        if len(glob_shape) <= 2:
            self.cmbJoinSplit.addItems(['Export to Single File',
            'Export to Separate Files'])

            for widget in self.TabFields:
            # Disable Tab Captions if we export to single file

                widget.setEnabled(False)
        else:
            self.cmbJoinSplit.addItems(['Export to Separate Files'])

        self.cmbJoinSplit.currentIndexChanged.connect(self.cmbJoinSplit_changed)

        self.layout.addWidget(self.cmbJoinSplit, last_text_field_row+2, 0)
        self.layout.addWidget(self.cbNoCaptions, last_text_field_row+2, 1)
        self.layout.addWidget(cmdOK, last_text_field_row+2, 2)

        self.ReturnValues = None
        self.exec_()

    def __add_widgets(self, size, widgets_list, col_index):
        if size <= Max_Widgets:
        # Add all widgets from the list

            for j, widget in enumerate(widgets_list):
                self.layout.addWidget(widget, j+1, col_index)
        else:
        # Add first <<Max_Widgets>> from the list

            for i in range(0, Max_Widgets):
                self.layout.addWidget(widgets_list[i], i+1, col_index)

    def cbNoCaptions_set(self):
        for widget in (self.RowFields + self.ColFields):
            widget.setEnabled(not self.cbNoCaptions.isChecked())

    def cmdOK_clicked(self):

        data_empty = False

        for text_field in (self.RowFields + self.ColFields + self.TabFields):
            if text_field.text() == '':
                data_empty = True
                popup_message('Please fill all fields!', QMessageBox.Warning,
                'Input Error', QMessageBox.Ok)

                break

        if not data_empty:

            new_rows = [r_f.text() for r_f in self.RowFields]
            new_cols = [c_f.text() for c_f in self.ColFields]
            new_tabs = [t_f.text() for t_f in self.TabFields]

            if self.cmbJoinSplit.currentText() == 'Export to Single File':
                joined, sim_by_sim = (True, False)
            else:
                joined, sim_by_sim = (False, True)

            if self.cbNoCaptions.isChecked():
                titles = (None, None, new_tabs)
            else:
                titles = (new_rows, new_cols, new_tabs)

            self.ReturnValues = ((joined, sim_by_sim), titles)
            self.close()

    def cmbJoinSplit_changed(self):

        if self.cmbJoinSplit.currentText() == 'Export to Separate Files':
            enabled = True
        else:
            enabled = False

        for widget in self.TabFields:
            widget.setEnabled(enabled)
```

Module 3 - System Identification Package `M3.py`

```
import sys
import numpy as np

from PyQt5.QtCore import Qt, QObject, QRunnable, pyqtSignal, pyqtSlot, QThreadPool
from PyQt5.QtGui import QColor

from PyQt5.QtWidgets import QComboBox, QMessageBox, QGridLayout, QDialog, QPushButton,
 QLineEdit, QLabel, QMdiArea

from PyQt5.QtWidgets import QMainWindow, QApplication, QMdiSubWindow, QWidget

from Main_Functions import load_object, parse_xl_nd, popup_message,
 popup_yes_no_dialog, popup_open_save

from Main_Functions import export_nd_xl, vector_nd, nd_vector, set_shapes,
 all_processed, global_shape

from Main_ConfigConstants import DefaultSimDir, MinimumMMTrain

from Main_WidgetClasses import Input_ExcelCaptions
from MyClasses1 import SnapshotMatrix

from M3_Metamodels import MM_LIST, MM_CALLER_2
from M3_GSA import GSA_LIST, GSA_CALLER
from M3_BackAnalysis import BA_CALLER, BA_LIST
from M3_Observers import MM_BA_Observer, BA_GSA_Worker
from M3_WidgetClasses import Snapshot_Table, BA_Parameters, MMParameterTable, PB,
 Simple_Table

from M3_Classes import Metamodel
from M3_Plotting import XY_Plot, Plotter_2D


def on_off_widgets(on_off, w_list):
    for widget in w_list:
        widget.setEnabled(on_off)


class TrainerSignals(QObject):

    finished = pyqtSignal(object, tuple)
    started = pyqtSignal()


class Trainer(QRunnable):

    def __init__(self, input_data, test_data):

        super(Trainer, self).__init__()

        self.TSignals = TrainerSignals()
        self.__InputData = input_data
        self.__TestData = test_data

    @pyqtSlot()
    def run(self):

        self.TSignals.started.emit()

        input1, input2, input3 = self.__InputData

        new_metamodel = Metamodel(input1, input2, input3)

        # Test new Metamodel

        input_test_m = self.__TestData[0]
        output_test_m = self.__TestData[1]
        nrmse_1 = new_metamodel.Training_NRMSE

        if input_test_m is not None:
            nrmse_2 = new_metamodel.test(input_test_m, output_test_m)
        else:
            nrmse_2 = None

        self.TSignals.finished.emit(new_metamodel, (nrmse_1, nrmse_2))


class BA_Window(QDialog):

    def __init__(self, metamodel):

        super(BA_Window, self).__init__()

        self.Metamodel = metamodel[0]
        self.InputCaptions = metamodel[1][3]
        self.SnapCaptions = metamodel[2][3]
        self.BAPool = QThreadPool()

        self.setFixedSize(700, 1000)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Back Analysis')

        layout = QGridLayout()
        self.setLayout(layout)

        self.tblInput = BA_Parameters(self.InputCaptions, metamodel[1][2], 'Input')
        self.tblSnapshots = BA_Parameters(self.SnapCaptions, metamodel[2][2],
         'Snapshots')

        self.pbBAProgress = PB()

        self.cmbBA = QComboBox()
        self.cmbBA.addItems(BA_LIST)

        self.cmdStart = QPushButton('Start Back Analysis')
        self.cmdClose = QPushButton('Close')
        self.cmdExport = QPushButton('Export Results to Excel')
        self.cmdPlotResults = QPushButton('Plot Results')

        self.cmdStart.clicked.connect(self.cmdStart_clicked)
        self.cmdClose.clicked.connect(self.cmdClose_clicked)
        self.cmdExport.clicked.connect(self.cmdExport_clicked)
        self.cmdPlotResults.clicked.connect(self.cmdPlotResults_clicked)

        layout.addWidget(QLabel('Select Back Analysis Algorithm'), 0, 0)
        layout.addWidget(self.cmbBA, 1, 0)
        layout.addWidget(QLabel('Input Parameters'), 2, 0)
        layout.addWidget(self.tblInput, 3, 0)
        layout.addWidget(QLabel('Snapshots'), 4, 0)
        layout.addWidget(self.tblSnapshots, 5, 0)
        layout.addWidget(self.cmdStart, 6, 0)
        layout.addWidget(self.cmdPlotResults, 7, 0)
        layout.addWidget(self.cmdExport, 8, 0)
        layout.addWidget(self.cmdClose, 9, 0)
        layout.addWidget(self.pbBAProgress, 10, 0)

        self.Identified = None
        self.Search = None
        self.BA_Object = None

        self.WList1 = [self.cmdStart, self.cmdClose]
        self.WList2 = [self.cmdExport, self.cmdPlotResults]
        on_off_widgets(False, self.WList2)

        self.exec_()

    def cmdStart_clicked(self):

        if popup_yes_no_dialog('Process Cannot be stopped! Proceed?',
         QMessageBox.Question, 'Confirm Start') == QMessageBox.Ok:

            ba_input = self.tblInput.analysis_data()
            # Obtain BA search parameters

            ba_output = self.tblSnapshots.analysis_data()

            self.Search = ba_output[1]

            if ba_input is not None and ba_output is not None:

                ba_algorithm_class, input_p_list = BA_CALLER[self.cmbBA.currentText()]
                  # Obtain BA from BA database

                ba_alg_input = InputParameters(input_p_list, 'Input
                 Parameters').ReturnValues

                if ba_alg_input is not None:

                    # Initialize Back analysis algorithm object instance and attach
                     observers for GUI

                    self.BA_Object = ba_algorithm_class(self.Metamodel, ba_alg_input,
                     ba_input, ba_output, self.InputCaptions, self.SnapCaptions)

                    self.BA_Object.attach_observer(MM_BA_Observer
                    (self.pbBAProgress.update))

                    self.pbBAProgress.initialize(self.BA_Object.MaxIterations)
                    on_off_widgets(False, self.WList1 + self.WList2)

                    worker = BA_GSA_Worker(self.BA_Object)
                    worker.Signals.finished.connect(self.BA_ended)
                    self.BAPool.start(worker)
            else:
                popup_message('Check Search Parameters and Repeat!',
                 QMessageBox.Warning, 'Input Error', QMessageBox.Ok)

    def cmdClose_clicked(self):
        self.close()

    def cmdExport_clicked(self):
        self.BA_Object.export_results()

    def cmdPlotResults_clicked(self):
        self.BA_Object.plot_results()

    @pyqtSlot(tuple)
    def BA_ended(self, result_tuple):

        self.Identified, of_value_final = result_tuple
        ident_vector = nd_vector(self.Metamodel.calculate(vector_nd(self.Identified)))

        self.__update_GUI_end(self.Identified, ident_vector)

    def __update_GUI_end(self, found_vector, identified_vector):
        on_off_widgets(True, self.WList1 + self.WList2)
        self.pbBAProgress.end()

        Dividend, Divisor = (0, 0)
        # Calculate NRMSE using identified parameters

        for i in range(0, len(identified_vector)):
            Dividend += (self.Search[i] - identified_vector[i]) ** 2
            Divisor += self.Search[i] ** 2
        nrmse_perc = np.sqrt(Dividend / Divisor)*100

        # Update I/O tables
        self.tblInput.update_column(5, found_vector)
        self.tblSnapshots.update_column(5, identified_vector)

        popup_message('Back Analysis Completed! NRMSE = ' + str(round(nrmse_perc, 2))
         + '%', QMessageBox.Information, 'Back Analysis Completed', QMessageBox.Ok)


class GSA_Window(QDialog):

    def __init__(self, metamodel):

        super(GSA_Window, self).__init__()

        self.Metamodel = metamodel[0]
        self.GSAPool = QThreadPool()

        self.InputCaptions = metamodel[1][3]
        self.OutputCaptions = metamodel[2][3]
        self.setFixedSize(300, 300)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Global Sensitivity Analysis')

        layout = QGridLayout()
        self.setLayout(layout)

        self.pbGSAProgress = PB()

        self.cmbGSA = QComboBox()
        self.cmbGSA.addItems(GSA_LIST)

        self.cmdStart = QPushButton('Start GSA')
        self.cmdClose = QPushButton('Close')

        self.cmdShow = QPushButton('Show Results')
        self.cmdExport = QPushButton('Export Results to Excel')
        self.cmdPlotResults = QPushButton('Plot Results')

        self.cmdStart.clicked.connect(self.cmdStart_clicked)
        self.cmdClose.clicked.connect(self.cmdClose_clicked)
        self.cmdShow.clicked.connect(self.cmdShow_clicked)
        self.cmdExport.clicked.connect(self.cmdExport_clicked)
        self.cmdPlotResults.clicked.connect(self.cmdPlotResults_clicked)

        self.WList_1 = [self.cmdStart, self.cmdClose]
        self.WList_2 = [self.cmdExport, self.cmdShow, self.cmdPlotResults]

        layout.addWidget(QLabel('Select GSA Algorithm'), 0, 0)
        layout.addWidget(self.cmbGSA, 1, 0)
        layout.addWidget(self.cmdStart, 2, 0)
        layout.addWidget(self.pbGSAProgress, 3, 0)
        layout.addWidget(self.cmdShow, 4, 0)
        layout.addWidget(self.cmdExport, 5, 0)
        layout.addWidget(self.cmdPlotResults, 6, 0)
        layout.addWidget(self.cmdClose, 7, 0)

        on_off_widgets(False, self.WList_2)
        self.GSA_Object = None

        self.exec_()

    def cmdShow_clicked(self):
        self.GSA_Object.show_results()

    def cmdExport_clicked(self):
        self.GSA_Object.export_results()

    def cmdPlotResults_clicked(self):
        self.GSA_Object.plot_results()

    def cmdClose_clicked(self):
        self.close()
    def cmdStart_clicked(self):

        if popup_yes_no_dialog('Process Cannot be stopped! Proceed?',
         QMessageBox.Question, 'Confirm Start') == QMessageBox.Ok:

            self.GSA_Object = None
            gsa_class, input_p_list = GSA_CALLER[self.cmbGSA.currentText()]
            gsa_input = InputParameters(input_p_list, 'Input Parameters').ReturnValues

            if gsa_input is not None:  # Data entered. Proceed to GSA

                self.GSA_Object = gsa_class((gsa_input, self.InputCaptions,
                 self.OutputCaptions), self.Metamodel)

                self.GSA_Object.attach_observer(MM_BA_Observer
                (self.pbGSAProgress.update))

                on_off_widgets(False, self.WList_1 + self.WList_2)
                self.pbGSAProgress.initialize(self.GSA_Object.TotalIterations)

                worker = BA_GSA_Worker(self.GSA_Object)
                worker.Signals.finished.connect(self.GSA_ended)
                self.GSAPool.start(worker)

    @pyqtSlot(tuple)
    def GSA_ended(self):
        popup_message('Global sensitivity analysis Completed!',
         QMessageBox.Information, 'GSA Completed', QMessageBox.Ok)

        on_off_widgets(True, self.WList_1 + self.WList_2)
        self.pbGSAProgress.end()


class Metamodel_Training_Window(QDialog):

    def __init__(self, i_set, o_set):

        super(Metamodel_Training_Window, self).__init__()

        # Unpack input data sets
        self.InputTrainingMatrix, self.InputTestMatrix, self.InputRanges,
         self.InputCaptions = i_set

        self.SnapshotTrainingMatrix, self.SnapshotTestMatrix, self.SnapshotRanges,
         self.SnapshotCaptions = o_set

        self.Metamodel = None
        self.TrainerPool = QThreadPool()

        self.setFixedSize(400, 200)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Metamodel Training')
```

```
layout = QGridLayout()
self.setLayout(layout)

# Write Captions with basic input data .................................
if self.InputTestMatrix is not None:
    test_pts = str(len(self.InputTestMatrix[0]))
else:
    test_pts = 'None'

layout.addWidget(QLabel('Number of Input Parameters: '), 0, 0)
layout.addWidget(QLabel(str(len(self.InputTrainingMatrix))), 0, 1)

layout.addWidget(QLabel('Number of Snapshots: '), 1, 0)
layout.addWidget(QLabel(str(len(self.SnapshotTrainingMatrix))), 1, 1)

layout.addWidget(QLabel('Number of Training Points: '), 2, 0)
layout.addWidget(QLabel(str(len(self.InputTrainingMatrix[0]))), 2, 1)

layout.addWidget(QLabel('Number of Test Points: '), 3, 0)
layout.addWidget(QLabel(test_pts), 3, 1)

self.lblNotice1 = QLabel('Current Status: ')
self.lblNotice2 = QLabel('*** Metamodel Not Trained ***')

self.lblNotice2.setEnabled(False)
self.lblNotice1.setEnabled(False)
# ...........................................................................
self.pbPlot = PB()

self.cmbMM = QComboBox()
self.cmbMM.addItems(MM_LIST)

self.cmdTrain = QPushButton('Train Metamodel')
self.cmdPlot = QPushButton('Plot Training Results')
self.cmdClose = QPushButton('Close')

self.cmdTrain.setEnabled(True)
self.cmdPlot.setEnabled(False)
self.cmdClose.setEnabled(True)

self.cmdTrain.clicked.connect(self.cmdTrain_clicked)
self.cmdPlot.clicked.connect(self.cmdPlot_clicked)
self.cmdClose.clicked.connect(self.cmdClose_clicked)

layout.addWidget(QLabel('Select Metamodel Algorithm'), 4, 0)
layout.addWidget(self.cmbMM, 4, 1)
layout.addWidget(self.lblNotice1, 5, 0)
layout.addWidget(self.lblNotice2, 5, 1)
layout.addWidget(self.cmdTrain, 6, 0)
layout.addWidget(self.cmdPlot, 6, 1)
layout.addWidget(self.cmdClose, 7, 0)
layout.addWidget(self.pbPlot, 7, 1)
```

```
        self.WList_1 = [self.lblNotice1, self.lblNotice2, self.cmdPlot,
        self.cmdTrain, self.cmdClose]

        self.exec_()

    def __MM_train_start(self):
        on_off_widgets(False, self.WList_1)
        self.lblNotice2.setText('*** Metamodel Training In Progress. ***')

    @staticmethod
    def __nrmse_popup(nrmse_tuple):

        nrmse_1 = nrmse_tuple[0]
        nrmse_2 = nrmse_tuple[1]

        if nrmse_2 is not None:
            nrmse_2_m = str(round(100 * nrmse_2, 2))
        else:
            nrmse_2_m = 'None'

        message = 'Metamodel Ready! NRMSE (Training) = ' + str(
            round(100 * nrmse_1, 2)) + '%, NRMSE (Test) = ' + nrmse_2_m + '%.'
        popup_message(message, QMessageBox.Information, 'Metamodel Ready',
         QMessageBox.Ok)

    @pyqtSlot(object, tuple)
    def __MM_prepared(self, new_mm, nrmse):

        self.Metamodel = new_mm

        Metamodel_Training_Window.__nrmse_popup(nrmse)      # Setup GUI Response
        on_off_widgets(True, self.WList_1)
        self.lblNotice2.setText('Metamodel Ready!')

    def __initialize_gui(self):

        if self.SnapshotTestMatrix is not None:
            plots_2 = len(self.SnapshotTestMatrix) * len(self.InputTestMatrix)
        else:
            plots_2 = 0

        total_plots = len(self.InputTrainingMatrix) * len(self.SnapshotTrainingMatrix)
         + plots_2

        self.pbPlot.initialize(total_plots)
        on_off_widgets(False, self.WList_1)

    def __plot_nrmse(self, input_m, snap_m, colors, tt):

        appr_matrix = self.Metamodel.calculate(input_m)

        for i in range(0, len(input_m)):
            x_limits = (self.InputRanges[i, 0], self.InputRanges[i, 1])
            x_label = self.InputCaptions[i]

            for j in range(0, len(snap_m)):
                y_label = self.SnapshotCaptions[j]

                plot1 = XY_Plot(input_m[i, :], snap_m[j, :], colors[0], '+', '-', 1,
                 'Forward Solver', None)
                plot2 = XY_Plot(input_m[i, :], appr_matrix[j, :], colors[1], 'x', '-',
                 1, 'Metamodel', None)

                plotter = Plotter_2D('Metamodel Evaluation - ' + tt + ' Data',
                 x_label, y_label, x_limit=x_limits)

                plotter.plot_multiple([plot1, plot2], 'SCATTER', True, False)

                self.pbPlot.update()

        on_off_widgets(True, self.WList_1)

    def cmdPlot_clicked(self):
        self.__initialize_gui()

        self.__plot_nrmse(self.InputTrainingMatrix, self.SnapshotTrainingMatrix,
        ['c', 'g'], 'Training')

        if self.SnapshotTestMatrix is not None:

            self.__plot_nrmse(self.InputTestMatrix, self.SnapshotTestMatrix, ['b',
             'r'], 'Test')

        self.pbPlot.end()

    def cmdTrain_clicked(self):

        # Input metamodel specific parameters for training algorithm. Obtain data
        from MM database using MM_CALLER_2

        model_input_parameters =
         InputParameters(MM_CALLER_2[self.cmbMM.currentText()], 'Input Metamodel
          Parameters').ReturnValues

        if model_input_parameters is not None:  # Data entered. Proceed to MM training

            worker = Trainer((model_input_parameters,
                              (self.InputTrainingMatrix, self.SnapshotTrainingMatrix,
                               self.InputRanges, self.SnapshotRanges),
                              self.cmbMM.currentText()),
                             (self.InputTestMatrix, self.SnapshotTestMatrix))

            worker.TSignals.started.connect(self.__MM_train_start)
            # Use Threaded worker for slow part of the code
            worker.TSignals.finished.connect(self.__MM_prepared)
            self.TrainerPool.start(worker)

    def cmdClose_clicked(self):
        self.close()


class InputParameters(QDialog):

    def __init__(self, input_list, title):

        super(InputParameters, self).__init__()

        self.ReturnValues = None

        self.setFixedSize(300, 300)
        self.setWindowTitle(title)

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdAccept = QPushButton('OK')
        self.cmdAccept.clicked.connect(self.cmdAccept_clicked)

        self.InputBoxes = []
        for j, item in enumerate(input_list):
            text_box = QLineEdit(str(item[1]))

            layout.addWidget(QLabel(item[0]), j, 0)
            layout.addWidget(text_box, j, 1)
            self.InputBoxes.append(text_box)

        layout.addWidget(self.cmdAccept, len(self.InputBoxes), 0)

        self.exec_()

    def cmdAccept_clicked(self):

        try:
            values = []
            for text_box in self.InputBoxes:

                text1 = text_box.text()

                if text1 == '':
                    raise ValueError
                    # Error #1 - Nothing Entered

                else:
                    values.append(float(text1))
                    # Possible Error #2 - string to float conversion
            if len(values) == len(self.InputBoxes):     # All data is OK
                self.ReturnValues = values
                self.close()

        except ValueError:
        # Catch input error during conversion from string to float + empty fields

            popup_message('Invalid Data! Please check and fill all Fields!',
             QMessageBox.Warning, 'Input Error', QMessageBox.Ok)


class Input_Form_MM_1(QDialog):

    def __init__(self, total_simulations):

        super(Input_Form_MM_1, self).__init__()

        self.ReturnValues = None
        # Return values container variable

        self.__TotalSims = total_simulations

        self.setWindowTitle('Set number of training and test points')

        layout = QGridLayout()
        self.setLayout(layout)

        self.txtTrainP = QLineEdit(str(MinimumMMTrain))
        self.txtTestP = QLineEdit('1')

        cmdOK = QPushButton('OK')
        cmdOK.clicked.connect(self.cmdOK_clicked)

        layout.addWidget(QLabel('Number of Training Points: '), 0, 0)
        layout.addWidget(QLabel('Number of Test Points: '), 1, 0)
        layout.addWidget(self.txtTrainP, 0, 1)
        layout.addWidget(self.txtTestP, 1, 1)
        layout.addWidget(cmdOK, 2, 1)

        self.exec_()

    def cmdOK_clicked(self):

        try:
            training_pts = int(self.txtTrainP.text())
            # Obtain data from input fields
            test_pts = int(self.txtTestP.text())

            if training_pts < MinimumMMTrain:
                popup_message('Invalid number of Training points. Minimum is ' +
                 str(MinimumMMTrain) + '!', QMessageBox.Critical, 'Input Error',
                  QMessageBox.Ok)
            else:
                if training_pts + test_pts != self.__TotalSims:
                    popup_message('Sum of Training and Test points must match total
                     number of simulations', QMessageBox.Warning, 'Input Error',
                      QMessageBox.Ok)

                else:
                    if test_pts == 0:
                        popup_message('Test points not selected! Training points will
                         be used for Metamodel test!', QMessageBox.Information,
                          'Information', QMessageBox.Ok)

                    self.ReturnValues = (training_pts, test_pts)
                    self.close()

        except ValueError:      # Cannot convert to integer
            popup_message('Invalid number of rows or columns!', QMessageBox.Critical,
             'Input Error', QMessageBox.Ok)


class Input_Form_SHNum(QDialog):

    def __init__(self):

        super(Input_Form_SHNum, self).__init__()

        self.setFixedSize(300, 200)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Select number of Snapshots')

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdOK = QPushButton('OK')
        self.cmdOK.clicked.connect(self.cmdOK_clicked)
        self.txtRows = QLineEdit('1')

        layout.addWidget(QLabel('Number of Snapshots (Rows)?'), 0, 0)
        layout.addWidget(self.txtRows, 1, 0)
        layout.addWidget(self.cmdOK, 2, 0)

        self.Answer = None

        self.exec_()

    def cmdOK_clicked(self):

        try:
            rows = int(self.txtRows.text())

            if rows <= 0:
                popup_message('Invalid number of rows', QMessageBox.Critical, 'Input
                 Error', QMessageBox.Ok)
            else:
                self.Answer = rows
                self.close()

        except ValueError:
            popup_message('Invalid number of rows', QMessageBox.Critical, 'Input
             Error', QMessageBox.Ok)


class DisplayMatrix(QDialog):

    def __init__(self, snapshot_matrix, export, message):

        super(DisplayMatrix, self).__init__()

        self.setFixedSize(300, 200)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Display Snapshot Matrix')

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdShow = QPushButton('Show Matrix')
        self.cmdExport = QPushButton('Export to Excel')
        self.cmdAdd = QPushButton('Add to Dataset')
        self.cmdClose = QPushButton('Close')

        self.cmdShow.clicked.connect(self.cmdShow_clicked)
        self.cmdExport.clicked.connect(self.cmdExport_clicked)
        self.cmdAdd.clicked.connect(self.cmdAdd_clicked)
        self.cmdClose.clicked.connect(self.cmdClose_clicked)

        layout.addWidget(QLabel(message), 0, 0)
        layout.addWidget(self.cmdShow, 1, 0)

        if export:
            layout.addWidget(self.cmdExport, 2, 0)

        layout.addWidget(self.cmdAdd, 3, 0)
        layout.addWidget(self.cmdClose, 4, 0)

        self.Answer = None
        self.__Matrix = snapshot_matrix

        self.exec_()

    def cmdClose_clicked(self):
        self.close()

    def cmdShow_clicked(self):
        Simple_Table(self.__Matrix)
    def cmdAdd_clicked(self):
        if popup_yes_no_dialog('Add to Dataset?', QMessageBox.Question, 'Confirm
         Change') == QMessageBox.Ok:
            self.Answer = 'Add To Dataset'

    def cmdExport_clicked(self):

        filename = popup_open_save('Save', DefaultSimDir, 'Save As', 'Excel files
         (*.xlsx)')

        if filename != '':

            wb = export_nd_xl(self.__Matrix.Captions, None, ['Snapshot Matrix'],
             self.__Matrix.Data)

            try:
                wb.save(filename)
                popup_message('Matrix Successfully Exported!',
                 QMessageBox.Information, 'File Error', QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)


class ChooseSnapshot(QDialog):

    def __init__(self, matrices):

        super(ChooseSnapshot, self).__init__()

        self.setFixedSize(300, 200)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Select Snapshot Matrix')

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdShow = QPushButton('Show Matrix')
        self.cmdSelect = QPushButton('Select')
        self.cmbSHM = QComboBox()
        self.cmbSHM.addItems(['Snapshot_' + str(j) for j in range(0, len(matrices))])

        self.cmdShow.clicked.connect(self.cmdShow_clicked)
        self.cmdSelect.clicked.connect(self.cmdSelect_clicked)

        layout.addWidget(QLabel('Select Snapshot Matrix'), 0, 0)
        layout.addWidget(self.cmbSHM, 1, 0)
        layout.addWidget(self.cmdShow, 2, 0)
        layout.addWidget(self.cmdSelect, 3, 0)

        self.Answer = None
        self.__Matrices = matrices
        self.exec_()

    def cmdSelect_clicked(self):
        self.Answer = self.cmbSHM.currentIndex()
        self.close()

    def cmdShow_clicked(self):
        Simple_Table(self.__Matrices[self.cmbSHM.currentIndex()])


class Create_Snapshot_Window(QDialog):

    def __init__(self, result_tables, global_shape_tuple):

        super(Create_Snapshot_Window, self).__init__()

        self.setFixedSize(500, 500)
        self.setWindowFlags(Qt.CustomizeWindowHint)
        self.setWindowTitle('Create Snapshot Matrix')

        layout = QGridLayout()
        self.setLayout(layout)

        self.cmdClose = QPushButton('Close')
        self.cmdAddSnapshot = QPushButton('Add Snapshot')
        self.cmdCreateMatrix = QPushButton('Create Matrix')

        self.tblSnapshots = Snapshot_Table(result_tables, list(global_shape_tuple))

        self.cmdClose.clicked.connect(self.cmdClose_clicked)
        self.cmdAddSnapshot.clicked.connect(self.cmdAddSnapshot_clicked)
        self.cmdCreateMatrix.clicked.connect(self.cmdCreateMatrix_clicked)

        layout.addWidget(self.tblSnapshots, 0, 0)
        layout.addWidget(self.cmdAddSnapshot, 1, 0)
        layout.addWidget(self.cmdCreateMatrix, 2, 0)
        layout.addWidget(self.cmdClose, 3, 0)

        self.AddedMatrices = []

        self.exec_()

    def cmdAddSnapshot_clicked(self):
        self.tblSnapshots.add_snapshot()

    def cmdClose_clicked(self):
        self.close()

    def cmdCreateMatrix_clicked(self):
        matrix = self.tblSnapshots.create_matrix()

        if matrix is not None and DisplayMatrix(matrix, True, 'Snapshot Matrix Created
        Successfully!').Answer == 'Add To Dataset':

                self.AddedMatrices.append(matrix)


class MainWindow(QMainWindow):

    def __init__(self, parent=None):

        super(MainWindow, self).__init__(parent)

        self.LOADED_DATASET = None
        self.Training_Test = (None, None)
        self.Labels = []
        self.ACTIVE_METAMODEL = None
        # ..................................................................
        self.setWindowTitle('METAMODEL CREATOR')
        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)
        # ..................................................................
        self.commands_window = QMdiSubWindow()
        self.mdi.addSubWindow(self.commands_window)
        top_widget_1 = QWidget()

        self.commands_window.setWindowTitle('Dataset Not Loaded')
        self.commands_window.setFixedSize(500, 300)
        self.commands_window.setWindowFlags(Qt.CustomizeWindowHint)
        self.commands_window.setWidget(top_widget_1)

        layout1 = QGridLayout(top_widget_1)
        top_widget_1.setLayout(layout1)
        # ..................................................................
        self.parameters_window = QMdiSubWindow()
        self.mdi.addSubWindow(self.parameters_window)
        top_widget_2 = QWidget()

        self.parameters_window.setWindowTitle('Input Parameters - Snapshots')
        self.parameters_window.setFixedSize(500, 1000)
        self.parameters_window.setWindowFlags(Qt.CustomizeWindowHint)
        self.parameters_window.setWidget(top_widget_2)

        self.Layout2 = QGridLayout(top_widget_2)
        top_widget_2.setLayout(self.Layout2)
        # ..................................................................
        cmdOpenDataset = QPushButton('LOAD DATASET')
        self.cmdSnapshotMatrix = QPushButton('CREATE SNAPSHOT MATRIX FROM RESULTS')
        self.cmdImportSnapshotMatrix = QPushButton('IMPORT SNAPSHOT MATRIX FROM FILE')
        self.cmdLoadSnapshotMatrix = QPushButton('SELECT EXISTING SNAPSHOT MATRIX')

        cmdOpenDataset.clicked.connect(self.cmdOpenDataset_clicked)
        self.cmdSnapshotMatrix.clicked.connect(self.cmdSnapshotMatrix_clicked)
        self.cmdImportSnapshotMatrix.clicked.connect
        (self.cmdImportSnapshotMatrix_clicked)
        self.cmdLoadSnapshotMatrix.clicked.connect(self.cmdLoadSnapshotMatrix_clicked)

        self.cmdSnapshotMatrix.setEnabled(False)
        self.cmdImportSnapshotMatrix.setEnabled(False)
        self.cmdLoadSnapshotMatrix.setEnabled(False)
        # ..............................................................................
        self.cmdTrainMM = QPushButton('Create Metamodel')
        self.cmdPlotSimple = QPushButton('Plot Selected Parameter Dependencies')
        self.cmdGSA = QPushButton('Global Sensitivity Analysis')
        self.cmdBA = QPushButton('Back Analysis')

        self.cmdTrainMM.clicked.connect(self.cmdTrainMM_clicked)
        self.cmdPlotSimple.clicked.connect(self.cmdPlotSimple_clicked)
        self.cmdGSA.clicked.connect(self.cmdGSA_clicked)
        self.cmdBA.clicked.connect(self.cmdBA_clicked)

        # ################################ ADD LAYOUTS ###########################
        layout1.addWidget(QLabel('Dataset Type'), 0, 0)
        layout1.addWidget(QLabel('Dataset Description'), 1, 0)
        layout1.addWidget(QLabel('Total Number of Simulations'), 2, 0)
        layout1.addWidget(cmdOpenDataset, 3, 0)
        layout1.addWidget(self.cmdSnapshotMatrix, 4, 0)
        layout1.addWidget(self.cmdImportSnapshotMatrix, 5, 0)
        layout1.addWidget(self.cmdLoadSnapshotMatrix, 3, 1)

        for j in range(0, 3):
            label = QLabel('***DATASET NOT LOADED***')
            self.Labels.append(label)
            layout1.addWidget(label, j, 1)
        # .............................................................................
        self.L2_Widgets = [QLabel('Input Parameters:'), None, QLabel('Snapshots:'),
         None, self.cmdPlotSimple, self.cmdTrainMM, self.cmdGSA, self.cmdBA]

        MainWindow.reset_layout(self.Layout2, self.L2_Widgets)
        self.__disable_layout2_buttons()

    @staticmethod
    def reset_layout(layout, widgets_list):

        for i in reversed(range(layout.count())):
        # Remove (detach) all widgets

            layout.itemAt(i).widget().setParent(None)

        for j, widget in enumerate(widgets_list):
        # Add (attach) new widgets

            if widget is not None:
                layout.addWidget(widget, j, 0)

    def __disable_layout2_buttons(self):
        self.cmdTrainMM.setEnabled(False)
        self.cmdPlotSimple.setEnabled(False)
        self.cmdGSA.setEnabled(False)
        self.cmdBA.setEnabled(False)

    def cmdSnapshotMatrix_clicked(self):

        result_tables = [item[0] for item in self.LOADED_DATASET.SummaryTable_3[1]]
        glob_shape = global_shape(set_shapes(result_tables))

        if all_processed(result_tables) and glob_shape is not None:
            added = Create_Snapshot_Window(result_tables, glob_shape).AddedMatrices

            if len(added) > 0 and popup_yes_no_dialog('Save changes to Dataset?',
             QMessageBox.Question, 'Confirm changes') == QMessageBox.Ok:

                self.LOADED_DATASET.add_snapshot_matrices(added)

        else:
            popup_message('Simulations are not processed or Results are not Consistent
             (different Shapes)! ', 'Check Result Tables and Repeat!',
              QMessageBox.Critical, 'Results Error', QMessageBox.Ok)

    def cmdImportSnapshotMatrix_clicked(self):

        rows = Input_Form_SHNum().Answer

        if rows is not None:

            cols = len(self.LOADED_DATASET.SummaryTable_3[1])
            filename = popup_open_save('Open', DefaultSimDir, 'Open Excel Table',
             'Excel files (*.xls *.xlsx)')

            if filename != '':
                parsed_xl = parse_xl_nd(filename, rows, cols)

                if parsed_xl is not None:

                    captions_dialog = Input_ExcelCaptions(rows, 'Input Snapshot
                     Titles', 'Title').ReturnValues

                    if captions_dialog is not None:
                        sh_captions = [caption_set[0] + ' [' + caption_set[1] + ']'
                         for caption_set in captions_dialog]

                        new_matrix = SnapshotMatrix(parsed_xl, sh_captions)

                        dlg1 = DisplayMatrix(new_matrix, False, 'Snapshot Matrix
                         Imported Successfully!').Answer

                        if dlg1 == 'Add To Dataset' and popup_yes_no_dialog('Save
                         changes to Dataset?', QMessageBox.Question,
                         'Confirm changes') == QMessageBox.Ok:
```

```
                        self.LOADED_DATASET.add_snapshot_matrices([new_matrix])
            else:
                popup_message('Error in the Excel file. Please check the data!',
                 QMessageBox.Critical, 'Data Error', QMessageBox.Ok)

    def cmdOpenDataset_clicked(self):

        def __train_test(ds_type1, mm_status1):

            if len(mm_status) < MinimumMMTrain:
            # Invalid dataset for metamodel purposes

                popup_message('Total number of simulations is smaller than minimum
                 number of metamodel training points',
                              QMessageBox.Critical, 'Invalid Dataset', QMessageBox.Ok)
                return None

            elif ds_type1 == 'Metamodel':
                return mm_status.count('Training'), mm_status.count('Test')

            else:
                return Input_Form_MM_1(len(mm_status1)).ReturnValues

        ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
        'Dataset files (*.dat)')

        if ds_filename != '':

            try:
                self.commands_window.setWindowTitle(ds_filename)

                self.LOADED_DATASET = load_object(ds_filename)
                self.LOADED_DATASET.CurrentPath = ds_filename
                self.LOADED_DATASET.update_records()

                self.cmdSnapshotMatrix.setEnabled(True)
                self.cmdImportSnapshotMatrix.setEnabled(True)
                self.cmdLoadSnapshotMatrix.setEnabled(True)

                self.ACTIVE_METAMODEL = None
                # Reset Metamodel upon dataset loading

                for lbl, value in zip(self.Labels,
                self.LOADED_DATASET.SummaryTable_3[0]):
                    lbl.setText(value)

                # --------------------------------------------------------------------
                if self.LOADED_DATASET.SummaryTable_3[2] is None:

                    popup_message('All Model Parameters are Constant!',
                     QMessageBox.Warning, 'Input Parameters', QMessageBox.Ok)

                    self.L2_Widgets[1] = None           # Reset I/O tables
                    self.L2_Widgets[3] = None

                    MainWindow.reset_layout(self.Layout2, self.L2_Widgets)
                    self.__disable_layout2_buttons()

                else:
                    mm_status = [item[1] for item in
                     self.LOADED_DATASET.SummaryTable_3[1]]

                    ds_type = self.LOADED_DATASET.SummaryTable_3[0][0]

                    TrainingTest = __train_test(ds_type, mm_status)
                    # Set training and test points

                    if TrainingTest is not None:

                        self.Training_Test = TrainingTest

                        self.L2_Widgets[1] = MMParameterTable
                        (self.LOADED_DATASET.SummaryTable_3[2][0],
                        self.LOADED_DATASET.SummaryTable_3[2][1],
                        'Input Parameters', self.Training_Test)
                        self.L2_Widgets[3] = None

                        MainWindow.reset_layout(self.Layout2, self.L2_Widgets)
                        self.__disable_layout2_buttons()

            except ImportError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

            except AttributeError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

    def cmdLoadSnapshotMatrix_clicked(self):

        if len(self.LOADED_DATASET.SnapMatrices) == 0:
            popup_message('No Snapshot Matrices in Dataset!', QMessageBox.Critical,
             'File Error', QMessageBox.Ok)

        else:
            j = ChooseSnapshot(self.LOADED_DATASET.SnapMatrices).Answer

            if j is not None:
                matrix = self.LOADED_DATASET.SnapMatrices[j]
                self.L2_Widgets[3] = MMParameterTable(matrix.Data, matrix.Captions,
                 'Snapshots', self.Training_Test)
```

```
                MainWindow.reset_layout(self.Layout2, self.L2_Widgets)
                self.cmdTrainMM.setEnabled(True)
                self.cmdPlotSimple.setEnabled(True)

    def cmdPlotSimple_clicked(self):
        x_set = self.L2_Widgets[1].PlotData
        y_set = self.L2_Widgets[3].PlotData

        if x_set is not None and y_set is not None:

            x_data, x_captions = x_set
            y_data, y_captions = y_set

            for x_values, x_caption in zip(x_data, x_captions):
                for y_values, y_caption in zip(y_data, y_captions):

                    plot1 = XY_Plot(x_values, y_values, 'b', 'o', '-', 1, None, None)

                    new_plotter = Plotter_2D('Input Parameter vs Snapshot Plot',
                     x_caption, y_caption)

                    new_plotter.plot_multiple([plot1], 'SCATTER', True, False)
        else:
            popup_message('Nothing Selected for Plotting!', QMessageBox.Warning,
             'Nothing Selected', QMessageBox.Ok)

    def cmdTrainMM_clicked(self):

        train = False

        # Check if metamodel already exists and ask for new training confirmation
        if self.ACTIVE_METAMODEL is not None:

            if popup_yes_no_dialog('Metamodel Already Active! Change?',
             QMessageBox.Question, 'Change Metamodel') == QMessageBox.Ok:
                train = True

        else:                   # MM inactive
            train = True

        if train:
            input_set = self.L2_Widgets[1].final_set()
            output_set = self.L2_Widgets[3].final_set()

            if input_set is None:
                popup_message('Check Input Parameters Table Ranges! Select at least
                 one Row!', QMessageBox.Warning, 'Input Table Error', QMessageBox.Ok)

            elif output_set is None:
                popup_message('Check Snapshots Table Ranges! Select at least one
                 Row!', QMessageBox.Warning, 'Snapshots Table Error', QMessageBox.Ok)

            else:
                created_mm = Metamodel_Training_Window(input_set,
                 output_set).Metamodel

                if created_mm is not None:
                    popup_message('Metamodel Active!', QMessageBox.Information,
                     'Metamodel Active', QMessageBox.Ok)

                    self.ACTIVE_METAMODEL = (created_mm, input_set, output_set)

                    self.L2_Widgets[1].color_sel_rows(QColor(153, 204, 255))
                    self.L2_Widgets[3].color_sel_rows(QColor(255, 178, 102))

                    self.cmdGSA.setEnabled(True)
                    self.cmdBA.setEnabled(True)

    def cmdGSA_clicked(self):

        if self.ACTIVE_METAMODEL is not None:
            GSA_Window(self.ACTIVE_METAMODEL)
        else:
            popup_message('Metamodel not Active! Train Metamodel and repeat!',
             QMessageBox.Warning, 'Metamodel not Active', QMessageBox.Ok)

    def cmdBA_clicked(self):

        if self.ACTIVE_METAMODEL is not None:

            return_vector = BA_Window(self.ACTIVE_METAMODEL).Identified

            if return_vector is not None:
                print(return_vector)

        else:
            popup_message('Metamodel not Active! Train Metamodel and repeat!',
             QMessageBox.Warning, 'Metamodel not Active', QMessageBox.Ok)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Main Classes `M3_Classes.py`

```
import numpy as np

from Main_Functions import norm_extrap_2D

from M3_Metamodels import MM_CALLER
```

```
class Metamodel:
    # Class instance can be observed to get response of the calculation progress.
    # Signal is emmited to observer each time when single calculation is made inside
     << calculation >>

    def __init__(self, m_par, data, name):

        self.Name = name

        self.Trainer = MM_CALLER[name][0]
        self.Evaluator = MM_CALLER[name][1]

        self.ModelParameters = m_par
        self.TrainingInput, self.TrainingSnapshots, self.InputRange,
         self.SnapshotRange = data

        self.TRAINING_SET = None

        self.Observers = []

        self.train()            # Start Metamodel Training

    @property
    def Training_NRMSE(self):
        return self.test(self.TrainingInput, self.TrainingSnapshots)

    @staticmethod
    def __nrmse(RealValues, ApproxValues):

        m, n = np.shape(RealValues)  # Calculate NRMSE (Normalized Root Mean Square
         Error)

        Dividend, Divisor = (0, 0)

        for i in range(0, n):
            for j in range(0, m):
                Dividend += (RealValues[j, i] - ApproxValues[j, i]) ** 2
                Divisor += RealValues[j, i] ** 2

        return np.sqrt(Dividend / Divisor)

    def train(self):

        U = norm_extrap_2D(self.TrainingSnapshots, self.SnapshotRange, 'N')
        P = norm_extrap_2D(self.TrainingInput, self.InputRange, 'N')

        self.TRAINING_SET = self.Trainer(U, P, self.ModelParameters)

    def calculate(self, input_matrix):

        if self.TRAINING_SET is not None:       # Already trained
```

```
            n = len(input_matrix[0])
            P_n = norm_extrap_2D(input_matrix, self.InputRange, 'N')
            U_n = np.zeros((len(self.SnapshotRange), n))

            for c in range(0, n):
                U_n[:, c] = self.Evaluator(P_n[:, c], self.TRAINING_SET)
                self.__update_observers()

            return norm_extrap_2D(U_n, self.SnapshotRange, 'E')

        else:
            return None

    def test(self, input_matrix, real_output):
        return Metamodel.__nrmse(real_output, self.calculate(input_matrix))

    def attach_observer(self, observer):
        self.Observers.append(observer)

    def __update_observers(self):
        for observer in self.Observers:
            observer()

    def vb_gsa_data(self):
        # Returns s, m and input ranges for VB GSA
        return len(self.TrainingInput), len(self.TrainingSnapshots), self.InputRange
```

Metamodel Algorithms `M3.py`

```
from numpy import linalg as la
import numpy as np


def __fL_fR_fB(x, g, n):

    if x <= -g:
        fL, fR = (-n * g**(n-1) * x + (1-n) * g**n, 0)

    elif x <= 0:
        fL, fR = (x**n, 0)

    elif x <= g:
        fL, fR = (0, x**n)

    else:
        fL, fR = (0, n * g**(n-1) * x + (1-n) * g**n)

    return fL, fR, x

# POD-ERBF METAMODEL ALGORITHM
#
```

```
# POD-ERBF Metamodel algorithm according to paper
"Robust and reliable metamodels for mechanized tunnel simulations" by K. Khaledi et
 al., published in Computers & Geotechnics 61 (2014)


def training_1(U, P, mm_parameters):

    m, n = np.shape(U)
    s, n = np.shape(P)

    tolerance = mm_parameters[0]        # Unpack model specific parameters
    c1 = mm_parameters[1]
    gamma = mm_parameters[2]
    n1 = mm_parameters[3]

    # --------------------------------------------------------------------------------

    U_T = np.transpose(U)

    D = np.dot(U_T, U)      # D = UT x U

    L, V = la.eig(D)
    # Eigen Values and Eigen Vectors of matrix U. Consider only real parts of eigen
     values

    L = L.real

    UV = np.dot(U, V)

    # Proper Orthogonal Decomposition -----------------------------------------------

    indices_eig_v = L.argsort()
    # Indices of sorted eigenvalues in ascending order

    k = 0
    for i in range(0, n):
        if L[i] > 0 and L[i] >= tolerance*max(L):
        # Choose POSITIVE eigen values between max and cut off min value

            k += 1
            # Count positive eigen values higher than cut off eigen value

    PHI_ = np.zeros((m, k))
    # Truncate BASIS POD Matrix - PHI1 [m x k]

    for i in range(0, k):
        j = indices_eig_v[n-1-i]
        # Pickup max eigen value index (backwards)

        # Put the highest eigen value vector to first column (and descend)
        uv = UV[:, j]
        eigen_root = float(L[j]**(-0.5))
        PHI_[:, i] = (np.multiply(uv, eigen_root)).real

    PHI__T = np.transpose(PHI_)

    A_ = np.dot(PHI__T, U)      # Equation 9, A_[k x n]

    # Radial Basis Functions  -------------------------------------------------------
    G = np.zeros((n, n))

    for r in range(0, n):
        for c in range(0, n):
            x = P[:, r]
            xi = P[:, c]
            G[r, c] = ((la.norm(x-xi))**2 + c1**2)**(-0.5)

    # H matrix and submatrices HL, HR, HB
    H = np.zeros((3*s*n, n))
    HL = np.zeros((s*n, n))
    HR = np.zeros((s*n, n))
    HB = np.zeros((s*n, n))

    for i in range(0, n):
        for j in range(0, n):
            for k1 in range(0, s):
                r = j*s+k1
                ksi = P[k1, i] - P[k1, j]      # Aprrox. vector

                # Fill submatrices
                HL[r, i], HR[r, i], HB[r, i] = __fL_fR_fB(ksi, gamma, n1)

    # -------------------------------------------------------------------------------

    # Fill matrix H using submatrices
    for i in range(0, s*n):         # Matrix HL
        H[i, :] = HL[i, :]

    for i in range(s*n, 2*s*n):     # Matrix HR
        H[i, :] = HR[i-s*n, :]

    for i in range(2*s*n, 3*s*n):   # Matrix HR
        H[i, :] = HB[i-2*s*n, :]

    # -------------------------------------------------------------------------------

    # Solve undetermined system of linear equations BG+CH = PHI_*U
    R = np.zeros((n*(3*s+1), n))

    # Fill matrix R using submatrices G and H
    for i in range(0, n):
        R[i, :] = G[i, :]

    for i in range(n, n*(3*s + 1)):
        R[i, :] = H[i-n, :]
    # Calculate vector of unknown coefficients using pseudo-inverse matrix R+
    B = np.zeros((k, n))
    C = np.zeros((k, 3*s*n))

    # Alpha Vector (Eq. 27)
    A1 = np.dot(A_, la.pinv(R))
    # multiply A with Moore-Penrose Pseudo Inverse Matrix R+

    # Separate vector of coefficients A1 into subvectors B and C
    for i in range(0, n):
        B[:, i] = A1[:, i]

    for i in range(n, (3*s+1)*n):
        C[:, i-n] = A1[:, i]

    return P, B, C, PHI_, c1, gamma, n1, tolerance


def evaluation_1(x_input, training_set):

    # Unpack metamodel input and training parameters  -------------------------------
    P, B, C, PHI_, c1, gamma, n1, tolerance = training_set
    s, n = np.shape(P)
    #  ------------------------------------------------------------------------------
    # Calculate approximation vectors g and h
    g = np.zeros((n, 1))

    h = np.zeros((3*n*s, 1))
    hL = np.zeros((n*s, 1))
    hR = np.zeros((n*s, 1))
    hB = np.zeros((n*s, 1))

    for r in range(0, n):
        xi = P[:, r]
        g[r] = ((la.norm(x_input-xi))**2 + c1**2)**(-0.5)
        # g vector calculated

        # Calculate subvectors hL, hR, hB
        for i in range(0, s):
            hL[r*s + i], hR[r*s + i], hB[r*s + i] = __fL_fR_fB(x_input[i] -
            P[i, r], gamma, n1)

    # Create full vector h
    for i in range(0, s*n):
        h[i] = hL[i]

    for i in range(s*n, 2*s*n):
        h[i] = hR[i - s*n]

    for i in range(2*s*n, 3*s*n):
        h[i] = hB[i - 2*s*n]
    return list(np.dot(PHI_, np.dot(B, g) + np.dot(C, h)))
    # Calculate response vector


INPUT_1 = [('Eigen Value Tolerance', 1e-9), ('c', 1), ('Gamma', 3), ('n', 2)]


# ERBF METAMODEL ALGORITHM


def training_2(U, P, mm_parameters):

    m, n = np.shape(U)
    s, n = np.shape(P)

    c1 = mm_parameters[0]           # Unpack model specific parameters
    gamma = mm_parameters[1]
    n1 = mm_parameters[2]

    # Radial Basis Functions --------------------------------------------------------

    G = np.zeros((n, n))
    for r in range(0, n):
        for c in range(0, n):
            x = P[:, r]
            xi = P[:, c]
            G[r, c] = ((la.norm(x-xi))**2 + c1**2)**(-0.5)

    H = np.zeros((3*s*n, n))        # H matrix and submatrices HL, HR, HB
    HL = np.zeros((s*n, n))
    HR = np.zeros((s*n, n))
    HB = np.zeros((s*n, n))

    for i in range(0, n):
        for j in range(0, n):
            for k in range(0, s):
                r = j*s+k
                xi1 = P[k, i] - P[k, j]      # Aprrox. vector

                # Fill submatrices
                HL[r, i], HR[r, i], HB[r, i] = __fL_fR_fB(xi1, gamma, n1)

    # Fill matrix H using submatrices
    for i in range(0, s*n):         # Matrix HL
        H[i, :] = HL[i, :]

    for i in range(s*n, 2*s*n):     # Matrix HR
        H[i, :] = HR[i-s*n, :]

    for i in range(2*s*n, 3*s*n):   # Matrix HR
        H[i, :] = HB[i-2*s*n, :]

    #  ------------------------------------------------------------------------------
    # Solve undetermined system of linear equations BG+CH = U
    R = np.zeros((n*(3*s+1), n))

    # Fill matrix R using submatrices G and H
    for i in range(0, n):
        R[i, :] = G[i, :]

    for i in range(n, n*(3*s + 1)):
        R[i, :] = H[i-n, :]

    # Calculate vector of unknown coefficients using pseudo-inverse matrix R+
    A1 = np.dot(U, la.pinv(R))

    # Separate vector of coefficients A1 into subvectors B and C
    B = np.zeros((m, n))
    C = np.zeros((m, 3*s*n))

    for i in range(0, n):
        B[:, i] = A1[:, i]

    for i in range(n, (3*s+1)*n):
        C[:, i-n] = A1[:, i]

    return P, B, C, c1, gamma, n1


def evaluation_2(x_input, training_set):

    # Unpack metamodel input and training parameters -------------------------------
    P, B, C, c1, gamma, n1 = training_set
    s, n = np.shape(P)
    # -------------------------------------------------------------------------------

    # Calculate approximation vectors g and h
    g = np.zeros((n, 1))
    h = np.zeros((3 * n * s, 1))

    hL = np.zeros((n * s, 1))
    hR = np.zeros((n * s, 1))
    hB = np.zeros((n * s, 1))

    for r in range(0, n):
        xi = P[:, r]
        g[r] = ((la.norm(x_input - xi)) ** 2 + c1 ** 2) ** (-0.5)
        # g vector calculated

        # Calculate subvectors hL, hR, hB
        for i in range(0, s):
            hL[r * s + i], hR[r * s + i], hB[r * s + i] = __fL_fR_fB(x_input[i]
             - P[i, r], gamma, n1)

    # Create full vector h
    for i in range(0, s * n):
        h[i] = hL[i]

    for i in range(s * n, 2 * s * n):
        h[i] = hR[i - s * n]

    for i in range(2 * s * n, 3 * s * n):
        h[i] = hB[i - 2 * s * n]

    return list(np.dot(B, g) + np.dot(C, h))          # Calculate response vector


INPUT_2 = [('c', 1), ('Gamma', 3), ('n', 2)]


# ##########################################################################################

MM_CALLER = {'POD-ERBF': (training_1, evaluation_1), 'ERBF': (training_2, evaluation_2)}
MM_CALLER_2 = {'POD-ERBF': INPUT_1, 'ERBF': INPUT_2}
MM_LIST = ['POD-ERBF', 'ERBF']
```

Back Analysis Algorithms `M3_BackAnalysis.py`

```
from pyswarm import pso
import numpy as np

from PyQt5.QtWidgets import QMessageBox

from Main_Functions import vector_nd
from Main_ConfigConstants import DefaultSimDir
from Main_Functions import export_nd_xl, popup_open_save, popup_message
from M3_Plotting import XY_Plot, Plotter_2D


class PSO:

    def __init__(self, metamodel, input_parameters, search_i, search_o,
     input_captions, snap_captions):

        s = len(search_i[0])
        swarm, omega, c1, c2, iterlimit, minf, minstep = input_parameters

        self.Observers = []

        self.Metamodel = metamodel
        self.Metamodel.Observers = []

        self.InputData = input_parameters
        self.SearchDataInput = search_i
        self.SearchDataOutput = search_o
        self.InputCaptions = input_captions
        self.SnapCaptions = snap_captions

        self.Swarm = int(swarm)
        self.MaxIterations = int((iterlimit+1))
        self.SwarmCount = int(swarm)

        self.Step = 0
        self.ResultsTable = np.zeros((s+1, self.Swarm*self.MaxIterations))
        self.ResultsTable2 = None
        self.ResultsTable3 = None

    def calculate(self):

        # Decompose input data
        swarm, omega, c1, c2, iterlimit, minf, minstep = self.InputData

        Weights, U_measured = self.SearchDataOutput

        lower_bounds, upper_bounds = self.SearchDataInput

        objf_arguments = (U_measured, Weights)      # Input for objective function

        xopt, fopt = pso(self.objf, lower_bounds, upper_bounds, args=objf_arguments,
         swarmsize=int(swarm), omega=omega, phip=c1, phig=c2, maxiter=int(iterlimit),
          minfunc=minf, minstep=minstep)

        self.__process_results()
        # Process raw results table after calculation

        return xopt, fopt

    def objf(self, x, *args):

        u_measured, weights = args
        u_calculated = self.Metamodel.calculate(vector_nd(x))
        # Calculate using metamodel

        ofv = 0
        # Calculate objective function

        for j in range(0, len(u_calculated)):
            ofv += weights[j] * (1 - u_calculated[j] / u_measured[j]) ** 2

        self.update_results_and_observers(x, ofv)

        return ofv


    def update_results_and_observers(self, x, ofv):

        s = len(x)
        self.ResultsTable[0:s, self.Step] = list(x)
        self.ResultsTable[s, self.Step] = ofv
        self.Step += 1

        # Update Swarm counter and observers
        self.SwarmCount -= 1
        if self.SwarmCount == 0:
        # Swarm empty - Iteration Ended. Restart counter and update observers

            self.SwarmCount = int(self.Swarm)
            self.update_observers()

    def attach_observer(self, observer):
        self.Observers.append(observer)

    def update_observers(self):
        for observer in self.Observers:
            observer()

    def plot_results(self):
        self.__plot_results_1()
        self.__plot_results_2()

    def export_results(self):
        data_to_export = self.ResultsTable[:, 0:self.Step].transpose()
        filename = popup_open_save('Save', DefaultSimDir, 'Save As',
        'Excel files (*.xlsx)')

        if filename != '':
            wb = export_nd_xl(['Particle_' + str(i) for i in range(0, self.Step)],
                              self.InputCaptions+['Objective Function Value'],
                              ['PSO Results'], data_to_export)
            try:
                wb.save(filename)
                popup_message('Results Exported Successfully!',
                 QMessageBox.Information, 'File Error', QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

    def __process_results(self):

        rows, max_particles = np.shape(self.ResultsTable)
        s = rows - 1
        full_iterations = self.Step // self.Swarm

        if max_particles - full_iterations * self.Swarm > 0:
            full_iterations += 1

        self.ResultsTable2 = np.zeros((s + 1, self.Swarm, full_iterations))
        self.ResultsTable3 = np.zeros((s + 1, 2, full_iterations))
        for k in range(0, full_iterations):
            self.ResultsTable2[:, :, k] = self.ResultsTable[:, k * self.Swarm:(k + 1)
             * self.Swarm]

        for k in range(0, full_iterations):
            of_vector = self.ResultsTable2[s, :, k]
            self.ResultsTable3[:, 0, k] = self.ResultsTable2[:, np.argmin(of_vector),
             k]

            self.ResultsTable3[:, 1, k] = self.ResultsTable2[:, np.argmax(of_vector),
             k]

    def __plot_results_1(self):
        rows, columns, done_iterations = np.shape(self.ResultsTable3)

        X = np.arange(1, done_iterations+1)
        Y1 = self.ResultsTable3[rows - 1, 1, :]         # Global Max of OF
        Y2 = self.ResultsTable3[rows - 1, 0, :]         # Global Min of OF
        minY = np.min(Y2)
        maxY = np.max(Y1)

        plotter_1 = Plotter_2D('OBJECTIVE FUNCTION GLOBAL MIN/MAX', 'Iteration',
         'Objective function values', x_limit=(1, done_iterations), y_limit=(minY,
          maxY))

        plotter_1.plot_start(scale_log=['y'])
        plotter_1.add_plot(XY_Plot(X, Y1, 'b', '.', '--', 1, 'Maximum', None), True,
         False)

        plotter_1.add_plot(XY_Plot(X, Y2, 'r', '.', '--', 1, 'Minimum', None), True,
         False)

        plotter_1.finalize()

    def __plot_results_2(self):
        rows, swarm, done_iterations = np.shape(self.ResultsTable2)

        X1 = np.arange(1, done_iterations + 1)

        for par in range(0, rows - 1):
            label = self.InputCaptions[par]

            # Calculate Ymin,max
            Ymin = np.zeros(done_iterations)
            Ymax = np.zeros(done_iterations)

            for i in range(0, done_iterations):
                Ymin[i] = np.min(self.ResultsTable2[par, :, i])
                Ymax[i] = np.max(self.ResultsTable2[par, :, i])

            plot1 = XY_Plot(X1, Ymin, 'b', '.', '--', 1, 'Minimum', None)
            plot2 = XY_Plot(X1, Ymax, 'r', '.', '--', 1, 'Maximum', None)
```

```
            plotter_2 = Plotter_2D('SEARCH FOR PARAMETER: ' + label, 'Iteration',
             label, x_limit=(1, done_iterations))

            plotter_2.plot_multiple([plot1, plot2], 'PLOT', True, False)


INPUT_1 = [('Swarm Size', 20),
           ('Inertia Weight', 0.8),
           ('Cognitive parameter (c1)', 0.5),
           ('Social parameter (c2)', 1.25),
           ('Max Iterations', 100),
           ('Min Objective Function Value', 1e-30),
           ('Min Objective Function Change', 1e-27)]

BA_LIST = ['Particle Swarm Optimization']

BA_CALLER = {'Particle Swarm Optimization': (PSO, INPUT_1)}
```

Global Sensitivity Analysis Algorithms `M3_GSA.py`

```
from pyDOE import lhs
import numpy as np

from PyQt5.QtGui import QColor
from PyQt5.QtWidgets import QTableWidget, QAbstractScrollArea, QDialog, QGridLayout,
 QMessageBox

from Main_WidgetClasses import StringCell
from Main_Functions import norm_extrap_2D, export_nd_xl, popup_open_save,
 popup_message

from Main_ConfigConstants import DefaultSimDir

from M3_Observers import MM_BA_Observer

# -------------------------------------------------------------------------------------
# VARIANCE BASED GLOBAL SENSITIVITY ANALYSIS ALGORITHM

# This algorithm is designed to be used with previously created metamodel, since it
 uses a lot of forward calculations

# Some of input data are unpacked from metamodel


class VB_GSA:

    def __init__(self, input_parameters, metamodel):

        # Prepare metamodel. Attach observer to monitor calculation progress
        self.Metamodel = metamodel
        self.Metamodel.Observers = []
        self.Metamodel.attach_observer(MM_BA_Observer(self.update_observers))

        s, m, input_ranges = self.Metamodel.vb_gsa_data()

        n = int(input_parameters[0][0])
        # Ensure that n is integer

        self.InputCaptions = input_parameters[1]
        self.SnapCaptions = input_parameters[2]

        self.TotalIterations = n * (s + 2)
        self.InputData = s, m, n, input_ranges

        self.Results = None
        self.Observers = []

    def calculate(self):

        # INPUT DATA - unpack
        s, m, n, input_ranges = self.InputData
        metamodel = self.Metamodel

        # VARIANCE BASED SENSITIVITY ANALYSIS ALGORITHM #############################

        # Generate random samples using Latin Hypercube Sampling - creates input
         samples [0, 1] - {s x n}

        A_01_T = lhs(n, samples=s)
        B_01_T = lhs(n, samples=s)

        # Extrapolate input matrices using input ranges
        A_T = norm_extrap_2D(A_01_T, input_ranges, 'E', limits=(0, 1))
        B_T = norm_extrap_2D(B_01_T, input_ranges, 'E', limits=(0, 1))

        A = np.transpose(A_T)
        B = np.transpose(B_T)  # Sample matrices A, B {n x s} created

        C = np.zeros((n, s, s))

        for j in range(0, s):  # Fill matrix C
            for i in range(0, s):
                if i == j:
                    C[:, i, j] = A[:, i]
                else:
                    C[:, i, j] = B[:, i]

        # Calculate responses for input matrices A, B, C using attached metamodel.

        # Metamodel calculates matrices shaped as {s x n}, so we provide transposed
         matrices A and B for calculation

        YA_T = metamodel.calculate(A_T)
        # Metamodel returns matrices shaped as {m x n}
```

```
YB_T = metamodel.calculate(B_T)

# Transpose output matrices to shape {n x m}
YA = np.transpose(YA_T)
YB = np.transpose(YB_T)

YC = np.zeros((n, m, s))

for ci in range(0, s):
    input_matrix = C[:, :, ci]  # Input matrix {n x s}

    # Provide matrix shaped as {s x n} to metamodel for calculation

    mm_output = metamodel.calculate(np.transpose(input_matrix))
    # Metamodel returns table shaped as {m x n}

    YC[:, :, ci] = np.transpose(mm_output)
    # Add metamodel output shaped as {n x m} to YC ndarray

# Evaluate Sobol Indices ....................................................
Si = np.zeros((s, m))
STi = np.zeros((s, m))
Sum_Si = []
Sum_STi = []

for k in range(0, m):

    yA = YA[:, k]
    yB = YB[:, k]

    yA_T = np.transpose(yA)
    yB_T = np.transpose(yB)

    yA_m = np.mean(yA)
    yB_m = np.mean(yB)

    divider_Si = (np.dot(yA_T, yA) - n * yA_m ** 2)
    # Divisors for Sobol sensitivity indices

    divider_STi = 2 * (np.dot(yB_T, yB) - n * yB_m ** 2)

    for i in range(0, s):
        yCi = YC[:, k, i]

        Si[i, k] = (np.dot(yA_T, yCi) - n * yA_m ** 2) / divider_Si
        STi[i, k] = (np.dot(np.transpose(yB - yCi), yB - yCi)) / divider_STi

    Sum_Si.append(np.sum(Si[:, k]))
    Sum_STi.append(np.sum(STi[:, k]))

# ##############################################################################
```

```
        self.Results = Si, STi, Sum_Si, Sum_STi

        return Si, STi, Sum_Si, Sum_STi

    def attach_observer(self, observer):
        self.Observers.append(observer)

    def update_observers(self):
        for observer in self.Observers:
            observer()

    def show_results(self):
        result_table = VB_Results_Table(self.Results)
        GSA_Results(result_table)

    def plot_results(self):
        this_plot.plot_bar(self.Results[0], 'Snapshots', 'Si [-]', self.InputCaptions,
         self.SnapCaptions, 'First Order Sensitivity Indices')

        this_plot.plot_bar(self.Results[1], 'Snapshots', 'STi [-]',
         self.InputCaptions, self.SnapCaptions, 'Total Effect Indices')

        this_plot.finalize()

    def export_results(self):

        # Prepare Data for export
        s, m = np.shape(self.Results[0])

        data_1 = np.zeros((s, m, 2))
        data_1[:, :, 0] = self.Results[0]
        data_1[:, :, 1] = self.Results[1]

        filename = popup_open_save('Save', DefaultSimDir, 'Save As',
        'Excel files (*.xlsx)')

        if filename != '':
            wb = export_nd_xl(self.InputCaptions, self.SnapCaptions, ['Si', 'STi'],
             data_1)

            try:
                wb.save(filename)
                popup_message('Results Exported Successfully!',
                 QMessageBox.Information, 'File Error', QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)


class VB_Results_Table(QTableWidget):

    def __init__(self, results):
        super().__init__()

        Si, STi, Sum_Si, Sum_STi = results

        s, m = np.shape(Si)

        rows = 2 * s + 5
        columns = m

        self.setRowCount(rows)
        self.setColumnCount(columns)

        vertical_headers = ['S' + str(i) for i in range(0, s)] + \
                           ['', 'Sum Si', ''] + \
                           ['ST' + str(i) for i in range(0, s)] + ['', 'Sum STi']

        horizontal_headers = ['Snapshot_' + str(i) for i in range(0, m)]

        self.setHorizontalHeaderLabels(horizontal_headers)
        self.setVerticalHeaderLabels(vertical_headers)
        self.horizontalHeader().setVisible(True)
        self.verticalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        # Clear Contents
        for r in range(0, rows):
            for c in range(0, columns):
                self.setItem(r, c, StringCell(''))

        # Fill Data
        for c in range(0, m):
            for r in range(0, s):
                self.setItem(r, c, StringCell(str(round(Si[r, c], 4))))

            for r in range(s + 3, 2 * s + 3):
                self.setItem(r, c, StringCell(str(round(STi[r - 3 - s, c], 4))))

            self.setItem(s + 1, c, StringCell(str(round(Sum_Si[c], 4))))
            self.setItem(2 * s + 4, c, StringCell(str(round(Sum_STi[c], 4))))

        self.color_wrong_values(self.check_VB_GSA(results), s, m)

        self.clearSelection()
        self.resizeColumnsToContents()

    @staticmethod
    def check_VB_GSA(results1):

        Si, STi, Sum_Si, Sum_STi = results1
        s, m = np.shape(Si)

        Sum_Si_Wrong = [i for i, item in enumerate(Sum_Si) if item > 1]
        # Case 1: Sum Si <= 1
        Sum_STi_Wrong = [i for i, item in enumerate(Sum_STi) if item < 1]
        # Case 2: Sum STi >= 1

        STi_Wrong = []
        Si_Wrong = []

        for r in range(0, s):
            for c in range(0, m):
                if Si[r, c] < 0 or Si[r, c] > 1:
                # Case 3: 0 =< Si =< 1

                    Si_Wrong.append((r, c))

        for r in range(0, s):
            for c in range(0, m):
                if Si[r, c] > STi[r, c]:
                # Case 4: STi > Si
                    STi_Wrong.append((r, c))
                    Si_Wrong.append((r, c))

        return Si_Wrong, STi_Wrong, Sum_Si_Wrong, Sum_STi_Wrong

    def paint(self, row, col):
        self.item(row, col).setForeground(QColor(255, 0, 0))

    def color_wrong_values(self, wrong_value_list, s, m):
        rows = 2 * s + 5
        columns = m

        Si_Wrong, STi_Wrong, Sum_Si_Wrong, Sum_STi_Wrong = wrong_value_list

        for r in range(0, rows):
            for c in range(0, columns):
                self.item(r, c).setForeground(QColor(0, 0, 0))

        for c in range(0, m):
            for r in range(0, s):
                if (r, c) in Si_Wrong:
                    self.paint(r, c)

            for r in range(s + 3, 2 * s + 3):
                if (r-3-s, c) in STi_Wrong:
                    self.paint(r, c)

            if c in Sum_Si_Wrong:
                self.paint(s+1, c)

            if c in Sum_STi_Wrong:
                self.paint(2*s+4, c)


class GSA_Results(QDialog):
    def __init__(self, widget):
        super().__init__()

        self.setWindowTitle('GSA Results')
        self.setFixedSize(500, 500)

        top_layout = QGridLayout()
        self.setLayout(top_layout)
        top_layout.addWidget(widget)

        self.exec_()


INPUT_1 = [('Number of Sample Points', 100)]

# ####################################################################################

GSA_CALLER = {'Variance Based': (VB_GSA, INPUT_1)}
GSA_LIST = ['Variance Based']
```

Visualization Procedures M3_Plotting.py

```
import matplotlib.pyplot as plt
import numpy as np

from Main_ConfigConstants import Colors, grid_yn, LegendFont, LegendPos


def color_picker(x):
    # Cycle colors by index

    total_colors = len(Colors)

    if x <= total_colors-1:
        return Colors[x]

    else:
        return Colors[x-(x // total_colors)*total_colors]


class XY_Plot:

    def __init__(self, x_data, y_data, color, marker, line_type, line_width,
     legend_label, normalizer):

        self.X = x_data
        self.Y = y_data

        self.LineColor = color
        self.LineMarker = marker
        self.LineType = line_type
        self.LineLabel = legend_label
        self.LineWidth = line_width

        self.Normalizer = normalizer


class Plotter_2D:

    Property_List = ['XLabel', 'YLabel', 'Title', 'XLimit', 'YLimit']
    Callers = ['xlabel', 'ylabel', 'title', 'xlim', 'ylim']

    def __init__(self, title, x_label, y_label, x_limit=None, y_limit=None):

        self.Title = title

        self.XLabel = x_label
        self.YLabel = y_label

        self.XLimit = x_limit
        self.YLimit = y_limit

        self.Plots = []

    def plot_start(self, scale_log=None):

        plt.figure()

        for attribute_name, caller_name in zip(Plotter_2D.Property_List,
         Plotter_2D.Callers):

            plt_attibute = getattr(plt, caller_name)
            attribute_value = getattr(self, attribute_name)

            if attribute_value is not None:
                plt_attibute(getattr(self, attribute_name))

        if grid_yn is not None:
            plt.grid(grid_yn)

        if scale_log is not None:
            for axis in scale_log:
                caller = getattr(plt, axis+'scale')
                caller('log')

    def plot_xy(self, plot_type, show_l=True, normalize=False, index=None):

        if index is None:
            plot_now = self.Plots[-1]       # Plot last added plot

        else:
            plot_now = self.Plots[index]    # Plot required plot
        x_data = plot_now.X

        if normalize:
            x_data = [x/plot_now.Normalizer for x in x_data]

        if plot_type == 'PLOT':

            plt.plot(x_data, plot_now.Y,
                     label=plot_now.LineLabel,
                     linestyle=plot_now.LineType,
                     linewidth=plot_now.LineWidth,
                     color=plot_now.LineColor,
                     marker=plot_now.LineMarker, )

        elif plot_type == 'SCATTER':

            plt.scatter(x_data, plot_now.Y,
                        label=plot_now.LineLabel,
                        color=plot_now.LineColor,
                        marker=plot_now.LineMarker)

        if show_l:
            plt.legend(loc=LegendPos, fontsize=LegendFont)

    def add_plot(self, plot, show_legend, normalize, plot_type='PLOT'):
        self.Plots.append(plot)
        self.plot_xy(plot_type, show_l=show_legend, normalize=normalize)

    @staticmethod
    def finalize():
        plt.show()

    def plot_multiple(self, plot_list, plot_type, show_legend, normalize):

        self.plot_start()

        for plot in plot_list:
            self.add_plot(plot, show_legend, normalize, plot_type=plot_type)

        plt.show()

    @staticmethod
    def plot_bar(data_table, bar_labels, group_labels, show_l=True):

        s, m = np.shape(data_table)
        index = np.arange(m)
        width = 1 / (s + 1)

        fig, ax = plt.subplots()

        for j in range(0, s):
            ax.bar(index + j * width, data_table[j, :], width, label=bar_labels[j],
             color=color_picker(j))
        plt.xticks(index + (width + 1) / 2, group_labels)
        plt.tight_layout()

        if show_l:
            plt.legend(loc=LegendPos, fontsize=LegendFont)
```

Observer Classes `M3_Observers.py`

```
from PyQt5.QtCore import QObject, QRunnable, pyqtSignal, pyqtSlot

class MM_BA_Observer:

    def __init__(self, receiver):
        self.Receiver = receiver
        # Function that will receive signal from observer and react (GUI function)

    def __call__(self):
        self.Receiver()


class BA_GSA_Worker_Signals(QObject):
    finished = pyqtSignal(tuple)


class BA_GSA_Worker(QRunnable):

    def __init__(self, input_object):
        super(BA_GSA_Worker, self).__init__()
        self.Signals = BA_GSA_Worker_Signals()
        self.Object = input_object

    @pyqtSlot()
    def run(self):

        results = self.Object.calculate()

        if results is not None:
            self.Signals.finished.emit(results)
```

User interface Classes `M3_WidgetClasses.py`

```
import numpy as np
import math

from PyQt5.QtGui import QColor
from PyQt5.QtWidgets import QComboBox, QMessageBox, QLineEdit, QTableWidget,
 QAbstractScrollArea, QTableWidgetItem
```

```
from PyQt5.QtWidgets import QCheckBox, QProgressBar, QDialog, QGridLayout

from Main_Functions import popup_message, np_slicer, remove_duplicates

from Main_WidgetClasses import StringCell, fill_table_items
from MyClasses1 import SnapshotMatrix


class Snapshot_Table(QTableWidget):

    def __init__(self, result_tables, global_shape1):

        super().__init__(1, len(global_shape1)+3)

        self.__ResultTables = result_tables
        self.__Snapshots = []
        self.GlobalShape = list(global_shape1)

        table_captions = ['Snapshot', 'Unit'] + ['Pos_' + str(i)
                                                 for i in range(0,
                                                  len(global_shape1))] +
                                                  ['Include in Matrix']

        self.setHorizontalHeaderLabels(table_captions)
        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        self.add_widgets_to_row(0)
        # Create first, initial snapshot row

        self.clearSelection()
        self.resizeColumnsToContents()

    @property
    def Selected_Snapshots(self):
        indices = [j for j, snapshot_set in enumerate(self.__Snapshots)
        if snapshot_set[-1].isChecked()]

        return [self.__Snapshots[j] for j in indices]

    def add_snapshot(self):
        self.insertRow(self.rowCount())
        self.add_widgets_to_row(self.rowCount()-1)

    def __has_duplicates(self):
        getter_vectors = [[int(combo.currentText())
        for combo in widgets[2:2+len(self.GlobalShape)]]

                          for widgets in self.Selected_Snapshots]

        return len(getter_vectors) > len(remove_duplicates(getter_vectors))

    def __titles_filled(self):
        for widgets in self.Selected_Snapshots:
            if widgets[0].text() == '' or widgets[1].text() == '':
                return False
        return True

    def add_widgets_to_row(self, row_index):

        Snapshot_Widgets = [QLineEdit('Title'), QLineEdit('Unit')]

        for c in range(0, len(self.GlobalShape)):       # Create Combo Boxes
            widget = QComboBox()
            widget.addItems([str(j) for j in range(0, self.GlobalShape[c])])
            Snapshot_Widgets.append(widget)

        Snapshot_Widgets.append(QCheckBox())
        # Snapshot widgets list created

        self.__Snapshots.append(Snapshot_Widgets)

        for j, widget in enumerate(Snapshot_Widgets):
        # Add widgets to table row

            self.setCellWidget(row_index, j, widget)

    def create_matrix(self):

        snapshot_matrix = None
        snapshot_captions = None

        if len(self.Selected_Snapshots) == 0:
            popup_message('Nothing selected! Select at least one snapshot!',
             QMessageBox.Warning, 'Error', QMessageBox.Ok)

        else:
            if self.__has_duplicates():
                popup_message('Duplicated Snapshots found! Remove duplicates from
                 Combo Boxes!', QMessageBox.Warning, 'Error', QMessageBox.Ok)

            else:
                if not self.__titles_filled():
                    popup_message('Please fill Name and Unit for each used Snapshot',
                     QMessageBox.Warning, 'Error', QMessageBox.Ok)
                else:
                    # All checkpoints passed. Now create snapshot matrix and titles

                    snapshot_matrix = np.zeros((len(self.Selected_Snapshots),
                     len(self.__ResultTables)))

                    snapshot_captions = []

                    for k, widgets_set in enumerate(self.Selected_Snapshots):
                        snapshot_captions.append(widgets_set[0].text()
                        + ' [' + widgets_set[1].text() + ']')
                    get_vector = [int(combo.currentText())
                    for combo in widgets_set[2:2+len(self.GlobalShape)]]

                    snapshot_matrix[k, :] = [table.item(tuple(get_vector))
                    for table in self.__ResultTables]

        if snapshot_matrix is not None:
            return SnapshotMatrix(snapshot_matrix, snapshot_captions)

        else:
            return None


class BA_Parameters(QTableWidget):

    def __init__(self, parameter_names, ranges, i_o):

        super().__init__(len(parameter_names), 6)

        self.Rows = len(parameter_names)
        self.Ranges = ranges

        if i_o == 'Input':

            headers = ['Input Parameter', 'Min', 'Max', 'Search Min', 'Search Max',
             'Found Value']

            col_3 = self.Ranges[:, 0]
            col_4 = self.Ranges[:, 1]
            col_5 = np.zeros(self.Rows)

        else:

            headers = ['Snapshot', 'Min', 'Max', 'Weight', 'Search for',
            'Identified using Metamodel']

            col_3, col_4, col_5 = ([], [], [])

            for i in range(0, self.Rows):
                col_3.append(1)
                col_4.append(self.Ranges[i, 0])
                col_5.append('')

        self.setHorizontalHeaderLabels(headers)
        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        for r in range(0, self.Rows):
            self.setItem(r, 0, StringCell(parameter_names[r]))
            self.setItem(r, 1, StringCell(str(round(self.Ranges[r, 0], 4))))
            self.setItem(r, 2, StringCell(str(round(self.Ranges[r, 1], 4))))
            self.setItem(r, 3, QTableWidgetItem(str(round(col_3[r], 4))))
            # Editable cell #

            self.setItem(r, 4, QTableWidgetItem(str(round(col_4[r], 4))))
            # Editable cell #

            self.setItem(r, 5, StringCell(str(col_5[r])))

        self.clearSelection()
        self.resizeColumnsToContents()

    def analysis_data(self):

        try:
            vector_1 = [float(self.item(i, 3).text()) for i in range(0, self.Rows)]
            vector_2 = [float(self.item(i, 4).text()) for i in range(0, self.Rows)]

            return vector_1, vector_2

        except ValueError:
        # Catch input error during conversion from string to float

            return None

    def update_column(self, col_num, vector):
        for r in range(0, self.Rows):
            self.setItem(r, col_num, StringCell(str(round(vector[r], 4))))


class MMParameterTable(QTableWidget):

    def __init__(self, table_data, parameter_names, title, tt):

        super().__init__(len(parameter_names), 5)

        training_pts = tt[0]
        test_pts = tt[1]

        self.Captions = parameter_names
        self.FullTable = table_data
        self.Minimums = [np.min(data_row) for data_row in self.FullTable]
        # Calculate global maximums and minimums

        self.Maximums = [np.max(data_row) for data_row in self.FullTable]

        # Setup Training and Test Matrix. If no test points, Test Matrix is None

        self.TrainingMatrix = np_slicer('C', list(range(0, training_pts)),
         self.FullTable)

        if test_pts > 0:
            self.TestMatrix = np_slicer('C', list(range(training_pts,
            training_pts+test_pts)), self.FullTable)
        else:
            self.TestMatrix = None

        self.setHorizontalHeaderLabels([title, 'Min', 'Max', 'Custom Min',
        'Custom Max'])

        self.horizontalHeader().setVisible(True)
        self.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        # Fill the data
        for r in range(0, len(self.FullTable)):
            self.setItem(r, 0, StringCell(parameter_names[r]))
            self.setItem(r, 1, StringCell(str(round(self.Minimums[r], 4))))
            self.setItem(r, 2, StringCell(str(round(self.Maximums[r], 4))))
            self.setItem(r, 3, QTableWidgetItem(str(round(self.Minimums[r], 4))))
             # Editable cell #

            self.setItem(r, 4, QTableWidgetItem(str(round(self.Maximums[r], 4))))
             # Editable cell #

        self.clearSelection()
        self.resizeColumnsToContents()

    @property
    def __SelectedRows(self):
        return sorted(set(i.row() for i in self.selectedIndexes()))

    def setup_ranges(self):

        ranges_matrix = np.zeros((len(self.__SelectedRows), 2))
        # Initialize Ranges Matrix

        for j, r in enumerate(self.__SelectedRows):

            try:
                ranges_matrix[j, 0] = min(self.Minimums[r],
                float(self.item(r, 3).text()))
                # Smaller minimum adopted

                ranges_matrix[j, 1] = max(self.Maximums[r],
                float(self.item(r, 4).text()))
                # Higher maximum adopted

            except ValueError:
            # Catch input error during conversion from string to float

                return None

        return ranges_matrix

    def final_set(self):

        """
            Creates universal metamodel input set - Training/Test matrix and
            Parameter Ranges + Titles
            If not a single row is selected, function returns None
            Full Training and Test matrices are sliced using .__SelectedRows list

            Return set: Training Matrix, Test Matrix, Ranges, Captions
        """

        if len(self.__SelectedRows) > 0:
            ranges_matrix = self.setup_ranges()

            if ranges_matrix is not None:

                training_matrix = np_slicer('R', self.__SelectedRows,
                 self.TrainingMatrix)
                 # Filter Training matrix
                test_matrix = None

                if self.TestMatrix is not None:
                    test_matrix = np_slicer('R', self.__SelectedRows,
                    self.TestMatrix)
                    # Filter Test matrix

                return training_matrix, test_matrix, ranges_matrix,
                [self.Captions[i] for i in self.__SelectedRows]

            else:
                return None
        else:
            return None

    @property
    def PlotData(self):
        if len(self.__SelectedRows) > 0:
            return np_slicer('R', self.__SelectedRows, self.FullTable),
             [self.Captions[i] for i in self.__SelectedRows]

        else:
            return None

    def color_sel_rows(self, color):

        for r in range(0, len(self.Minimums)):

            if r in self.__SelectedRows:
                c1 = color                      # Shade color
            else:
                c1 = QColor(255, 255, 255)      # White

            for c in range(0, 5):
                self.item(r, c).setBackground(c1)
```

```
class PB(QProgressBar):

    def __init__(self):
        super().__init__()
        self.Total = None
        self.Remaining = None

    def initialize(self, total):
        self.Total = int(total)
        self.Remaining = int(total)
        self.setValue(0)

    def update(self):
        self.Remaining -= 1
        remain_perc = float(self.Remaining/self.Total)
        new_value = math.floor((1-remain_perc)*100)

        self.setValue(new_value)

    def end(self):
        self.setValue(100)


class Simple_Table(QDialog):

    def __init__(self, matrix):

        super().__init__()

        self.setWindowTitle('Snapshot Matrix')
        self.setFixedSize(500, 500)

        top_layout = QGridLayout()
        self.setLayout(top_layout)

        table_widget = QTableWidget(len(matrix.Data), len(matrix.Data[0]))

        table_widget.setVerticalHeaderLabels(matrix.Captions)
        table_widget.verticalHeader().setVisible(True)
        table_widget.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

        fill_table_items(table_widget, matrix.Data)

        table_widget.resizeColumnsToContents()
        table_widget.clearSelection()

        top_layout.addWidget(table_widget)

        self.exec_()
```

Module 4 - Multiple Dataset Postprocessor `M4.py`

```
import sys

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QMainWindow, QApplication, QMdiArea, QMdiSubWindow,
 QMessageBox, QLabel, QGridLayout

from PyQt5.QtWidgets import QWidget, QPushButton

from Main_Functions import popup_open_save, popup_message
from Main_ConfigConstants import DefaultSimDir

from Main_WidgetClasses import load_dataset_sim_table


class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

        self.LOADED_DATASET = None
        self.LOADED_DATASET2 = None
        self.SimsTable = None
        self.SimsTable2 = None

        # WINDOWS ##################################################################
        self.setWindowTitle('MERGE DATASETS')
        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)         # Sub Windowed GUI

        top_widget_1 = QWidget()
        top_widget_2 = QWidget()
        top_widget_3 = QWidget()

        layout1 = QGridLayout(top_widget_1)
        self.Layout2 = QGridLayout(top_widget_2)
        self.Layout3 = QGridLayout(top_widget_3)

        top_widget_1.setLayout(layout1)
        top_widget_2.setLayout(self.Layout2)
        top_widget_3.setLayout(self.Layout3)

        self.commands_window = QMdiSubWindow()
        sim_window = QMdiSubWindow()
        sim_window2 = QMdiSubWindow()

        self.mdi.addSubWindow(self.commands_window)
        self.mdi.addSubWindow(sim_window)
        self.mdi.addSubWindow(sim_window2)

        self.commands_window.setWindowTitle('Dataset Not Loaded')
        sim_window.setWindowTitle('Parent Dataset Simulations')
        sim_window2.setWindowTitle('Child Dataset Simulations')

        self.commands_window.setFixedSize(400, 200)
        sim_window.setFixedSize(1000, 500)
        sim_window2.setFixedSize(1000, 500)

        self.commands_window.setWindowFlags(Qt.CustomizeWindowHint)
        sim_window.setWindowFlags(Qt.CustomizeWindowHint)
        sim_window2.setWindowFlags(Qt.CustomizeWindowHint)

        self.commands_window.setWidget(top_widget_1)
        sim_window.setWidget(top_widget_2)
        sim_window2.setWidget(top_widget_3)

        # WIDGETS ##################################################################

        cmdOpenDataset = QPushButton('LOAD PARENT DATASET')
        cmdOpenDataset2 = QPushButton('LOAD CHILD DATASET')
        self.cmdUpdateDS = QPushButton('UPDATE PARENT DATASET')

        self.cmdUpdateDS.setEnabled(False)

        cmdOpenDataset.clicked.connect(self.cmdOpenDataset_clicked)
        cmdOpenDataset2.clicked.connect(self.cmdOpenDataset2_clicked)
        self.cmdUpdateDS.clicked.connect(self.cmdUpdate_clicked)

        layout1.addWidget(QLabel('Dataset Type'), 0, 0)
        layout1.addWidget(QLabel('Dataset Description'), 1, 0)
        layout1.addWidget(QLabel('Modelling Sequence'), 2, 0)
        layout1.addWidget(cmdOpenDataset, 3, 0)
        layout1.addWidget(cmdOpenDataset2, 3, 1)
        layout1.addWidget(self.cmdUpdateDS, 4, 0)

        self.Labels = []
        for j in range(0, 3):                              # Fill label values
            label = QLabel('***DATASET NOT LOADED***')
            self.Labels.append(label)
            layout1.addWidget(label, j, 1)

        # Show sub windows  --------------------------------------------------------
        self.commands_window.show()
        sim_window.show()
        sim_window2.show()

    def cmdOpenDataset_clicked(self):

        ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
        'Dataset files (*.dat)')

        if ds_filename != '':
            try:
                self.LOADED_DATASET, self.SimsTable =
                 load_dataset_sim_table(self.Layout2, ds_filename)

                if self.LOADED_DATASET2 is not None:
                    self.cmdUpdateDS.setEnabled(True)

                self.commands_window.setWindowTitle(ds_filename)

                for lbl, value in zip(self.Labels,
                 self.LOADED_DATASET.SummaryTable_2[0]):

                    lbl.setText(value)

            except ImportError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

            except AttributeError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

    def cmdOpenDataset2_clicked(self):

        ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
        'Dataset files (*.dat)')

        if ds_filename != '':
            try:
                self.LOADED_DATASET2, self.SimsTable2 =
                 load_dataset_sim_table(self.Layout3, ds_filename)

                if self.LOADED_DATASET is not None:
                    self.cmdUpdateDS.setEnabled(True)

            except ImportError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

            except AttributeError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

    def cmdUpdate_clicked(self):

        child_selected = self.SimsTable2.SelectedRows
        if len(child_selected) > 0:
        # Something is Selected

            if max(child_selected) > self.SimsTable.rowCount()-1:
            # Last child simulation can’t be added to parent

                popup_message(’Child dataset size does not match the parent dataset!’,
                 QMessageBox.Critical, ’Error’, QMessageBox.Ok)

            else:
            # Size match. Proceed with an update

                self.LOADED_DATASET.update_from_child(self.LOADED_DATASET2,
                 child_selected)

                popup_message(’Dataset Successfully Updated’, QMessageBox.Information,
                 ’Update Dataset’, QMessageBox.Ok)

                # Reload parent DS
                reload_path = self.LOADED_DATASET.CurrentPath
                del self.LOADED_DATASET  # DS Unloaded

                self.LOADED_DATASET, self.SimsTable =
                 load_dataset_sim_table(self.Layout2, reload_path)
        else:
            popup_message(’Nothing selected!’, QMessageBox.Warning, ’Error’,
             QMessageBox.Ok)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Pile Group Analysis Program

Configuration Constants PG_Main.py

```
import os

ldc_measure_Z = 0

PLOTS_1 = os.getcwd() + ’\\PLOT_SETS.xlsx’

SINGLE_PILES_FN = os.getcwd() + ’\\SINGLE_PILES.sps’

py_z_increments = 11
poly_order = 10
poly_order_ldc = 6
```

```
spline_order = 5
spline_knots_perc = 0.1

z_toler = 0.01
ldc_nodes_added = 50

Z_Fixed_Getters = [0,
                   -0.05, -0.10,
                   -0.15, -0.20,
                   -0.25, -0.30,
                   -0.35, -0.40,
                   -0.45, -0.50,
                   -0.55, -0.60,
                   -0.65, -0.70,
                   -0.75, -0.80,
                   -0.85, -0.90, -0.95]

Disp_Fixed_Getters = [0.005, 0.010,
                      0.015, 0.020,
                      0.025, 0.030,
                      0.035, 0.040,
                      0.045, 0.050,
                      0.055, 0.060,
                      0.065, 0.070,
                      0.075, 0.080,
                      0.085, 0.090,
                      0.095, 0.100]    # Normalized by pile diameter (y/D [-])

# ------------------------------------------------------------------------------------
PS_PlotSet = {'P': ('v', ':', 'Polyfit - ', 'g'),

              'S': ('s', '--', 'Spline - ', 'r'),

              'C': ('o', '-', 'Calculated - ', 'b')}

Colors_2 = ['b', 'r', 'g', 'm', 'y', 'c', 'k']

Z_Title = {True: 'Normalized Pile Length z/L [-]',
           False: 'Pile Length Z [m]'}

# -------------------------------------------------------------------------------------

Pile_Functions = {'LP': ('lp_pile_curve', 'lp_row_curve', None, 'Load Proportions ',
 'Load Proportion [-]'),

                  'IF': ('if_pile_curve', 'if_row_curve', None,
                  'Interaction Factors ', 'Interaction Factor [-]'),

                  'LDC': ('q_y_pile_curve', 'q_y_row_curve', 'q_y_group_curve',
                         'Load-Displacement Curves ', 'Horizontal Force [kN]'),

                  'GW': (None, None, 'gw_curve', 'Group efficiency ', 'GW [-]'),
                  'PY': (None, None, None, 'PY Curves ',
                  'Soil resistance per unit length [kN/m]')}

Markers = ['o', 's', '^', 'D', 'x', '8']

Norm_Disp_Title = {True: 'Normalized Horizontal Displacements y/D [-]',
                   False: 'Horizontal Displacements [m]'}

Title_Add = {'Pile': '- Piles', 'Row': '- Pile Rows', 'Group': '- Pile Group'}
```

Basic Classes PG_Classes.py

```
import matplotlib.pyplot as plt
import openpyxl as xl

from Main_Functions import fill_xl_row_col, xy_series_xl


class XY_Series:

    def __init__(self, x_data, y_data, color, marker, line_type, legend_label,
    norm_x, norm_y):

        # Store XY Data. Normalize XY Data.
        self.X = x_data
        self.Y = y_data

        self.XN = [x / norm_x for x in x_data]
        self.YN = [y / norm_y for y in y_data]

        self.LineColor = color
        self.LineMarker = marker
        self.LineType = line_type
        self.LineLabel = legend_label

    def get_xy(self, normalized):

        if normalized:
            return self.XN, self.YN

        else:
            return self.X, self.Y


class XY_Diagram:

    def __init__(self, title, x_label, y_label, xy_series):

        self.Title = title
        self.XLabel = x_label
        self.YLabel = y_label

        self.Plots = xy_series

    def __matplot_figure(self, show_legend, normalized):

        plt.figure()
        plt.grid(True)

        plt.title(self.Title)
        plt.xlabel(self.XLabel)
        plt.ylabel(self.YLabel)

        for plot in self.Plots:

            X, Y = plot.get_xy(normalized)

            plt.plot(X, Y, label=plot.LineLabel,
            linestyle=plot.LineType, color=plot.LineColor, marker=plot.LineMarker)

            if show_legend:
                plt.legend(loc='best', fontsize=10)

        plt.show()

    def fill_xl_ws(self, normalized, sim_name, ws, start_r, start_c, jumper):

        for plot in self.Plots:

            X, Y = plot.get_xy(normalized)

            # Fill Headers
            fill_xl_row_col(ws, 'R', [sim_name, ''], start=(start_r, start_c))
            fill_xl_row_col(ws, 'R', [plot.LineLabel, ''], start=(start_r + 1,
             start_c))

            fill_xl_row_col(ws, 'R', [self.XLabel, self.YLabel], start=(start_r + 2,
             start_c))

            fill_xl_row_col(ws, 'R', ['X', 'Y'], start=(start_r + 3, start_c))

            # Fill XY Data.
            fill_xl_row_col(ws, 'C', X, start=(start_r+4, start_c))
            fill_xl_row_col(ws, 'C', Y, start=(start_r+4, start_c+1))

            start_c += jumper       # Jump to next column block

    def create_xl_series(self, ws, start_r, start_c, jumper, sim_index):

        all_series = []

        for j, plot in enumerate(self.Plots):
            all_series.append(xy_series_xl(ws, start_r, start_c, len(plot.X),
             str(sim_index)+'_' + str(j), 'triangle'))

            start_c += jumper

        return all_series

    def execute(self, plot_export, show_legend, normalized, export_fn, sim_name):

        if plot_export == 'Plot':
            self.__matplot_figure(show_legend, normalized)

        else:
            wb = xl.Workbook()
            ws = wb.worksheets[0]

            self.fill_xl_ws(normalized, sim_name, ws, 1, 1, 2)

            wb.save(export_fn)

            print('Data Table ' + export_fn + ' succesfully exported!')
```

Pile and Pile Group Models Classes PG_ModelClasses.py

```
import numpy as np
from scipy.interpolate import make_lsq_spline

from M2_Solvers_PLAXIS_Functions import I_Circle, Sinus, Cosinus

from PG_Config import *
from PG_Classes import XY_Series, XY_Diagram


def color_picker(x):

    total = len(Colors_2)

    if x <= total-1:
        return Colors_2[x]

    return Colors_2[x-(x // total)*total]


def bspline(x1, y1, k=spline_order):

    delta = max(x1)-min(x1)

    knots = int(spline_knots_perc*len(x1))

    knots_uniform = np.r_[(x1[-1],)*k, [min(x1) + i*delta/knots for i in range(0,
     knots+1)], (x1[0],)*k]
    return make_lsq_spline(x1[::-1], y1[::-1], knots_uniform, k)


def create_diagram(t1, title, x_label, y_label, plot_sets, plot_export, leg, norm,
 export_filename, unique_fn):

    diagram = XY_Diagram(title, x_label, y_label, plot_sets)

    if t1 == 0:
    # Trigger 0 >>> Create and plot/xl process single diagram with given plot sets

        diagram.execute(plot_export, leg, norm, export_filename, unique_fn)

    return diagram
    # Trigger 1 >>> Return diagram without plotting/exporting, for further processing


def unpack_input(input_vector, single_group):

    n, m, sx, sy, d, l1, l2_0 = input_vector[0:7]
    angle = input_vector[9]
    E = input_vector[25]
    cap_thick, cap_type = input_vector[29:31]

    if single_group == 'SINGLE':
        return d, l1, l2_0 + cap_thick*d, E, int(cap_type), cap_thick
    else:
        return n, m, sx, sy, d, l1, l2_0 + cap_thick*d, E, angle, int(cap_type),
         cap_thick


def interpolate(xl, yl, xr, yr, x):

    return yl + (x-xl)*(yr-yl)/(xr-xl)


def extrapolate_right(xl, yl, xr, yr, x):

    return yr + (x-xr)*(yr-yl)/(xr-xl)


def extrapolate_left(xl, yl, xr, yr, x):

    return yl + (x-xl)*(yr-yl)/(xr-xl)


def find_bounds(input_list, x):
    # input_list must contain at least 2 elements. works better with sorted list
    # Returns search status and boundary indices

    min_x = min(input_list)
    max_x = max(input_list)
    size_list = len(input_list)

    if min_x <= x <= max_x:         # Search value is inside list boundaries

        for i, x_list in enumerate(input_list[:-1]):
        # Search all but last element of the list, starting from 0.

            if x >= x_list:
                return 'Interpolate', i, i+1

    elif x < min_x:
        return 'ExtrapolateLeft', 0, 1

    else:
        return 'ExtrapolateRight', size_list-2, size_list-1


Interpolator = {'Interpolate': interpolate,
                'ExtrapolateRight': extrapolate_right,
                'ExtrapolateLeft': extrapolate_left}


class Pile:

    def __init__(self, diameter, length_below, length_above, E_pile, cap_type,
     cap_thickness):

        self.Diameter = diameter
        self.LengthTotal = length_above + length_below

        self.ZBottom = -length_below            # Z=0 at the ground level
        self.ZTop = length_above

        if cap_type == 0:
            self.HeadConditions = 'Free'
            self.CapThickness = 0

        else:
            self.HeadConditions = 'Fixed'
            self.CapThickness = cap_thickness*self.Diameter
            # Measured from pile top to down

        self.E = E_pile
        self.I = I_Circle(diameter)

        # Initialize class attributes for later use ##################################

        # Group attributes  ..........................................................
        self.IndexPG = None
        self.X = None
        self.Y = None
        self.LoadAngle = None
        # Result Slots ................................................................
        self.ZRaw = None
        self.Results = None
        self.Steps = None
        self.QForcesRaw = None

        self.Ux_Fitting = None
        self.Uy_Fitting = None
        self.d2Mn_Fitting = None
        self.Fitted = None

        self.LDC_Fitted_P = None
        self.LDC_Fitted_S = None

        self.LDC_U_C = None
        self.LDC_Q_P = None
        self.LDC_Q_S = None

        self.PY = None
        self.PY_Z = None

    @property
    def Z(self):
        return self.Results[:, 2, 0]

    # Class methods. By order of appearance in the instance life ####################

    # Pile initialization and results processing functions ........................

    def add_to_group_rectangular(self, i, x, y, beta):
        self.IndexPG = i
        self.X = x
        self.Y = y
        self.LoadAngle = beta

    def set_alone(self):
        self.IndexPG = 0
        self.X = 0.0
        self.Y = 0.0
        self.LoadAngle = 0.0

    def __match_my_axis(self, node_coordinates):

        x, y, z = node_coordinates

        check_x = abs(self.X-x) <= z_toler

        check_y = abs(self.Y-y) <= z_toler

        check_z = self.ZBottom-z_toler <= z <= self.ZTop+z_toler

        return check_x and check_y and check_z
```

```
@staticmethod
def __sort_by_Z(unsorted):

    sorted_results = np.zeros(np.shape(unsorted))
    # Initialize results

    for step in range(0, len(unsorted[0, 0, :])):
    # Cycle each step

        step_data = unsorted[:, :, step]
        # Pick step data 2D table

        z_sorted_indices = step_data[:, 2].argsort()
        # Sort according to Z column (with index 2)

        # Sort per Z (opposite direction)
        for r1, r in enumerate(z_sorted_indices[::-1]):
        # Cycle sorted indices backwards

            sorted_results[r1, :, step] = step_data[r, :]

    return sorted_results

def process_raw_results(self, results_raw):

    rows, result_types, steps = np.shape(results_raw)

    results_pile = np.zeros((rows, result_types, steps))
    # Initialize empty table with same dim. as raw results

    self.Steps = steps

    # Filtering the raw results to the pile axis ................................
    added_rows = 0

    for r in range(0, rows):
        if self.__match_my_axis(results_raw[r, 0:3, 0]):
            results_pile[added_rows, :, :] = results_raw[r, :, :]
            # Add filtered results to new table

            added_rows += 1

    pile_results_unsorted = np.resize(results_pile,
    (added_rows, result_types, steps))

    results_sorted = self.__sort_by_Z(pile_results_unsorted)

    # Keep raw Q and Z results for further use. These vectors contain ALL
     EXTRACTED, but SOME DUPLICATED results

    self.QForcesRaw = results_sorted[:, 8:10, :]
    self.ZRaw = results_sorted[:, 2, :]
```

```
        # Remove duplicated values from results ....................................
        results_unique = np.zeros(np.shape(results_sorted))

        foundrows, r = (0, 0)
        # Remove duplicated rows. Array is already sorted by Z

        while r < added_rows:
            results_unique[foundrows, :, :] = results_sorted[r, :, :]

            duplicated_rows = []
            for r1 in range(r+1, added_rows):
            # Check next rows

                if results_sorted[r1, 2, 0] == results_sorted[r, 2, 0]:
                # Check Z coordiate equality

                    duplicated_rows.append(r1)

                else:
                    break

            r += 1+len(duplicated_rows)
            # Skip Duplicated Rows

            foundrows += 1
            # Increase unique row counter by 1

        # Save unique results table to a Pile object instance
        self.Results = np.resize(results_unique, (foundrows, result_types, steps))

        self.__fit_beam_curves()

    def __count_cap_nodes(self):

        if self.HeadConditions == 'Free':
            return 0, 0

        else:
            cap_bottom_Z = self.ZTop - (self.CapThickness + z_toler)

            cap_points_M = len([z for z in self.Z if z >= cap_bottom_Z])

            cap_points_Q = len([z for z in self.ZRaw[:, 0] if z >= cap_bottom_Z])

            return cap_points_M, cap_points_Q
            # M-unique, Q-raw nodes

    def __fit_beam_curves(self):

        FITTED_RESULTS = np.zeros((len(self.Results), 24, self.Steps))
        # Initialize fitted results data table

        # Fitting pile results using splines and polynoms.
```

```
# Unique Data OUTSIDE cap zone is used.
# Store fitting parameters for further use.
# Use Moments !!BELOW PILE CAP!! for approximation of results.

cap_points_M, cap_points_Q = self.__count_cap_nodes()

self.Ux_Fitting = []
self.Uy_Fitting = []

for step in range(0, self.Steps):

    # Get initial CALCULATED data .........................................
    Ux = self.Results[:, 3, step]
    Uy = self.Results[:, 4, step]

    M2 = self.Results[:, 10, step]
    M3 = self.Results[:, 11, step]

    M2_cap = M2[cap_points_M:]          # Bending moments below pile cap
    M3_cap = M3[cap_points_M:]
    Z_M = self.Z[cap_points_M:]         # Z nodes below pile cap

    # Calculate approximation polynoms and B-splines ......................
    M2_fitp = np.polyfit(Z_M, M2_cap, poly_order)
    M3_fitp = np.polyfit(Z_M, M3_cap, poly_order)
    M2_spline = bspline(Z_M, M2_cap)
    M3_spline = bspline(Z_M, M3_cap)

    Ux_fitp = np.polyfit(self.Z, Ux, poly_order)
    Uy_fitp = np.polyfit(self.Z, Uy, poly_order)
    Ux_spline = bspline(self.Z, Ux)
    Uy_spline = bspline(self.Z, Uy)

    self.Ux_Fitting.append((Ux_fitp, Ux_spline))
    self.Uy_Fitting.append((Uy_fitp, Uy_spline))

    # Calculate approximation derivatives ................................
    d1M2_fitp = np.polyder(M2_fitp)
    d2M2_fitp = np.polyder(M2_fitp, 2)

    d1M3_fitp = np.polyder(M3_fitp)
    d2M3_fitp = np.polyder(M3_fitp, 2)

    d1M2_spline = M2_spline.derivative()
    d2M2_spline = M2_spline.derivative(2)

    d1M3_spline = M3_spline.derivative()
    d2M3_spline = M3_spline.derivative(2)

    # Calculate. All fittings are of the length of Z (unique Z nodes)

    FITTED_RESULTS[:, 0, step] = Ux
    # Ux calculated
```

```
FITTED_RESULTS[:, 1, step] = Uy
# Uy calculated

FITTED_RESULTS[:, 2, step] = M2
# M2 calculated

FITTED_RESULTS[:, 3, step] = M3
# M3 calculated

FITTED_RESULTS[:, 4, step] = [np.polyval(M2_fitp, z) for z in self.Z]
# M2 polyfit

FITTED_RESULTS[:, 5, step] = M2_spline(self.Z)
# M2 spline fit

FITTED_RESULTS[:, 6, step] = [np.polyval(M3_fitp, z) for z in self.Z]
# M3 polyfit

FITTED_RESULTS[:, 7, step] = M3_spline(self.Z)
# M3 spline fit

FITTED_RESULTS[:, 8, step] = [-np.polyval(d1M3_fitp, z) for z in self.Z]
# Q2=dM3/dz polyfit

FITTED_RESULTS[:, 9, step] = -d1M3_spline(self.Z)
# Q2=dM3/dz spline fit

FITTED_RESULTS[:, 10, step] = [np.polyval(d1M2_fitp, z) for z in self.Z]
# Q3=dM2/dz polyfit

FITTED_RESULTS[:, 11, step] = d1M2_spline(self.Z)
# Q3=dM2/dz spline fit

# P=d2M/dz2=dQ/dz - Soil reaction per unit length [kN/m]
# p=P/D - Soil pressure [kN/m2]

FITTED_RESULTS[:, 12, step] =
[-1 / self.Diameter * np.polyval(d2M3_fitp, z) for z in self.Z]
# p2 polyfit

FITTED_RESULTS[:, 13, step] =
-1 / self.Diameter * d2M3_spline(self.Z)
# p2 spline

FITTED_RESULTS[:, 14, step] =
[1 / self.Diameter * np.polyval(d2M2_fitp, z) for z in self.Z]
# p3 polyfit

FITTED_RESULTS[:, 15, step]
= 1 / self.Diameter * d2M2_spline(self.Z)
# p3 spline
# ###############################################################################
# Calculate displacement and moments projections using loading angle
SinB, CosB = (Sinus(self.LoadAngle), Cosinus(self.LoadAngle))

for step in range(0, self.Steps):
    for node in range(0, len(self.Results)):

        Ux = self.Results[node, 3, step]
        Uy = self.Results[node, 4, step]

        Us = Ux * CosB + Uy * SinB

        M2 = self.Results[node, 10, step]
        M3 = self.Results[node, 11, step]

        Mn = -M2 * SinB + M3 * CosB

        FITTED_RESULTS[node, 16, step] = Us
        # Displacements along loading direction - Us calculated

        FITTED_RESULTS[node, 17, step] = Mn
        # Moments in the loading dir. plane

# Approximate Max Moments using polynoms and splines.
# Recalculate Q12, Q13 from moment derivatives

self.d2Mn_Fitting = []

for step in range(0, self.Steps):

    Mn = FITTED_RESULTS[:, 17, step]

    Mn_cap = Mn[cap_points_M:]
    # Remove pile cap nodes from bending moments

    Z_M = self.Z[cap_points_M:]

    # Calculate polyfit/spline of Mn. Calculate moment derivatives
    Mn_fitp = np.polyfit(Z_M, Mn_cap, poly_order)
    Mn_spline = bspline(Z_M, Mn_cap)

    d1Mn_fitp = np.polyder(Mn_fitp)
    d2Mn_fitp = np.polyder(Mn_fitp, 2)

    d1Mn_spline = Mn_spline.derivative()
    d2Mn_spline = Mn_spline.derivative(2)

    # Calculate MAX bending moments, Q-forces and pressures

    FITTED_RESULTS[:, 18, step] = [np.polyval(Mn_fitp, z) for z in self.Z]
    # Mn polyfit (MAX M)

    FITTED_RESULTS[:, 19, step] = Mn_spline(self.Z)
```

```
        # Mn spline fit

        FITTED_RESULTS[:, 20, step] = [-np.polyval(d1Mn_fitp, z) for z in self.Z]
        # Qs polyfit (Max Q)

        FITTED_RESULTS[:, 21, step] = -d1Mn_spline(self.Z)
        # Qs spline fit

        FITTED_RESULTS[:, 22, step] =
        [-1 / self.Diameter * np.polyval(d2Mn_fitp, z) for z in self.Z]
        # ps polyfit

        FITTED_RESULTS[:, 23, step] =
        -1 / self.Diameter * d2Mn_spline(self.Z)
        # ps spline

        self.d2Mn_Fitting.append((d2Mn_fitp, d2Mn_spline))
        # Save moment derivative fitting for further use

    self.Fitted = FITTED_RESULTS

    self.__create_LDC()
    self.__create_PY_nodes()

def __find_Z_node(self, z_to_find):
    # Return index of the results node (from unique results table) closest
    to the provided Z coordinate

    for j, z in enumerate(self.Z):

        lower = z_to_find - z_toler
        upper = z_to_find + z_toler

        if lower <= z <= upper:
            return j

def __create_LDC(self):

    results_table = np.zeros((3, self.Steps))
    # Initialize empty table {3 x Steps}: [Us/Q_poly/Q_spline]

    observation_Z = self.__find_Z_node(ldc_measure_Z)
    # Point where displacements are measured (GOK by default)

    for step in range(1, self.Steps):
    # Starts from 1 because start point is already (0,0) - ZEROS

        results_table[0, step] = self.Fitted[observation_Z, 16, step]
        # Us calculated

        results_table[1, step] = self.Fitted[observation_Z, 20, step]
        # Qs polyfit
```

```
            results_table[2, step] = self.Fitted[observation_Z, 21, step]
            # Qs spline

        # Results table created. Split it into pile attributes for further use.
        # Polyfit/Spline Q results and calculated Us are stored
        self.LDC_U_C = results_table[0, :]
        self.LDC_Q_P = results_table[1, :]
        self.LDC_Q_S = results_table[2, :]

        # Fit Load-Displacement Curves using polynoms (order of poly_order_ldc),
        USING POLY/SPLINE DATA.

        self.LDC_Fitted_P = np.polyfit(self.LDC_U_C, self.LDC_Q_P, poly_order_ldc)
        self.LDC_Fitted_S = np.polyfit(self.LDC_U_C, self.LDC_Q_S, poly_order_ldc)

    def __create_PY_nodes(self):
        # Creates PY curves for all depths below GOK (Z < 0)

        self.PY_Z = [z for z in self.Z if z < 0]        # Get Z values below GOK

        PY = np.zeros((3, self.Steps, len(self.PY_Z)))  # 3D Array to store PY Curves

        first_z_below = self.__find_Z_node(0) + 1

        for j, z in enumerate(self.PY_Z):
            for step in range(1, self.Steps):
                PY[0, step, j] = abs(self.Fitted[first_z_below + j, 16, step])
                # Us - Calculated

                PY[1, step, j] = abs(self.Fitted[first_z_below + j, 22,
                 step])*self.Diameter
                 # Ps - polyfit

                PY[2, step, j] = abs(self.Fitted[first_z_below + j, 23,
                 step])*self.Diameter
                 # Ps - spline fit

        self.PY = PY

    # At this point, all pile results are processed and stored inside
    Pile object instance

    # Additional simple results DISPLAY functions
    def __raw_results(self, results_index, step):

        if results_index >= 100:
            return self.QForcesRaw[:, results_index-100, step], self.ZRaw[:, step]
            # Raw Q forces - Index 100+

        else:
            return self.Fitted[:, results_index, step], self.Z
            # Other results - Index 0+
    def __RAW_XY(self, results_index, step, p_s_c):

        X, Y = self.__raw_results(results_index, step)
        # Get XY results from stored result tables, by index

        marker, linetype, label, color = PS_PlotSet[p_s_c]

        return XY_Series(X, Y, color, marker, linetype, label + 'Step-' + str(step),
         1, -self.ZBottom)

    def RAW_PLOT(self, x_label, i, steps, p_s_c, leg, p_exp, norm, unique_fn, t1):

        export_filename = unique_fn + '_' + x_label + '_' + p_s_c + '.xlsx'
        return create_diagram(t1, 'Pile-' + str(self.IndexPG), x_label, Z_Title[norm],
                              [self.__RAW_XY(i, s, p_s_c) for s in steps], p_exp, leg,
                               norm, export_filename, unique_fn)

    def get_steps(self, which_steps):

        if which_steps == 'LAST':
        # Return last step index (as the list)
            return [self.Steps-1]

        else:
            return [x for x in range(0, self.Steps)]
            # Return all steps indices list

    # Pile LDC Processing functions
    def q_y(self, y, p_s):

        # Interpolate load for any given displacement value (recalculate from stored
         pile LDC POLYNOMIAL TRENDLINE)
        # NO LINEAR INTERPOLATION IS DONE!!!!!!

        if p_s == 'P':
            return np.polyval(self.LDC_Fitted_P, y)

        else:
            return np.polyval(self.LDC_Fitted_S, y)

    def qy_curve(self, U, p_s):
        # Calculate loads for given vector of displacements U. Set first value to 0
         (starting point)

        curve = [self.q_y(u, p_s) for u in U]
        curve[0] = 0

        return curve

    def __LDC_XY(self, p_s, fitted_calculated):

        marker, linetype, label, color = PS_PlotSet[p_s]
        X = self.LDC_U_C
        # Saved vector of calculated Us displacements

        if p_s == 'P':
            Y = self.LDC_Q_P
            # Saved vector of Q-forces by polynomial interpolation of bending moments

        else:
            Y = self.LDC_Q_S
            # Saved vector of Q-forces by spline interpolation of bending moments

        return XY_Series(X, Y, color, marker, linetype, label, self.Diameter, 1)

    def LDC_PY_PLOT(self, ldc_py, p_s, curve_parameter, leg, p_exp, norm,
    unique_fn, t1):

        # curve_parameter = FITTED/CALCULATED...NODES/Z

        if ldc_py == 'LDC':
            plot_series = [self.__LDC_XY(p_s, curve_parameter)]

        else:
            plot_series = self.__PY_XY_SERIES(curve_parameter, p_s)

        return create_diagram(t1,
                              'Pile-' + str(self.IndexPG),
                              Norm_Disp_Title[norm],
                              Pile_Functions[ldc_py][4],
                              plot_series,
                              p_exp,
                              leg,
                              norm,
                              unique_fn + '_' + ldc_py + '_' + str(curve_parameter)
                               + '_' + p_s + '.xlsx',
                              unique_fn)

    # PY Curve funcions - Check later if needed

    def __PY_z(self, z, p_s):

        # Create py curve at desired z. First point is set to (0, 0)

        SinB, CosB = Sinus(self.LoadAngle), Cosinus(self.LoadAngle)

        PY = np.zeros((4, self.Steps))

        for step in range(1, self.Steps):

            Ux_fitp, Ux_spline = self.Ux_Fitting[step]
            Uy_fitp, Uy_spline = self.Uy_Fitting[step]
            d2Mn_fitp, d2Mn_spline = self.d2Mn_Fitting[step]
            # Use previously created fitting polynoms and splines
```

```
            Ux1 = np.polyval(Ux_fitp, z)
            Uy1 = np.polyval(Uy_fitp, z)

            Ux2 = Ux_spline(z)
            Uy2 = Uy_spline(z)

            PY[0, step] = abs(Ux1 * CosB + Uy1 * SinB)      # Displacements - Polyfit
            PY[1, step] = abs(np.polyval(d2Mn_fitp, z))
            # Soil resistances - Polyfit

            PY[2, step] = abs(Ux2 * CosB + Uy2 * SinB)      # Displacements - Splines
            PY[3, step] = abs(d2Mn_spline(z))
            # Soil resistances - Spline

        if p_s == 'P':
            return PY[0:2, :]

        return PY[2:, :]

    def __PY_node(self, z_node, p_s):

        X = self.PY[0, :, z_node]

        if p_s == 'P':
            Y = self.PY[1, :, z_node]

        else:
            Y = self.PY[2, :, z_node]

        return X, Y

    def __PY_Z_XY(self, p_s, z, j):

        marker, linetype, label, color = PS_PlotSet[p_s]

        py_curve = self.__PY_z(z, p_s)
        X = py_curve[0, :]
        Y = py_curve[1, :]

        return XY_Series(X, Y, color_picker(j), marker, linetype,
        label+'Z='+str(round(z, 2)), self.Diameter, 1)

    def __PY_NODE_XY(self, p_s, z, j):

        marker, linetype, label, color = PS_PlotSet[p_s]

        X = self.PY[0, :, j]

        if p_s == 'P':
            Y = self.PY[1, :, j]
        else:
            Y = self.PY[2, :, j]
        return XY_Series(X, Y, color_picker(j), marker, linetype, label+'Z='
        + str(round(z, 2)), self.Diameter, 1)

    def __PY_XY_SERIES(self, nodes_or_z, p_s):

        if nodes_or_z == 'Z':
            z_list = [-i * self.Diameter for i in range(1, py_z_increments)]
            py_curve_f = self.__PY_Z_XY

        else:
            z_list = self.PY_Z
            py_curve_f = self.__PY_NODE_XY

        return [py_curve_f(p_s, z, j) for j, z in enumerate(z_list)]


    def interpolate_step_results(self, x, results_index):

        x_list = self.LDC_U_C[:]

        action, i0, i1 = find_bounds(x_list, x)

        xl, xr = x_list[i0], x_list[i1]

        list_y0, list_y1 = self.Fitted[:, results_index, i0],
        self.Fitted[:, results_index, i1]

        return [Interpolator[action](xl, yl, xr, yr, x)
        for yl, yr in zip(list_y0, list_y1)]


    def get_single_step_result(self, y_disp, z, results_index):

        y_list = self.interpolate_step_results(y_disp, results_index)
        action, i0, i1 = find_bounds(self.Z, z)

        return Interpolator[action](self.Z[i0], y_list[i0], self.Z[i1], y_list[i1], z)


    def get_maxM(self, y, p_s):

        if p_s == 'P':
            return max(self.interpolate_step_results(y, 18))

        else:
            return max(self.interpolate_step_results(y, 19))


class PGModel_PLAXIS:

    def __init__(self, input_vector, raw_results, eq_sp, soil):

        n, m, sx, sy, d, l1, l2, E_pile, angle, cap_type, cap_thick =
         unpack_input(input_vector, 'GROUP')

        self.N = int(n)
        self.M = int(m)
        self.Sx = sx
        self.Sy = sy

        if self.N == 1 and self.M == 1:     # Single Pile
            beta = 0
        else:                               # Pile Group
            beta = angle

        self.Piles = []
        # Create pile instances and add it to group

        for r in range(0, self.N):
            for c in range(0, self.M):
                pile = Pile(d, l1, l2, E_pile, cap_type, cap_thick)
                pile.add_to_group_rectangular(r*self.M + c, r*sx*d, c*sy*d, beta)
                pile.process_raw_results(raw_results)

                self.Piles.append(pile)
                # Pile is added after results processing. Equivalent single pile is
                 empty!

        self.EqSP = eq_sp

        self.Soil = soil
        self.DBResults_1 = None
        self.DBResults_2 = None
        self.DBResults_3 = []

    @property
    def __Displacements(self):
        # Create equidistant displacement points 0-Max U

        all_piles = self.Piles[:]

        if self.EqSP is not None:
            all_piles.append(self.EqSP)

        max_pile_U = [np.max(pile.LDC_U_C) for pile in all_piles]

        MAX_disp = max(max_pile_U)

        return [j*MAX_disp/(ldc_nodes_added-1) for j in range(0, ldc_nodes_added)]

    def __get_pile_by_id(self, index):

        for pile in self.Piles:
            if pile.IndexPG == index:
                return pile
        return None

    # LOAD-DISPLACEMENT CURVES .........................................
    def q_y_pile_curve(self, p_i, p_s):

        pile = self.__get_pile_by_id(p_i)

        return pile.qy_curve(self.__Displacements, p_s)

    def __q_y_row(self, y, r, t_a, p_s):

        row_values = []
        for c in range(0, self.M):
            pile = self.__get_pile_by_id(r*self.M + c)

            row_values.append(pile.q_y(y, p_s))

        if t_a == 'Average':
            return sum(row_values)/len(row_values)

        return sum(row_values)

    def __q_y_group(self, y, t_a, p_s):

        group_Q = sum([pile.q_y(y, p_s) for pile in self.Piles])

        if t_a == 'Average':
            return group_Q/len(self.Piles)

        return group_Q

    def q_y_row_curve(self, r, p_s, t_a='Total'):

        curve = [self.__q_y_row(y, r, t_a, p_s) for y in self.__Displacements]
        curve[0] = 0

        return curve

    def q_y_group_curve(self, p_s, t_a='Total'):

        curve = [self.__q_y_group(y, t_a, p_s) for y in self.__Displacements]
        curve[0] = 0

        return curve


    # LOAD PROPORTIONS

    def __lp_y_pile(self, y, p_i, p_s):

        pile = self.__get_pile_by_id(p_i)

        return pile.q_y(y, p_s)/self.__q_y_group(y, 'Total', p_s)
```

```
def __lp_y_row(self, y, r, p_s):

    return sum([self.__lp_y_pile(y, r*self.M + c, p_s)
    for c in range(0, self.M)])

def lp_pile_curve(self, p_i, p_s):

    curve = [self.__lp_y_pile(y, p_i, p_s) for y in self.__Displacements]
    curve[0] = 1/len(self.Piles)

    return curve

def lp_row_curve(self, r, p_s, t_a='Total'):

    curve = [self.__lp_y_row(y, r, p_s) for y in self.__Displacements]
    curve[0] = 1/self.N

    return curve

# INTERACTION FACTORS

def __if_y_pile(self, y, p_i, p_s):

    pile = self.__get_pile_by_id(p_i)
    return pile.q_y(y, p_s) / self.EqSP.q_y(y, p_s)

def __if_y_row(self, y, r, p_s):
    return self.__q_y_row(y, r, 'Total', p_s) / self.EqSP.q_y(y, p_s)

def if_pile_curve(self, p_i, p_s):

    curve = [self.__if_y_pile(y, p_i, p_s) for y in self.__Displacements]
    curve[0] = 1

    return curve

def if_row_curve(self, r, p_s, t_a='Total'):

    curve = [self.__if_y_row(y, r, p_s) for y in self.__Displacements]
    curve[0] = 1

    return curve

# GROUP EFFICIENCY
def __gw_y(self, y, p_s):

    Qg_total = self.__q_y_group(y, 'Total', p_s)
    Qsp = self.EqSP.q_y(y, p_s)

    return Qg_total / (Qsp*len(self.Piles))

def gw_curve(self, p_s, t_a):
    curve = [self.__gw_y(y, p_s) for y in self.__Displacements]
    curve[0] = 1

    return curve

# FINAL PLOT GENERATOR FUNCTIONS - PILES/ROWS/GROUP
def __PILE_XY(self, r, c, lp_if_ldc_gw, p_s):

    marker, linetype, label, color = PS_PlotSet[p_s]
    label += 'Pile-' + str(r) + str(c)

    pile_curve_f = getattr(self, Pile_Functions[lp_if_ldc_gw][0])

    return XY_Series(self.__Displacements, pile_curve_f(r*self.M + c, p_s),
                     Colors_2[r], Markers[c], linetype, label,
                      self.Piles[0].Diameter, 1)


def __ROW_XY(self, r, lp_if_ldc_gw, p_s, t_a):

    marker, linetype, label, color = PS_PlotSet[p_s]
    label += 'Row-' + str(r)

    row_curve_f = getattr(self, Pile_Functions[lp_if_ldc_gw][1])

    Y = row_curve_f(r, p_s)

    if t_a is not None:
        label += '-' + t_a
        Y = row_curve_f(r, p_s, t_a)

    return XY_Series(self.__Displacements, Y, Colors_2[r], marker, linetype,
     label, self.Piles[0].Diameter, 1)


def __GROUP_XY(self, lp_if_ldc_gw, p_s, t_a):

    marker, linetype, label, color = PS_PlotSet[p_s]

    group_curve_f = getattr(self, Pile_Functions[lp_if_ldc_gw][2])

    Y = group_curve_f(p_s, t_a)

    if t_a is not None:
        Y = group_curve_f(p_s, t_a)
        label += '-' + t_a

    return XY_Series(self.__Displacements, Y, color, marker, linetype, label,
     self.Piles[0].Diameter, 1)

def __PICK_XY(self, r, c, lp_if_ldc_gw, p_s, prg, t_a):
    if prg == 'Pile':
        return self.__PILE_XY(r, c, lp_if_ldc_gw, p_s)

    elif prg == 'Row':
        return self.__ROW_XY(r, lp_if_ldc_gw, p_s, t_a)

    else:
        return self.__GROUP_XY(lp_if_ldc_gw, p_s, t_a)


def __XY_GROUP(self, lp_if_ldc_gw, p_s, prg, t_a):

    rows, columns = (self.N, self.M)

    if prg == 'Row':
        rows, columns = (self.N, 1)

    elif prg == 'Group':
        rows, columns = (1, 1)

    PlotSeries = []
    for r in range(0, rows):
        for c in range(0, columns):
            PlotSeries.append(self.__PICK_XY(r, c, lp_if_ldc_gw, p_s, prg, t_a))

    return PlotSeries

def PG_PLOT(self, prg, t_a, lp_if_ldc_gw, p_s, leg, p_exp, norm, unique_fn, t1):

    exp_fn = unique_fn + '_' + lp_if_ldc_gw + '_' + prg + '_' + t_a
     + '_' + p_s + '.xlsx'

    return create_diagram(t1, Pile_Functions[lp_if_ldc_gw][3]
     + Title_Add[prg], Norm_Disp_Title[norm],

                          Pile_Functions[lp_if_ldc_gw][4],
                           self.__XY_GROUP(lp_if_ldc_gw, p_s, prg, t_a),

                          p_exp, leg, norm, exp_fn, unique_fn)

# Create Fixed Displacement Results based on configuration input list
 Displacement_Fixed_Getters

def get_basics(self):
    return self.Soil, self.Piles[0].HeadConditions, self.Piles[0].LoadAngle,
     self.Sx, self.N, self.Sy, self.M

def set_fixed_results(self, p_s):

    z_snaps = len(Z_Fixed_Getters)

    new_table_1 = np.zeros((len(self.Piles), 3 + 2*z_snaps,
     len(Disp_Fixed_Getters)))
```

```
new_table_2 = np.zeros((len(Disp_Fixed_Getters), 4))

for k, y_norm in enumerate(Disp_Fixed_Getters):

    y = y_norm * self.Piles[0].Diameter

    # Get results for each pile - DBResults_1 ...............................

    for p_i, pile in enumerate(self.Piles):

        # These results are independent of Z - calculated for predefined Z
         level (GROUND LEVEL)

        new_table_1[p_i, 0, k] = pile.q_y(y, p_s)
        new_table_1[p_i, 1, k] = self.__lp_y_pile(y, p_i, p_s)
        new_table_1[p_i, 2, k] = self.__if_y_pile(y, p_i, p_s)

        # Get results for each pile, along z axis, on predefined Z getters
        for z_i, z_norm in enumerate(Z_Fixed_Getters):
            new_table_1[p_i, 3+z_i, k] = pile.get_single_step_result
            (y, -z_norm*pile.ZBottom, 16)

            new_table_1[p_i, 3+z_i + z_snaps, k] =
             pile.get_single_step_result(y, -z_norm*pile.ZBottom, 19)

    # Get results for whole group - DBResults_2 ............................
    new_table_2[k, :] = [self.__gw_y(y, p_s),
                         self.__q_y_group(y, 'Total', p_s),
                         self.__q_y_group(y, 'Average', p_s),
                         self.EqSP.q_y(y, p_s)]

# Store results in the database. These results are independent of the pile
 shape

self.DBResults_1 = new_table_1
self.DBResults_2 = new_table_2

# Get interpolated curves for each pile, including single pile ##############
all_piles = self.Piles[:]
all_piles.append(self.EqSP)

for pile in all_piles:

    pile_table = np.zeros((len(pile.Z), 2, len(Disp_Fixed_Getters)))

    for k, y_norm in enumerate(Disp_Fixed_Getters):
        y = y_norm * self.Piles[0].Diameter

        pile_table[:, 0, k] = pile.interpolate_step_results(y, 16)
        pile_table[:, 1, k] = pile.interpolate_step_results(y, 19)

    self.DBResults_3.append(pile_table)
    def db_results(self, p_s):

        results_table_piles = np.zeros((len(Disp_Fixed_Getters), len(self.Piles), 3))
        results_group = []

        for i, y_norm in enumerate(Disp_Fixed_Getters):
            y = y_norm * self.Piles[0].Diameter

            results_group.append(self.__gw_y(y, p_s))

            for p_i, pile in enumerate(self.Piles):

                results_table_piles[i, p_i, :] = [self.__lp_y_pile(y, p_i, p_s),
                                                  self.__if_y_pile(y, p_i, p_s),
                                                  pile.get_maxM(y, p_s)]

        return results_table_piles, results_group

    def get_all_ldc(self):
        return [self.DBResults_1[p_i, 0, :] for p_i, pile in enumerate(self.Piles)]
```

Main Program PG_Main.py

```
import sys
import openpyxl as xl
import zipfile

from PyQt5.QtWidgets import QApplication, QMdiArea, QMessageBox, QLabel, QComboBox,
 QPushButton, QMainWindow

from Main_WidgetClasses_small import new_subwindow
from M1_WidgetClasses import SummaryTableDS

from Main_Functions import popup_open_save, load_object, popup_message,
 popup_simple_dialog, show_notice, get_xl_row

from Main_Functions import xl_scatter_chart, save_object, fill_combo
from Main_ConfigConstants import DefaultSimDir

from PG_Combos import *
from PG_Config import PLOTS_1, SINGLE_PILES_FN
from PG_ModelClasses import PGModel_PLAXIS

from PG_Config_DB import *


def pos_db(soil, pile_top_bc, load_angle, sx, sy, n, m):

    pos0 = DB_SOILS.index(soil)
    pos1 = DB_BC.index(pile_top_bc)
    pos2 = DB_ANGLE.index(int(load_angle))
    pos3 = DB_SX.index(int(sx))

    pos5 = DB_N.index(int(n))
    pos6 = DB_SY.index(int(sy))
    pos7 = DB_M.index(int(m))

    return pos0, pos1, pos2, pos3, pos5, pos6, pos7


def save_databases(db_fn_list):

    for db, fn in db_fn_list:
        save_object(db, fn)


# Plots creator Settings ..................................................

SINGLE_PILES = load_object(SINGLE_PILES_FN)

P_S_FINAL = 'S'

BATCH_FULL_NORMALIZED = False
BATCH_SETTINGS = [7]
# 7 - No Legend/Export to XLSX/Don't normalize axes

BATCH_PLOTS = [2, 11, 19, 25, 27]
# BATCH_PLOTS = [2, 11, 19, 25, 27, 35, 41, 43, 45, 49, 51]

BATCH_STEPS = 'LAST'
BATCH_CHART_STYLE = 12
BATCH_CHART_CREATE = True
BATCH_PILES = [0, 1, 2, 3]
# List of group indices - piles for processing

ADD_SP_TRIGGER = True
BATCH_FULL = True

# Dataset specific input parameters ......................................
SIM_FILTER = [x for x in range(32, 48)] + [x for x in range(80, 96)]
SP_LINKER = [0]*48 + [1]*48
SOIL_TYPES = ['Dense']*48 + ['Loose']*48


class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)
        self.setWindowTitle('PILE GROUP ANALYSIS')
        self.commands_window, self.Layout1 = new_subwindow('Dataset Not Loaded',
        (400, 400))

        self.mdi.addSubWindow(self.commands_window)
        self.commands_window.show()

        piles_window, self.Layout2 = new_subwindow('Piles', (500, 500))
        self.mdi.addSubWindow(piles_window)
        piles_window.show()

        self.plot_choose_window, self.Layout3 = new_subwindow('Plotting', (500, 200))
        self.mdi.addSubWindow(self.plot_choose_window)
        self.plot_choose_window.show()

        # SETTING WINDOWS WIDGETS ###################################################

        # Window 1 ..........................................................
        cmdOpenDataset = QPushButton('LOAD DATASET')
        self.cmdDataTable = QPushButton('SHOW INPUT PARAMETERS')
        self.cmdShowPG = QPushButton('SHOW PILE GROUP')

        self.cmdDataTable.clicked.connect(self.cmdDataTable_clicked)
        self.cmdShowPG.clicked.connect(self.cmdShowPG_clicked)
        cmdOpenDataset.clicked.connect(self.cmdOpenDataset_clicked)

        self.label_1 = QLabel('***DATASET NOT LOADED***')
        self.label_2 = QLabel('***DATASET NOT LOADED***')
        self.cmbSims = QComboBox()

        self.Layout1.addWidget(QLabel('Dataset Description'), 0, 0)
        self.Layout1.addWidget(self.label_1, 0, 1)

        self.Layout1.addWidget(QLabel('Total Number of Simulations'), 1, 0)
        self.Layout1.addWidget(self.label_2, 1, 1)

        self.Layout1.addWidget(cmdOpenDataset, 2, 0)
        self.Layout1.addWidget(self.cmdDataTable, 2, 1)

        self.Layout1.addWidget(QLabel('Select Simulation'), 3, 0)
        self.Layout1.addWidget(self.cmbSims, 3, 1)

        self.Layout1.addWidget(self.cmdShowPG, 4, 0)

        self.__on_off_widgets(False)

        # Window 2 ..........................................................
        self.cmdPGResults = QPushButton('Plot Group Results')
        self.cmdPGResults.clicked.connect(self.plot_pg_results)
        self.cmdPGResults.setEnabled(False)

        self.cmdPGResults_All = QPushButton('Plot All Results')
        self.cmdPGResults_All.clicked.connect(self.pile_pg_results_batch)
        self.cmdPGResults_All.setEnabled(False)
        self.Layout2.addWidget(QLabel('Select Pile Diagram to Plot'), 0, 2)
        self.Layout2.addWidget(self.cmdPGResults, 3, 2)
        self.Layout2.addWidget(self.cmdPGResults_All, 4, 2)

        # Window 3 ..............................................................
        self.lblSettings3 = QLabel('Normalize Z axis')

        self.Combo0 = QComboBox()
        self.Combo1 = QComboBox()
        self.Combo2 = QComboBox()
        self.Combo3 = QComboBox()

        self.Combo00 = QComboBox()
        self.Combo01 = QComboBox()
        self.Combo02 = QComboBox()
        self.Combo03 = QComboBox()

        self.Combo00.addItems(['Plot', 'Export'])
        self.Combo01.addItems(['True', 'False'])
        self.Combo02.addItems(['True', 'False'])

        self.Combo0.currentIndexChanged.connect(self.Combo0_currentIndexChanged)
        self.Combo1.currentIndexChanged.connect(self.Combo1_currentIndexChanged)
        self.Combo2.currentIndexChanged.connect(self.Combo2_currentIndexChanged)

        # -----------------------------------------------------------------------------
        self.Layout3.addWidget(QLabel('Plot Group'), 0, 0)
        self.Layout3.addWidget(QLabel('Plot Type'), 0, 1)
        self.Layout3.addWidget(QLabel('Settings 1'), 0, 2)
        self.Layout3.addWidget(QLabel('Settings 2'), 0, 3)

        self.Layout3.addWidget(QLabel('Plot/Export'), 2, 0)
        self.Layout3.addWidget(QLabel('Show Legend'), 2, 1)
        self.Layout3.addWidget(self.lblSettings3, 2, 2)
        self.Layout3.addWidget(QLabel('Fitting'), 2, 3)

        self.Layout3.addWidget(self.Combo0, 1, 0)
        self.Layout3.addWidget(self.Combo1, 1, 1)
        self.Layout3.addWidget(self.Combo2, 1, 2)
        self.Layout3.addWidget(self.Combo3, 1, 3)

        self.Layout3.addWidget(self.Combo00, 4, 0)
        self.Layout3.addWidget(self.Combo01, 4, 1)
        self.Layout3.addWidget(self.Combo02, 4, 2)
        self.Layout3.addWidget(self.Combo03, 4, 3)

        fill_combo(self.Combo0, COMBO_0)

        # ----------------------------------------------------------------------------
        self.LOADED_DATASET = None
        self.ActivePG = None
        self.PileButtons = None
```

```
        self.SimName = None

        self.PlotSets = self.__load_plots()

        # Databases..............................................................
        self.DB_Piles = load_object(DB1_FN)
        self.DB_Group = load_object(DB2_FN)
        self.AllLDC = load_object(ALL_LDC_FN)

    @staticmethod
    def __parse_xl(ws, r_start, columns, check_c=1):

        plot_sets = []

        check = ws.cell(row=r_start, column=check_c).value

        while check is not None:
            plot_sets.append(get_xl_row(ws, (r_start, 1), columns))
            r_start += 1

            check = ws.cell(row=r_start, column=check_c).value

        return plot_sets

    def __load_plots(self):

        try:
            wb = xl.load_workbook(PLOTS_1)

            plots = self.__parse_xl(wb['Plots'], 2, 9)
            settings = self.__parse_xl(wb['Settings'], 2, 3)

            return plots, settings

        except FileNotFoundError:
            popup_message('Plots File Not Found',
            QMessageBox.Critical, 'File Error', QMessageBox.Ok)

        except zipfile.BadZipFile:
            popup_message('Invalid File', QMessageBox.Critical, 'File Error',
             QMessageBox.Ok)

    def __on_off_widgets(self, value):
        self.cmdShowPG.setEnabled(value)
        self.cmdDataTable.setEnabled(value)

    def cmdOpenDataset_clicked(self):

        ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
        'Dataset files (*.dat)')

        if ds_filename != '':
```

```
        try:
            self.commands_window.setWindowTitle(ds_filename)
            self.LOADED_DATASET = load_object(ds_filename)
            self.LOADED_DATASET.CurrentPath = ds_filename

            self.label_1.setText(self.LOADED_DATASET.SummaryTable[0][1])
            self.label_2.setText(self.LOADED_DATASET.SummaryTable[0][3])

            # Fill Simulations Combo Box
            fill_combo(self.cmbSims, [item[0]
            for item in self.LOADED_DATASET.SummaryTable[1][1]])

            self.__on_off_widgets(True)       # Activate needed buttons

            if BATCH_FULL:
                self.full_batch(BATCH_FULL_NORMALIZED)

        except ImportError:
            popup_message('Invalid File', QMessageBox.Critical, 'File Error',
             QMessageBox.Ok)

        except PermissionError:
            popup_message('Permission Error. File already opened. Close it and
             repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

        except AttributeError:
            popup_message('Invalid File', QMessageBox.Critical, 'File Error',
             QMessageBox.Ok)

def cmdDataTable_clicked(self):

    data = (self.LOADED_DATASET.SummaryTable[1],
     self.LOADED_DATASET.SummaryTable[2])

    popup_simple_dialog('Data Table', [[SummaryTableDS(data[1][1], data[1][0],
     [item[0] for item in data[0][1]])]])

def cmdShowPG_clicked(self):

    s_i = self.cmbSims.currentIndex()
    raw_results = self.LOADED_DATASET.SummaryTable_4[s_i][1]

    if raw_results is not None:

        self.__create_pg(s_i, raw_results, SOIL_TYPES[s_i])

        self.PileButtons = []
        for r in range(0, self.ActivePG.N):
            for c in range(0, self.ActivePG.M):
                new_button = QPushButton('Pile-' + str(r*self.ActivePG.M+c))
                new_button.clicked.connect(self.pile_clicked)
                self.PileButtons.append(new_button)
                self.Layout2.addWidget(new_button, self.ActivePG.M-1-c, r)

        self.cmdPGResults.setEnabled(True)
        self.cmdPGResults_All.setEnabled(True)
        self.SimName = self.cmbSims.currentText()

    else:
        popup_message('No Results Found', QMessageBox.Warning, 'No Results',
         QMessageBox.Ok)

def Combo0_currentIndexChanged(self):

    i0 = self.Combo0.currentIndex()
    fill_combo(self.Combo1, COMBO_1[i0])

    # Adjust Settings Combos
    if i0 == 0:
        show_notice(self.lblSettings3, 'Normalize Z axis')
        self.Combo03.clear()
    else:
        show_notice(self.lblSettings3, 'Normalize Displacements')
        fill_combo(self.Combo03, ['Polyfit', 'Spline'])

def Combo1_currentIndexChanged(self):
    i0 = self.Combo0.currentIndex()
    i1 = self.Combo1.currentIndex()

    fill_combo(self.Combo2, COMBO_2[i0][i1])

def Combo2_currentIndexChanged(self):
    i0 = self.Combo0.currentIndex()
    i1 = self.Combo1.currentIndex()
    i2 = self.Combo2.currentIndex()

    combo_3_list = []

    if i0 == 2:
        combo_3_list = COMBO_3_2[i1][i2]

    fill_combo(self.Combo3, combo_3_list)

def __combo_settings(self):

    plot_export = self.Combo00.currentText()
    show_legend = Combo_Translate[self.Combo01.currentText()]
    normalized = Combo_Translate[self.Combo02.currentText()]

    plot_type = self.Combo0.currentIndex()

    if plot_type == 1:
        p1 = Combo_Translate[self.Combo03.currentText()]
        p2 = Combo_Translate[self.Combo1.currentText()]
        p3 = Combo_Translate[self.Combo2.currentText()]
        p4 = None

    elif plot_type == 0:
        p1 = self.Combo1.currentText()
        p2 = Combo_Translate[self.Combo2.currentText()]
        p3 = INDICES[self.Combo1.currentIndex()][self.Combo2.currentIndex()]
        p4 = None

    else:
        p1 = Combo_Translate[self.Combo1.currentText()]
        p2 = self.Combo2.currentText()
        p3 = self.Combo3.currentText()
        p4 = Combo_Translate[self.Combo03.currentText()]

    return plot_type, p1, p2, p3, p4, plot_export, show_legend, normalized

def pile_clicked(self):

    pile_index = 0
    for pile_index, button in enumerate(self.PileButtons):
        if self.sender() == button:
            break
            # Pile index found

    plot_type, p1, p2, p3, p4, p_exp, show_legend, normalized =
     self.__combo_settings()

    if plot_type < 2:
        self.__plot_caller(plot_type, pile_index, p1, p2, p3, p4, p_exp,
         show_legend, normalized, 0, self.SimName)

def plot_pg_results(self):

    plot_type, p1, p2, p3, p4, plot_export, show_legend, normalized =
     self.__combo_settings()

    if plot_type == 2:
        self.__plot_caller(plot_type, 0, p1, p2, p3, p4, plot_export, show_legend,
         normalized, 0, self.SimName)

def __plot_caller(self, plot_type, pile_id, p1, p2, p3, p4, plot_export, leg,
 normalized, t1, sim_name):

    pile = self.ActivePG.Piles[pile_id]

    if plot_type == 0:
        return pile.RAW_PLOT(p1, p3, pile.get_steps(BATCH_STEPS), p2, leg,
         plot_export, normalized, sim_name, t1)

    elif plot_type == 1:
        return pile.LDC_PY_PLOT(p2, p1, p3, leg, plot_export, normalized,
         sim_name, t1)
    else:
        return self.ActivePG.PG_PLOT(p2, p3, p1, p4, leg, plot_export, normalized,
         sim_name, t1)

@staticmethod
def __translate_plot_set(plot_set):

    plot_id, plot_type, caption, i1, lp_if_ldc_gw, curve_parameter, prg, t_a,
     p_s_c = plot_set

    if plot_type == 0:
        return caption, p_s_c, i1, None, plot_type

    elif plot_type == 1:
        return p_s_c, lp_if_ldc_gw, curve_parameter, None, plot_type

    else:
        return lp_if_ldc_gw, prg, t_a, p_s_c, plot_type

# BATCH PROCESSING FUNCTIONS #####################################################

def pile_pg_results_batch(self):

    all_plots, all_settings = self.PlotSets

    settings_list = [all_settings[j] for j in BATCH_SETTINGS]
    plot_list = [all_plots[j] for j in BATCH_PLOTS]

    for pile_id in BATCH_PILES:
        for settings in settings_list:
            for plot_set in plot_list:

                p1, p2, p3, p4, plot_type = self.__translate_plot_set(plot_set)
                leg, p_exp, norm = settings

                self.__plot_caller(plot_type, pile_id, p1, p2, p3, p4, p_exp, leg,
                 norm, 0, self.SimName)

def __create_pg(self, sim_index, results, soil):

    if ADD_SP_TRIGGER:
        pile_index = SP_LINKER[sim_index]
        single_pile = SINGLE_PILES[pile_index]
        print('Equivalent single pile ', pile_index, ' added to PG simulation ',
         sim_index)

    else:
        single_pile = None

    self.ActivePG = PGModel_PLAXIS(self.LOADED_DATASET.SummaryTable_4[sim_index
    ][0], results, single_pile, soil)

    self.ActivePG.set_fixed_results(P_S_FINAL)
    # Extract fixed results for PG (DB_Results123)

def full_batch(self, norm):

    DATASET_DATA = []

    sim_names = [item[0] for item in self.LOADED_DATASET.SummaryTable[1][1]]
    sim_names = [sim_names[i] for i in SIM_FILTER]

    # Cycle all dataset simulations and collect data if exists
    for s_i, sim_n in zip(SIM_FILTER, sim_names):

        SIM_DATA = []
        soil_type = SOIL_TYPES[s_i]
        raw_results = self.LOADED_DATASET.SummaryTable_4[s_i][1]

        if raw_results is not None:
            print('*** Results Found. Processing Simulation: ', sim_n, ' ***')
            self.__create_pg(s_i, raw_results, soil_type)

            # ##########################$$$$$$$###################################
            # OBTAIN DATA FROM PILE GROUP AND INPUT IT IN ALREADY PREPARED
             DATABASE

            # Temporary extract LDC for every pile for checking
            self.AllLDC += self.ActivePG.get_all_ldc()

            # All PG important data is extracted at this point.
            #
            # Fill the databases
            # DB parameters based on the input vector data ---
            defines position of the results in the DB

            soil_type, bc, load_angle, sx, n, sy, m = self.ActivePG.get_basics()

            pos0, pos1, pos2, pos3, pos5, pos6, pos7 = pos_db(soil_type, bc,
             load_angle, sx, sy, n, m)

            # Fill the database #################################################
            piles_results, group_results = self.ActivePG.db_results(P_S_FINAL)

            # Cycle each pile results and write it to database
            for r in range(0, n):
                for c in range(0, m):

                    self.DB_Piles[pos0, pos1, pos2, pos3, :, r, pos6, c, :] =
                     piles_results[:, r*m + c, :]

            self.DB_Group[pos0, pos1, pos2, pos3, :, pos5, pos6, pos7] =
             group_results
            save_databases([(self.DB_Piles, DB1_FN), (self.DB_Group, DB2_FN),
             (self.AllLDC, ALL_LDC_FN)])
            # ######################################################################
            for pile_i in BATCH_PILES:
                PILE_DATA = []

                for j in BATCH_PLOTS:
                    p1, p2, p3, p4, plot_type =
                     self.__translate_plot_set(self.PlotSets[0][j])

                    data = self.__plot_caller(plot_type, pile_i, p1, p2, p3, p4,
                     'Export', False, norm, 1, sim_n)

                    PILE_DATA.append((pile_i, j, data))

                SIM_DATA.append(PILE_DATA)
        else:
            print('*** No Results ***')

        DATASET_DATA.append(SIM_DATA)

    # ##########################################################################
    # Reorganize dataset before excel export ------------------------------------
    DATASET_DATA_1 = []

    for j, plot_id in enumerate(BATCH_PLOTS):
        PLOT_DATA = []

        for s_i, sim_data_set in enumerate(DATASET_DATA):

            PILE_DATA = []
            if not sim_data_set == []:      # Results exists

                for pile_data_set in sim_data_set:
                    PILE_DATA.append(pile_data_set[j])

            PLOT_DATA.append(PILE_DATA)

        DATASET_DATA_1.append((plot_id, PLOT_DATA))

    # Create xl wb FOR EACH PROCESSED PILE (Case 0, 1).
    Case 2 can use single pile index, and data goes together

    print('*****************************************************************')
    print('Creating excel file...')
    print('')

    for j1, p_i in enumerate(BATCH_PILES):
        self.batch_pile_wb(sim_names, j1, p_i, DATASET_DATA_1, norm)
        print('Workbook successfully created!')

    print('*****************************************************************')
    print('')

def batch_pile_wb(self, sim_names, j1, p_i, ds_data_1, norm):
        # Create xlsx for each pile. Create tab for each plot type.
        # Add data from all simulations in dataset.

        wb = xl.Workbook()

        wb_name = self.LOADED_DATASET.get_name() + '_Pile_' + str(p_i) + '.xlsx'

        for j2, pt_i in enumerate(BATCH_PLOTS):
        # Cycle plot types. Add data to appropriate TABSHEET

            sheet_title = 'Plot_' + str(pt_i)
            wb.create_sheet(sheet_title)

            ws = wb.worksheets[j2 + 1]

            if BATCH_CHART_CREATE:
                new_chart = xl_scatter_chart(sheet_title, BATCH_CHART_STYLE, 'X', 'Y')

            start_c = 1
            for j3, simulation_data in enumerate(ds_data_1[j2][1]):
            # Cycle data sim by sim and add it to ws

                if not simulation_data == []:

                    xy_diagram = simulation_data[j1][2]

                    xy_diagram.fill_xl_ws(norm, sim_names[j3], ws, 1, start_c, 2)

                    added_columns = len(xy_diagram.Plots) * 2

                    if BATCH_CHART_CREATE:
                        for series in xy_diagram.create_xl_series(ws, 5, start_c, 2,
                         j3):
                            new_chart.series.append(series)

                    start_c += added_columns + 1
                    # Split simulations by empty column

            ws.add_chart(new_chart, 'A1')

        wb.remove_sheet(wb.worksheets[0])
        wb.save(wb_name)


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Equivalent Single Pile Processor `EQSP_Process.py`

```
import sys

from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QApplication, QMdiArea, QMessageBox, QLabel, QPushButton,
 QMainWindow, QWidget

from PyQt5.QtWidgets import QMdiSubWindow, QGridLayout

from M1_WidgetClasses import SummaryTableDS
from Main_Functions import popup_open_save, load_object, popup_message,
 popup_simple_dialog, save_object

from Main_ConfigConstants import DefaultSimDir
from PG_ModelClasses import Pile, unpack_input
from PG_Config import SINGLE_PILES_FN

SIM_FILTER = [x for x in range(0, 2)] # Enter simulation indices containing EQSP


def new_subwindow(title, dimensions):

    new_window = QMdiSubWindow()
    top_widget = QWidget()
    new_layout = QGridLayout(top_widget)
    top_widget.setLayout(new_layout)
    new_window.setWindowTitle(title)
    new_window.setFixedSize(dimensions[0], dimensions[1])
    new_window.setWindowFlags(Qt.CustomizeWindowHint)
    new_window.setWidget(top_widget)
    return new_window, new_layout


class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)

        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)
        self.setWindowTitle('Equivalent Single Pile Processor')

        self.commands_window, self.Layout1 = new_subwindow('Dataset Not Loaded',
        (400, 400))

        self.mdi.addSubWindow(self.commands_window)
        self.commands_window.show()

        cmdOpenDataset = QPushButton('LOAD DATASET')
        self.cmdDataTable = QPushButton('SHOW INPUT PARAMETERS')
        self.cmdProcessSP = QPushButton('PROCESS SINGLE PILE SIMS')
        self.label_1 = QLabel('***DATASET NOT LOADED***')
        self.label_2 = QLabel('***DATASET NOT LOADED***')

        cmdOpenDataset.clicked.connect(self.cmdOpenDataset_clicked)
        self.cmdDataTable.clicked.connect(self.cmdDataTable_clicked)
        self.cmdProcessSP.clicked.connect(self.cmdProcessSP_clicked)

        self.Layout1.addWidget(QLabel('Dataset Description'), 0, 0)
        self.Layout1.addWidget(self.label_1, 0, 1)
        self.Layout1.addWidget(QLabel('Total Number of Simulations'), 1, 0)
        self.Layout1.addWidget(self.label_2, 1, 1)
        self.Layout1.addWidget(cmdOpenDataset, 2, 0)
        self.Layout1.addWidget(self.cmdDataTable, 2, 1)
        self.Layout1.addWidget(self.cmdProcessSP, 3, 0)

        self.cmdProcessSP.setEnabled(False)
        self.cmdDataTable.setEnabled(False)

        self.LOADED_DATASET = None

    def cmdOpenDataset_clicked(self):

        ds_filename = popup_open_save('Open', DefaultSimDir, 'Open Dataset',
        'Dataset files (*.dat)')

        if ds_filename != '':

            try:
                self.commands_window.setWindowTitle(ds_filename)
                self.LOADED_DATASET = load_object(ds_filename)
                self.LOADED_DATASET.CurrentPath = ds_filename

                self.label_1.setText(self.LOADED_DATASET.SummaryTable[0][1])
                self.label_2.setText(self.LOADED_DATASET.SummaryTable[0][3])

                self.cmdProcessSP.setEnabled(True)
                self.cmdDataTable.setEnabled(True)

            except ImportError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

            except PermissionError:
                popup_message('Permission Error. File already opened. Close it and
                 repeat.', QMessageBox.Critical, 'File Error', QMessageBox.Ok)

            except AttributeError:
                popup_message('Invalid File', QMessageBox.Critical, 'File Error',
                 QMessageBox.Ok)

    def cmdDataTable_clicked(self):
        data = (self.LOADED_DATASET.SummaryTable[1],
         self.LOADED_DATASET.SummaryTable[2])

        popup_simple_dialog('Data Table', [[SummaryTableDS(data[1][1], data[1][0],
         [item[0] for item in data[0][1]])]])

    def cmdProcessSP_clicked(self):
        Processed_Piles = []

        for sim_index in SIM_FILTER:

            input_vector, results_table =
             self.LOADED_DATASET.SummaryTable_4[sim_index]

            if results_table is not None:

                d, l1, l2, E_pile, cap_type, cap_thick =
                unpack_input(input_vector, 'SINGLE')

                pile = Pile(d, l1, l2, E_pile, cap_type, cap_thick)
                pile.set_alone()
                pile.process_raw_results(results_table)

                Processed_Piles.append(pile)

            else:
                Processed_Piles.append(None)

            print('Processed Single Pile Simulation: ', sim_index)

        # All piles are processed. Save list of processed piles
        save_object(Processed_Piles, SINGLE_PILES_FN)

        print('')
        print('--------------------------------------------------')
        print('Single Pile object instances sucessfully saved')


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Results Database Maker `PG_DB.py`

```
import openpyxl as xl

from Main_Functions import load_object, xl_scatter_chart, fill_xl_row_col,
 xy_series_xl
```

```
from PG_Config_DB import *


def xy_xl_fill_basic(ws, x_data, y_data, plot_i, start_r, start_c):

    fill_xl_row_col(ws, 'R', [plot_i, ''], start=(start_r, start_c))
    fill_xl_row_col(ws, 'R', ['X', 'Y'], start=(start_r + 1, start_c))

    fill_xl_row_col(ws, 'C', x_data, start=(start_r + 2, start_c))
    fill_xl_row_col(ws, 'C', y_data, start=(start_r + 2, start_c+1))


SQDB_Piles = load_object(DB1_SQ_FN)
SQDB_Group = load_object(DB2_SQ_FN)
AllLDC = load_object(ALL_LDC_FN)

# Setup ##############################################################################
DB1_TRIGGER = True
DB2_TRIGGER = True
DB3_TRIGGER = True

SOIL_FILTER = ['Dense', 'Loose']
BC_FILTER = ['Free']
NM_FILTER = [2, 3, 4]
S_FILTER = [2, 3, 4, 5]
RESULTS_FILTER = ['LP', 'IF', 'Mmax']

MAX_SERIES3 = 50
# ####################################################################################

if DB1_TRIGGER:

    X1_data = DB_SQ_ANGLE[:]

    for soil in SOIL_FILTER:

        for bc in BC_FILTER:

            for results_type in RESULTS_FILTER:

                for n_m in NM_FILTER:

                    list_index = DB_N_M.index(n_m)

                    pos0 = DB_SOILS.index(soil)
                    pos1 = DB_BC.index(bc)
                    pos6 = DB_SQ_PILES_RESULTS.index(results_type)

                    filename1 = os.getcwd() + '\\DB1_' + soil + '_' + bc + '_' +
                     str(n_m) + 'x' + str(n_m) + '_' + results_type + '.xlsx'

                    wb1 = xl.Workbook()
                    # Create workbook and add all empty worksheets for different y/D

                    for disp_level_norm in Disp_Fixed_Getters:
                        wb1.create_sheet('yD = ' + str(disp_level_norm))

                    wb1.remove_sheet(wb1.worksheets[0])

                    for pos4, disp_level_norm in enumerate(Disp_Fixed_Getters):

                        ws1 = wb1.worksheets[pos4]
                        start_col = 1

                        for s in S_FILTER:
                            pos3 = DB_SQ_S.index(s)

                            new_chart = xl_scatter_chart('s/D = ' + str(s), 20,
                             'Loading Angle [degrees]', results_type)

                            for pos5 in range(0, n_m**2):

                                Y1_data = SQDB_Piles[list_index][pos0, pos1, :, pos3,
                                 pos4, pos5, pos6]

                                xy_xl_fill_basic(ws1, X1_data, Y1_data, 'Pile_' +
                                 str(pos5), 1, start_col)

                                new_chart.series.append(xy_series_xl(ws1, 3,
                                 start_col, len(X1_data), 'Pile_' + str(pos5),
                                  'triangle'))

                                start_col += 2

                            ws1.add_chart(new_chart)

                    wb1.save(filename1)

                    print('Workbook ' + filename1 + ' saved.')

    print('------------------------------------------------------------------------')
    print('Database 1 processed')


if DB2_TRIGGER:

    X2_data = DB_SQ_ANGLE[:]

    for soil in SOIL_FILTER:

        for bc in BC_FILTER:

            for n_m in NM_FILTER:

                pos0 = DB_SOILS.index(soil)
                pos1 = DB_BC.index(bc)
                pos5 = DB_N_M.index(n_m)

                filename2 = os.getcwd() + '\\DB2_' + soil + '_' + bc + '_' + str(n_m)
                 + 'x' + str(n_m) + '.xlsx'

                wb2 = xl.Workbook()
                ws2 = wb2.worksheets[0]
                start_col = 1

                for pos4, disp_level_norm in enumerate(Disp_Fixed_Getters):
                # Create chart for each displacement level

                    new_chart = xl_scatter_chart('y/D = ' + str(disp_level_norm),
                                                 12, 'Loading Angle [degrees]',
                                                 'GW [-]')

                    for s in S_FILTER:
                        pos3 = DB_SQ_S.index(s)

                        series_title = 's/D = ' + str(s)

                        Y2_data = SQDB_Group[pos0, pos1, :, pos3, pos4, pos5]
                        # Filter database and create plot xlsx

                        xy_xl_fill_basic(ws2, X2_data, Y2_data, series_title, 1,
                         start_col)

                        new_chart.series.append(xy_series_xl(ws2, 3, start_col,
                         len(X2_data), series_title, 'triangle'))

                        start_col += 2

                    ws2.add_chart(new_chart)

                wb2.save(filename2)
                print('Workbook ' + filename2 + ' saved.')
    print('--------------------------------------------------------------------------')
    print('Database 2 processed')


if DB3_TRIGGER:
    print('Processing Database 3')

    X3_data = Disp_Fixed_Getters[:]

    wb3 = xl.Workbook()
    ws3 = wb3.worksheets[0]

    full_plots, last_plot = divmod(len(AllLDC), MAX_SERIES3)
    splitters = [MAX_SERIES3]*full_plots + [last_plot]

    start_curve, start_col = 0, 1
```

```
# Create Excel Workbook with diagrams using all data from database 3
for plot_index, splitter in enumerate(splitters):

    new_chart = xl_scatter_chart('Plot ' + str(plot_index), 12, 'y/D [-]',
     'Horizontal Loading')

    for curve_index in range(start_curve, start_curve+splitter):

        Y_data = AllLDC[curve_index]

        xy_xl_fill_basic(ws3, X3_data, Y_data, curve_index, 1, start_col)

        xy_chart_series = xy_series_xl(ws3, 3, start_col, len(X3_data),
         str(curve_index), 'triangle')

        new_chart.series.append(xy_chart_series)

        start_col += 2

    ws3.add_chart(new_chart)

    start_curve += splitter

wb3.save(ALL_LDC_XL_FN)
print('workbook 3 saved.')
```

Изјава о ауторству

Милош С. Марјановић

Број индекса 902/2010.

Изјављујем

да је докторска дисертација под насловом

ANALYSIS OF INTERACTION INSIDE THE PILE GROUP SUBJECTED TO ARBITRARY HORIZONTAL LOADING

- резултат сопственог истраживачког рада,
- да дисертација у целини ни у деловима није била предложена за стицање друге дипломе према студијским програмима других високошколских установа
- да су резултати коректно наведени и
- да нисам кршио ауторска права и користио интелектуалну својину других лица.

Потпис аутора

У Београду, јул 2020.

Милош Марјановић

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора	Милош Марјановић, мастер инж. грађ.
Број индекса	902/2010.
Студијски програм	Грађевинарство
Наслов рада	ANALYSIS OF INTERACTION INSIDE THE PILE GROUP SUBJECTED TO ARBITRARY HORIZONTAL LOADING
Ментори	Проф. др Мирјана Вукићевић Универзитет у Београду, Грађевински факултет Dr.-Ing. Diethard König (AkadOR) Ruhr University Bochum, Chair of Soil Mechanics, Foundation Engineering and Environmental Geotechnics

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао ради похрањивања у **Дигиталном репозиторијуму Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског назива доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис аутора

У Београду, јул 2020.

Милош Марјановић

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

ANALYSIS OF INTERACTION INSIDE THE PILE GROUP SUBJECTED TO ARBITRARY HORIZONTAL LOADING

која је моје ауторско дело.

Дисертацију са свим прилозима предао сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигиталном репозиторијуму Универзитета у Београду и доступну у отвореном приступу могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио.

1. Ауторство (CC BY)

2. Ауторство – некомерцијално (CC BY-NC)

3. Ауторство – некомерцијално – без прерада (CC BY-NC-ND)

[✓ 4.] Ауторство – некомерцијално – делити под истим условима (CC BY-NC-SA)

5. Ауторство – без прерада (CC BY-ND)

6. Ауторство – делити под истим условима (CC BY-SA)

(Молимо да заокружите само једну од шест понуђених лиценци.

Кратак опис лиценци је саставни део ове изјаве).

Потпис аутора

У Београду, јул 2020.

[illegible]

1. Ауторство. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прерада. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

[✓ **4.**] **Ауторство - некомерцијално – делити под истим условима.** Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прерада. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.