NTNU
Norwegian University of
Science and Technology

Extending a derivative-free model-based trust-region optimization algorithm to account for constraints and partial gradient information

Application to oil field development

Joakim Rostrup Andersen

Master of Science in Cybernetics and Robotics
Submission date: July 2018
Supervisor: Morten Hovd, ITK
Co-supervisor: Mathias Bellout, ITK
Andres Duarte Codas, IBM Research, Brazil

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

This master's thesis is motivated by the oil field development challenge. This is a very big and complicated task and only one part of it will be in focus, namely, the well placement challenge. When planning a new oil field (e.g., in the North sea) or when more wells are being added to an existing field, then the placement of the wells are crucial. If the wells have been placed wisely, then the total amount of recovered oil may be improved considerably. In addition, it is preferable to not produce (i.e, extract from the reservoir) water. Produced water must be cleaned before it is released back into the ocean and there is a limitation on how much water that can be processed at once. To help aid in the decision of the placement of the wells, an oil reservoir simulator can be used. Collected data from the real field are fed into the simulator, and this simulator can be used in the decision making process.

This task can be viewed upon as mathematical programming: there is an input (the well location), a function (the simulator) and an output (e.g., a number that represents the value of the accumulated production of oil, gas and water).

This master's thesis addresses the challenge of finding a minimum of an unknown function. The function may be either an ordinary mathematical function, or a function whose value depends on the output of a simulation. The available information, when searching for the minimum, is only the function evaluations. The first step towards finding the minimum, is to sample the unknown function and create a model of the relationship between the inputs and the outputs. The model will be valid within a region which is known as the trust-region. The second step, involves minimizing the model, and hopefully this minimum will lead towards a minimum point of the unknown function. New points around the newly found point will be needed to create a new model that is valid within the new trust-region. The trust-region is centered at the newly found point. These steps are repeated until a termination criterion is satisfied.

This type of method is called a derivative-free model-based trust-region because the optimization is done on a model only trusted within a specific region, and the unknown function is not differentiated.

In addition to learn the theory for the given method, two extensions are made. For most real-life usages, the user would need to be able to impose constraints on the decision variables. There exists several different approach, which each has their different pros and cons. The selected method makes sure that all the points that are to be evaluated will always obey the constraints. This is imposed by adding all the user defined constraints into the minimization of the model within the trust-region. This means that the area where a minimum is searched for will be smaller. If there are constraints on some of the outputs of the unknown function, then these must be handled differently. They are modelled in the

same way as the unknown function, and these constraint models can be included into the minimization of the model of the unknown function.

Adding these constraints makes it harder to find the minimum of the model. Therefore a Sequential Quadratic Programming software is used to solve this task. The given constraint handling technique for input constraints has been implemented and the preliminary results are satisfying.

The second extension is concerned with the possibility of using less sample points to create the model. Two different approaches have been explored. The first approach is to use the old model as an approximation to the new model and perform some minimization of the difference after the interpolation conditions have been satisfied. The interpolation conditions will make sure that the model provides the same output as the unknown functions at the sample points, whereas the minimization will make sure that the model is uniquely defined.

The second approach is concerned with a slightly different scenario. Until now only the function evaluations have been available. If derivatives of the unknown function with respect to some of the variables of interest were available, this information could be used to speed up the model-making process. Simulators often provide this additional type of information. This is included by solving a very similar minimization problem as is done for the first approach. These two approaches can be combined.

Including gradient information into the model-making process makes the algorithm converge faster. I.e., less function evaluations are needed. However, the found local optimum is often worse than the one found without using gradient information.

Sammendrag

Denne masteroppgaven er motivert av oljefeltutviklingsutfordringen. Dette er en veldig stor og komplisert oppgave, og bare en del av denne oppgaven vil være i fokus, nemlig brønnplasseringsutfordringen. Ved planlegging av et nytt oljefelt (f.eks. I Nordsjøen) eller når flere brønner legges til i et eksisterende felt, er plasseringen av brønnene avgjørende. Hvis brønnene har blitt plassert på en god måte, kan den totale mengden utvunnet olje forbedres betraktelig. I tillegg er det foretrukket å ikke produsere (dvs. hende ut fra reservoaret) vann. Produsert vann må rengjøres før det slippes tilbake i havet, og det er en begrensning på hvor mye vann som kan behandles samtidig. For å hjelpe til med beslutningen om plassering av brønnene, kan en oljereservoarsimulator brukes. Innsamlet data fra det virkelige feltet blir matet inn i simulatoren, og denne simulatoren kan brukes i beslutningsprosessen.

Denne oppgaven kan betraktes som matematisk programmering: det er en innputt (brønnplasseringen), en funksjon (simulatoren) og en utputt (f.eks. et tall som representerer verdien av den akkumulerte produksjonen av olje, gass og vann).

Denne masteroppgaven studerer utfordringen i å finne et minimum av en ukjent funksjon. Funksjonen kan enten være en vanlig matematisk funksjon, eller en funksjon hvis verdi avhenger av resultatet av en simulering. Den tilgjengelige informasjonen, når man søker etter et minimum, er kun funksjonsevalueringene. Det første skrittet mot å finne minimumet, er å sample den ukjente funksjonen og lage en modell av forholdet mellom inngangene og utgangen. Modellen vil være gyldig innenfor et område som er kjent som tillitsregionen (engelsk: trust-region). Det andre trinnet innebærer å minimere modellen, og forhåpentligvis vil dette minimumet lede algoritmen mot et minimumspunkt for den ukjente funksjonen. Nye punkter rundt det nylig funnet punktet vil være nødvendig for å opprette en ny modell som er gyldig innenfor den nye tillitsregionen. Tillitsregionen er sentrert på det nylig funnet punktet. Disse trinnene gjentas til et sluttkriterium er oppfylt.

En optimaliseringsmetode av denne typen kalles for en derivasjonsfri modellbasert tillitsregion metode fordi optimaliseringen er utført på en modell som bare er gyldig i en bestemt region, og den ukjente funksjonen er ikke derivert.

I tillegg til å lære teorien for den oppgitte metoden, er to utvidelser lagt til. For de fleste praktiske bruksområder i virkeligheten må brukeren ha mulighet til å legge inn begrensninger på beslutningsvariablene. Det finnes flere forskjellige metoder for å gjøre dette, som hver har sine forskjellige fordeler og ulemper. Den valgte metoden sørger for at alle punktene som skal evalueres av den ukjente funksjonen alltid overholder begrensningene. Dette oppnås ved å legge til alle de brukerdefinerte beskrankningene i minimering av modellen i tillitregionen. Dette betyr at området der et minimum blir søkt etter vil være mindre. Hvis det er begrensninger på noen av utgangene til den ukjente funksjonen, må

disse håndteres annerledes. De er modellert på samme måte som den ukjente funksjonen, og disse begrensningsmodellene kan inkluderes i minimering av modellen av den ukjente funksjonen.

Å legge til disse beskrankningene gjør det vanskeligere å finne minimum av modellen. Derfor brukes en sekvensiell kvadratisk programmeringsprogramvare for å løse denne oppgaven. Den foreslåtte metoden for beskrankninger på innputtene er implementert og de foreløpige resultatene er tilfredsstillende.

Den andre utvidelsen utforsker mulighetene for å bruke færre punkter for å lage modellen. To forskjellige metoder har blitt utforsket. Den første metoden er å bruke den gamle modellen som en approksimasjon til den nye modellen og utføre en minimalisering av forskjellen av de to modellene etter at interpolasjonsbetingelsene er oppfylt. Interpolasjonsbetingelsene vil sørge for at modellen gir samme verdi som den ukjente funksjonen på punktene som er brukt til å lage modellen, mens minimeringen vil sørge for at modellen er unikt definert.

Den andre metoden tar for seg et litt annet scenario. Hittil har bare funksjonsevalueringen vært tilgjengelig. Hvis deriverte av den ukjente funksjonen med hensyn til noen av variablene av interesse var tilgjengelige, kunne denne informasjonen ha blitt brukt til å akselerere modellbyggingsprosessen. Simulatorer gir ofte denne typen tilleggsinformasjon. Denne informasjonen er inkludert ved å løse et veldig lignende minimeringsproblem som ble gjort for den første metoden. Disse to metodene kan kombineres.

Å inkludere gradientinformasjon i modellbyggingsprosessen gjør at algoritmen konvergerer fortere. Dvs., mindre funksjonsevalueringer er nødvendig. Imidlertid er den lokale optimale løsningen ofte verre enn den som ble funnet uten bruk av gradientinformasjon.

Preface

This Master's thesis is a continuation of the specialization project conducted last semester. The main outcome of the specialization project was a collection of important theory, specifically theory on how to update and maintain the surrogate model throughout the optimization procedure. The algorithm of focus was also selected. There was implemented some methods to deal with the model building and updating.

The objective of the Master's thesis is to extend the results of the specialization project as follows:

- Explore further reductions in sample points for constructing the surrogate model. The idea is to incorporate available gradient information into the model-making process. The scenario is that the derivatives of the objective function with respect to some of the variables are available.

- Include constrains into the optimization formulation. We would like to be able to specify both simple bounds on the variable, general linear constraints and nonlinear constraints.

- Implement the algorithm and test it on ordinary mathematical functions for scenarios where we have different combinations of different constraints and different amount of available gradient information. In addition, the algorithm is tested on a problem from the application area. This test is merely to show that the algorithm can be applied in the field of interest.

Due to the nature of this project, some of the theory is based upon the theory from the previous work, but expanded upon to encompass the work in this thesis. Most of the theory has been improved upon, but some of it remains the same. The following list contains more or less unchanged theory: 3.1, 3.3.4, 3.3.5, 3.9.1, 3.9.2, 3.11 and 3.12. The theory in 3.3 was rather complex, thus, I have tried to add more sections to make it more readable and easier to understand. Section 3.12 has been improved upon and some of the theory that was there in the specialization project has been moved into separate sections (such as 3.10 and 3.11).

The algorithm in 3.12.1 have been modified slightly. The optimization algorithm has to be modified because of the constraints. The change is that the gradient of the model is replaced by the gradient of the Lagrangian of the constrained problem. This is, as far as I can see, not suggested in the [1] book where the algorithm is taken from. However, if we don't use this gradient instead, the algorithm doesn't make sense anymore. The reasoning for this change is given in the theory chapter.

The literature review is extended upon. The last 5 paragraphs are new. Most of the theory was found last semester, thus, the literature review this year is almost the same.

The theory in sections 3.4.1 and 3.4.2 are produced by the author, which is the reason why they "lack" references.

The scope and prerequisites

I would like to point out that the scope of the task will be mathematical programming, and that the application area only will be used to explain why the different theory is needed, e.g., why different kinds of constraints are desired. This is a Master's thesis in cybernetics and not petroleum engineering.

I was given a third-party solver to solve the "subproblem" (which is a constrained mathematical programming in the case of constraints). However, at least three weeks was used to get it to work with my application. The problem was that it was compiled with the wrong settings, meaning that there was a mismatch between what FieldOpt[1] and the third-party solver defined as a "double" (I.e., how many bytes that should be used to represent a floating point).

To implement the constraints in FieldOpt is not straightforward. The current constraint handling process is based upon adding penalty terms to the objective function. That means that the optimization algorithm finds a point, and first then the constraints are dealt with. If the point is infeasible, the point will be projected back into the feasible area.

The road taken in this thesis is quite different. The constraints are always included such that only feasible points are produced. The difficulty arises because of how FieldOpt is constructed. The idea in FieldOpt is that the optimization algorithms should be agnostic to the variable type. I.e., they should not care if the variable represents a z-coordinate or the bottom hole pressure (BHP). The order of the elements of the vector of decision variables is random. This also complicates the implementation of our type of algorithm. In contrast to the agnostic philosophy of FieldOpt, we need to know the variable type and we need to know the exact meaning of the variable. E.g., if we want to impose a restriction of the well, we must know which variables corresponds to the heel and the toe of the well. In addition, we would like to scale the variables differently.

Because of these difficulties only one hard coded constraint was included when the oil reservoir simulator was used.

The functionality to extract the gradients are not ready in FieldOpt. However, based upon the results in Chapter 4, this is not a big drawback.

[1] A software that will be explained later.

My coadvisor, Andrés D. Codas, helped me guide my thoughts to come up with the theory in section 3.4.2.

Further, I would like to say that everything has been implemented in C++. The language has the advantage of having probably the fastest run time and the least amount of restrictions. However, it is not a great language for prototyping. A lot of time would have been saved if the framework (i.e., FieldOpt) was written in Python or something similar.

Gratitude

I would like to thank my coadvisors for great help during the last year. Mathias Bellout has been a true motivator throughout this time. His knowledge in the application area has been very valuable for a novice like me. He has provided useful insight into the field development planning. Andrés D. Codas has helped me with both mathematics and implementation issues. Most importantly, he has showed me the exciting world of optimization!

I would also like to thank my advisor, Morten Hovd, for always being responsive and providing clear and constructive answers and feedback.

Thank you, all!

Table of Contents

Abstract **i**

Sammendrag **iii**

Preface **v**

Table of Contents **xi**

List of Tables **xiv**

List of Figures **xvi**

Abbreviations **xvii**

1 Introduction **1**
1.1 Derivative-free optimization 3
1.1.1 Why use derivative-free optimization methods? 5
1.2 FieldOpt: Field Development Optimization Framework 6
1.3 Outline of the report 7

2 Literature Review **9**

3 Theory **17**
3.1 Notations 19
3.2 The surrogate model 19
3.3 Updating of the interpolation model 21
3.3.1 Derivation of the solution of a convex quadratic program 21
3.3.2 Create the Lagrange polynomials 23
3.3.3 Simple updating scheme 24
3.3.4 Sophisticated updating scheme 25
3.3.5 Formulas for shifting the center point 32
3.4 Gradient enhanced interpolation model 34

3.4.1 Set the true gradients at the center point of the model 34
3.4.2 Including the remaining available gradients 37
3.5 Constraint handling 39
3.5.1 Incorporate constraints into the algorithm 40
3.5.2 Constraints in the well-placement challenge 41
3.6 Robustness against noise 44
3.6.1 Scenario: Including gradients 45
3.7 Poisedness - Geometry of the interpolation points 47
3.8 Model improvement algorithm 48
3.9 Solving the subproblem 51
3.9.1 The exact solution 53
3.9.2 Approximate solutions 53
3.9.3 The constrained case 55
3.10 The scaling factor, r 57
3.11 Certifiably fully linear models 58
3.12 The algorithm 58
3.12.1 The derivative-free model-based trust-region algorithm 59
3.12.2 Explanation of the algorithm 61
3.12.3 Comparison with Powell's algorithms 63

4 Testing of the algorithm **67**
4.1 Incorporating constraints 67
4.1.1 Implementation details 67
4.1.2 Test problems for the constraint handling 68
4.1.3 Results - constraint handling 71
4.2 Gradient enhanced models 77
4.2.1 Convex functions 77
4.2.2 Nonconvex function 78
4.2.3 10 dimensional nonconvex function 80
4.3 Testing on an oil reservoir simulator 82
4.3.1 Scaling 83
4.3.2 Results of the well placement challenge 83

5 Conclusion **89**
5.1 Further work 90
5.1.1 Implementation 90
5.1.2 Initialization 90
5.1.3 Optimize the parameters for the application area 91
5.1.4 Scaling 91
5.1.5 Global optimization of subproblem 91
5.1.6 Make it a global algorithm 91
5.1.7 Gradient enhanced models 91

Bibliography **93**

Appendix A - Results of testing the algorithm **A1**

Appendix B - The implementation **B1**

List of Tables

4.1 The table shows the answers of the different cases for both the Matyas function and the Rosenbrock function. The optimums are all global. . . . 71
4.2 The Matyas function. Nonlinear constraint. 72
4.3 The table shows the difference in the amount of function evaluations needed to converge when different parameters are set. The Matyas function, (4.1), was used as test function. In all scenarios, the global optimum was found. 77
4.4 The table shows the amount of function evaluations needed to converge when different parameters are set. The 10 dimensional sphere was used as test function. In all scenarios, the global optimum was found. 78
4.5 The table shows the difference in the amount of function evaluations needed to converge when different parameters are set. The Ackley function was used as test function. In all scenarios, the global optimum was found. . . 78
4.6 The table shows how fragile the gradient enhanced model is to the selection of points. The gradient enhanced model is compared with the regular model. The only difference for for each test is the initial point. The second coordinate is changed from 4.0 to 4.5 as is shown in the y-column. The first coordinate remains unchanged. 79
4.7 The table shows how the algorithm performs on a nonconvex 10 dimensional function. All parameters not mentioned in the table remains unchanged. 82

A1 The Matyas function. No constraints. A11
A2 The Matyas function. Bounds. A11
A3 The Matyas function. Bounds and linear constraint. A11
A4 The Matyas function. Nonlinear constraint. A11
A5 The Matyas function. Bounds and nonlinear nonconvex constraint A11
A6 The Rosenbrock function. No constraints. A12
A7 The Rosenbrock function. Bounds. A12
A8 The Rosenbrock function. Bounds and linear constraint. A12
A9 The Rosenbrock function. Nonlinear constraint. A12

A10 The Rosenbrock function. Bounds and nonlinear nonconvex function. . . A12

List of Figures

1.1 World's total primary energy supply by fuel type. "Other" includes geothermal, solar, wind, tide/wave/ocean and heat. [2]. 1
1.2 Outlook for world total primary energy supply (TPES) to 2040. The values are in million tonnes of oil equivalent (Mtoe). [2]. 2
1.3 The four steps of the well index calculator in FieldOpt. 6

3.1 Overview of the different step in the derivative-free model-based trust-region optimization method. This diagram is based upon Algorithm 11.2 in Introduction to derivative-free optimization[1]. 17
3.2 An illustration of the difference between the nonconvex and the convex minimum distance constraint. We have assumed that we keep $\hat{\mathbf{x}}_1$ and only move $\hat{\mathbf{x}}_2$. 42
3.3 Linearization of the Euclidean norm in 2D. The green polygon is the one of interest, as it only contains feasible points. The image is taken from [3] 43
3.4 The Ackley function with multiple local optimums. 44
3.5 The Ackley function plotted together with the interpolation model. The circle represents the boundary of the trust-region. The red lines are the sample points. Center point is $[0, 0]$ and the trust-region radius is 5. $m = 2n + 1 = 5$. 45
3.6 The Ackley function plotted together with the interpolation model. The circle represents the boundary of the trust-region. The red lines are the sample points. Center point is $[2.1, 3]$ and the trust-region radius is 5. $m = 2n + 1 = 5$ 46
3.7 The red circles are sample points. The direction of the gradients are drawn, the size of the arrow is not representative. The figures illustrates why including gradient information might be a very bad idea. 47
3.8 The figures show how the poisedness is improved by using Algorithm 1. The red crosses are the points that have changed from one iteration to the next. "Iteration 1" is the initial set of points, i.e., $\mathbf{Y}_1$. "Iteration 2" is the points in $\mathbf{Y}_2$, and so on. 52

4.1 In the figures above, the Matyas function is plotted. The darker the blue color is, the lower the function value is. 69
4.2 In the figures above, the Rosenbrock function is plotted. The darker the blue color is, the lower the function value is. 70
4.3 The top left plot shows the contour of the function together with the constraint. The three other plots shows which points have been evaluated. The difference is the amount of sample points used to create the model, i.e., m. The value is given in the title of each figure. Green mark means final solution. Red mark means point found by solving the subproblem. Black mark is the initial point. The black line is the constraint. 73
4.4 The algorithm converges with all the different constraints. The Matyas function is used and $m = 5$ for all tests. 75
4.5 The algorithm converges with all the different constraints. The Rosenbrock function is used and $m = 5$ for all tests. 76
4.6 The 2 dimensional function of (4.5). I.e., $n = 2$. The figure illustrates that the this function is highly nonlinear and nonconvex. 81
4.7 The reservoir which has been used for testing. Red indicates oil, brown indicates gas and blue indicates water 85
4.8 The initial and optimized placement of the well. The fault (crack) of the reservoir can be used as reference point for comparison. 86
4.9 Key information for the base case and the optimized case. 87
4.10 The average pressure value. The important thing to note is that it remains stable. 88

A1 The points evaluated during optimizations runs of the Matyas function without constraints. See Table A1 for more information. A1
A2 The points evaluated during optimizations runs of the Rosenbrock function without constraints. See Table A6 for more information. A2
A3 The points evaluated during optimizations runs of the Matyas function with bounds. See Table A2 for more information. A3
A4 The points evaluated during optimizations runs of the Rosenbrock function with bounds. See Table A7 for more information. A4
A5 The points evaluated during optimizations runs of the Matyas function with bounds and linear constraint. See Table A3 for more information. . . A5
A6 The points evaluated during optimizations runs of the Rosenbrock function with bounds and linear constraint. See Table A8 for more information. . . A6
A7 The points evaluated during optimizations runs of the Matyas function with a nonlinear constraint. See Table A4 for more information. A7
A8 The points evaluated during optimizations runs of the Rosenbrock function with a nonlinear constraint. See Table A9 for more information. A8
A9 The points evaluated during optimizations runs of the Matyas function with a nonlinear nonconvex constraint and bounds. See Table A5 for more information. A9
A10 The points evaluated during optimizations runs of the Rosenbrock function with a nonlinear nonconvex constraint and bounds. See Table A10 for more information. A10

Abbreviations

IEA	=	International Energy Agency
TPES	=	Total Primary Energy Supply
Mtoe	=	Million tonnes of oil equivalent
PCG	=	Petroleum Cybernetics Group
IGP	=	Department of Geoscience and Petroleum
ITK	=	Department of Engineering Cybernetics
TCGM	=	Truncated Conjugate Gradient Method
FL	=	Fully Linear
CFL	=	Certifiably Fully Linear

Chapter 1

Introduction

There is currently an energy deficit in the world. In 2016, around 1.1 billion people did not have access to electricity[4]. Most of these people are located in developing countries in sub-Saharan Africa and in Asia[4]. The world's population is increasing and there is already an energy deficit. Thus, to help the underdeveloped countries and emerging economies to move forward while trying to reduce the environmental impact, clean and reliable energy sources are needed.

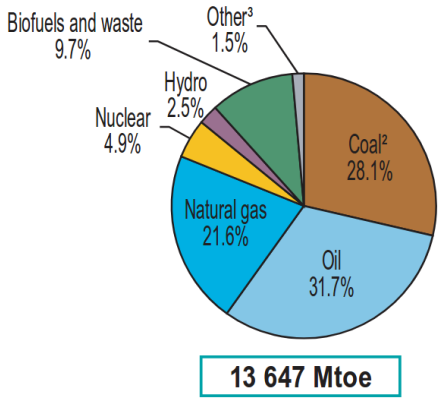


Figure 1.1: World's total primary energy supply by fuel type. "Other" includes geothermal, solar, wind, tide/wave/ocean and heat. [2].

Even though renewable energy sources have gotten more attention in the last decade, it only constituted 1.5 % of the world's total energy supply in 2015[2]. Fossil fuel is the dominant energy source, as can be seen in Figure 1.1. The "Other" part of the pie chart includes geothermal, solar, wind, tide/wave/ocean and heat.

International Energy Agency (IEA) creates different scenarios of the future. The base scenario is the New Policies Scenario. It assumes that policy commitments and plans that have been announced will be followed and obeyed. Another scenario is the 450 Scenario, which takes into account that the global increase in temperature should be no more than 2 degrees Celsius. The goal is to limit the emission of greenhouse gases such that the concentration in the atmosphere will be approximately 450 parts per million of CO2. This scenario serves as an energy pathway for the future.

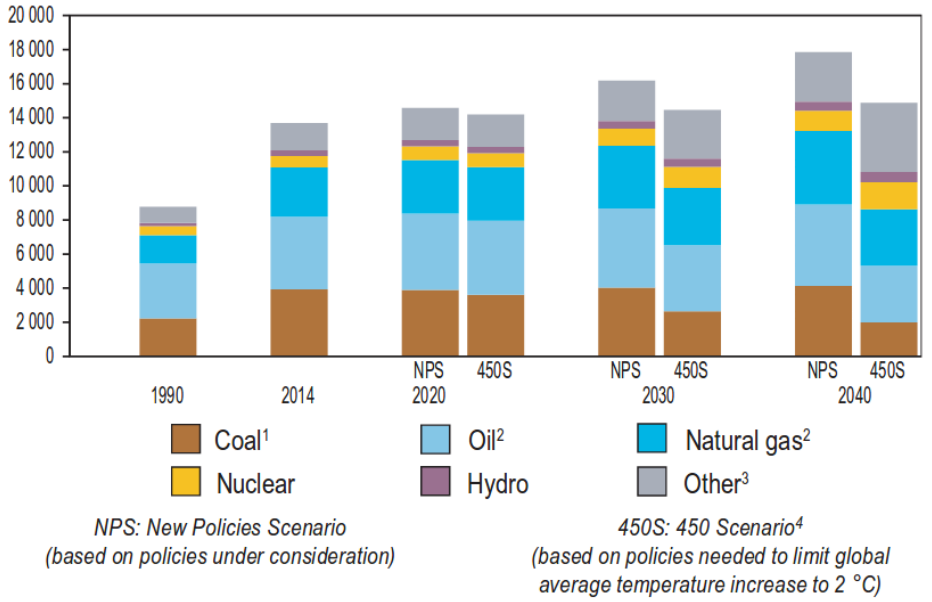


Figure 1.2: Outlook for world total primary energy supply (TPES) to 2040. The values are in million tonnes of oil equivalent (Mtoe). [2].

Figure 1.2 shows the predicted energy sources in both of these scenarios. The amount of energy coming from renewable sources are expected to increase, however, fossil fuel will still be the dominant source of energy for many years. In particular, oil will remain the most important source of energy.

Even though oil as an energy source is not clean, the impact on the environment from extracting it can be reduced in several steps. An oil field's life span and recovery rate[1] can be extended by taking advantage of technology. The expected recovery rate for oil is about 46 % in the Norwegian Continental Shelf, whereas this number is around 22 % globally[5]. Thus, there is a lot more to extract from oil fields when they are shut down.

[1]The recovery rate is a number between 0 and 1. It reflects the expected percentage of how much oil that can be recovered from the reservoir. Both the amount of oil in the reservoir and the amount that can be extracted are expected values. It is not known with certainty. If the recovery rate was 1, then all the predicted oil in the reservoir would have been expected to be extracted.

Increasing the recovery rate will delay the need of stepping into new territories to search for new fields. While planning new fields, decisions regarding the number of wells, types of wells, control of wells and placement of wells are all important. If these and other decisions regarding field development are done in an optimal manner, the recovery rate and life of the field could be increased. In 2011, it was estimated that an increase of 1 % of the recovery rate for fields that were operating on the Norwegian Continental Shelf would give an additional 570 million barrels of oil[5].

Field development planning is a complex and complicated process. Several dependent decisions must be taken which makes it troublesome. Taking smart choices will have a large impact on the expected oil recovery rate. Reservoir simulators may be used to aid in the decision making process. These kind of simulators are computationally expensive (one simulation of one possible configuration may take several hours). Thus, an oil company is likely to use a lot of resources to search for suitable positioning of the wells. Trying different well placement configurations are usually a manual process. This decision taking process is constrained by both physical (e.g., platform location) and practical (e.g., the high cost of running one simulation) limitations. An engineering team will use their expertise to try to find an optimal placement configuration. They use estimates of fluids in place, expert judgment regarding the geology of the field and experience. A mathematical optimization procedure can be used to aid this decision. The algorithm must be given an initial well placement configuration, and it will return an improved configuration. The improvement is measured by, e.g., accumulative oil production. The found solution must the be evaluated by the experts to investigate the viability of the solution. [6]

The focus of this thesis is the well placement configuration. The placement of the wells are discrete variables and gradients of the objective function with respect to such variables are (in general) not readily available. Thus, to solve this problem a derivative-free optimization algorithm will be the main topic.

1.1 Derivative-free optimization

In this section, the problem of this project will be explained, and some basic concepts for the chosen type of derivative-free optimization will be given.

The first task is to find a minimum of "some" function. This function can be both an ordinary mathematical function or a simulator or even a mixture. This function will be referred to as the "*true function*" or the *black-box*. The situation is that the only available information is the function evaluations[2]. Thus, no first or second order derivative information are available.

Finding the global minimum of a possibly highly nonlinear and nonconvex function is in general a very demanding, or maybe even close to impossible, task. However, this is

[2]The scenario when also some derivatives of the function are available will also be explored.

not our goal. The idea is that the user will suggest a starting point, and the algorithm will improve upon this point. As mentioned above, this project is concerned with the placement of wells. This decision will not be taken solely based upon an algorithm. Application expertise is highly important to suggest good initial points and realistic restrictions. Furthermore, the solution must be evaluated by people who have knowledge about field development planning.

The second task will consider almost the same scenario, but with some extra specifications. To find a minimum of a black-box without being able to impose constraints into the optimization problem, might be very useless. Let's say that we would like to improve upon a suggested location of a well. If we are not able to give any constraints whatsoever, the chance of the output being a new suggestion where the well is very, very long is highly likely. This is because a longer well will be able to extract (in general) more oil. However, there are natural limitations on how long a well can be due to limitations in drilling equipment and the cost of drilling. Thus, the second task of this project is to research how to include constraint handling into the chosen derivative-free method.

The last extension of this derivative-free method is to include available derivative information into the model-making process. This is of interest because simulators might provide gradients with respect to some of the variables, but not to all of them. Including such information into the model-making process will do such that less sample points are needed. Or, we could use the same amount of points, but using the gradient information to create even better models.

The chosen algorithm is a derivative-free model-based trust-region method. The different words in the name of the algorithm will now be explained.

The first term is: *model-based*. The knowledge of the relationship between the input and the output of the black-box is limited. In the classical scenario, the only information that can be obtained is the corresponding output given an input. In our extended scenario, we might also have the gradients of the function with respect to some of the variables. Either way, the idea is to try to increase the knowledge by creating a model based upon the information we have available. The model is called a *surrogate model* and we will use a second order polynomial. This is why this method is referred to as a *model-based* procedure.

The constructed model will only be valid close to the area where the sample points are taken. The true function might be highly nonlinear and to expect that the model will be able to mimic the true function far away the points is an unwise idea. The area where the model is expected to represent the true function in a satisfactory manner is called the *trust-region*. This region is defined by a point, a norm (e.g., the Euclidean norm or the infinity norm) and the *trust-region radius*.

The term *derivative-free* comes from the fact that the true function is not, in the classical scenario, differentiated. In one of our scenarios, some of those gradients might actually

be included. In addition, there is no derivation of the surrogate model because it is a second order polynomial and the derivatives are readily available from the representation of the model.

1.1.1 Why use derivative-free optimization methods?

There are many scenarios where derivative-free optimization is the only choice. Due to the wide usage of simulators, the need of good derivative-free methods is still present and increasing. The simulators are (in general) expensive to evaluate because they are solving a set of partial differential equations which controls the underlying physics. The black-box might be be licensed under an open-source license, but gradients may not be readily available for extraction. Implementing it yourself might not be an easy task, and some variables are naturally discrete and defining the gradient with respect to them is not an easy task. If the source code is closed, then derivative-free methods could be the only choice.

Another approach in these scenarios are to use finite-differences. However, there are two cases where this is not practical: if evaluating the true function is expensive (computationally-wise) or when the true function is noisy. If the decision variable is of size n, then (normally) no less than $n + 1$ function evaluations are needed to create one single gradient, which makes it a less preferable approach. In the other case, the created gradients might be utterly useless. If evaluating the function is not too expensive, finite-differences might be a tempting idea. However, if one chooses this path, then one must be sure that the perturbations are of the correct sizes. If the perturbations are too big, then the gradient might be too imprecise. On the other side, if the perturbation is too small it might be lost in the discretization (e.g., the grid size). Meaning that the perturbed vector of variables will give the exact same output as the non-perturbed one. One must also make sure that the perturbation will not be cancelled out by the tolerances throughout the code. [1]

There is a generally accepted (though not proved) statement[1] that derivative-free methods are able to find a "good" local minimum if there are a large number of them. These multiple minima could for example be due to noise. In such scenarios the derivative-free methods have a tendency to go to regions where the true function is in general low, during the initial iterations, because of their near blindness (everything that is in between the sample points are ignored). In later iterations the methods still tend to smooth the true function, which is a valuable property in the case of noise. This "robustness to noise" property will be explored in the theory chapter, and in Chapter 4 it will be tested in practice.

However, if usable gradients are available, then a gradient-based optimization strategy is the approach to take[1].

1.2 FieldOpt: Field Development Optimization Framework

To help solve the well placement configuration problem a framework named *FieldOpt* is used. This is a software developed by the *Petroleum Cybernetics Group (PCG)* at NTNU. This group is a collaboration between the *Department of Geoscience and Petroleum (IGP)* and the *Department of Engineering Cybernetics (ITK)*. The main author is Einar Baumann who is a PhD candidate at IGP, NTNU. FieldOpt is a framework which enables efficient prototyping and testing of mathematical programming techniques within realistic petroleum workflows. It handles everything regarding logging, writing and reading of simulator input and output files, and it manages the scheduling of simulations.

FieldOpt offers a convenient way of comparing different optimization procedures, constraints and reservoir simulators. All of these options can be specified in something called a *driver file*, which contains all of the settings and options for FieldOpt. E.g., to test another reservoir simulator, simply change one word in the driver file[3].

The software also provides two other practical features; parallelization and the well index calculator. Reservoir simulations are often time consuming and running them all one-by-one isn't always preferable. It is remarkably easy to run simulations in parallel within this framework. The only additional workload is that related to keeping track of the IDs of the different simulations. The second feature is the well index calculator. It is common that optimization algorithms specify the placement of a well as a spline (i.e., two points and the line between). However, most reservoir simulators need another parametrization of the wells. Namely, they need to know which cell-blocks the spline passes through and also the well index for each of these blocks. The well index is a proportionality factor that connects the pressure difference between the reservoir and the well, with how much flow one produces from the well. Figure 1.3 shows the different steps of this calculator. First, the well placement is specified by a toe and a heel. Then the endpoints are snapped into valid cells (if they are not already in one). The blocks that the spline is passing through is calculated before the well index is computed for each block.

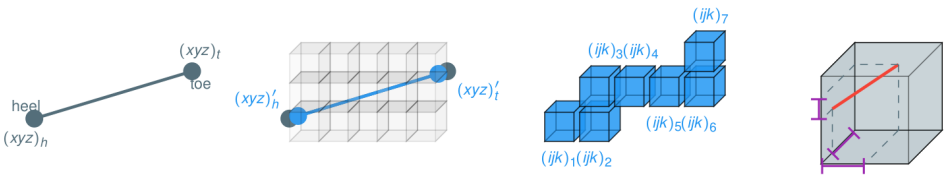


Figure 1.3: The four steps of the well index calculator in FieldOpt.

These given features make FieldOpt highly useful for engineers and researchers who works with optimization of oil reservoirs, but do not have the deep understanding of how

[3] This requires, of course, that the simulator is installed on your system.

oil reservoir simulators work. The task of optimization can be abstracted from the application area and it can be treated as a mathematical problem instead. However, some knowledge of the application area is, of course, still needed, such as knowing that one simulation is expensive (time-wise), knowing what is reasonable step sizes for new iterates of the optimization algorithm, knowing reasonable limitations, etc.

1.3 Outline of the report

The next chapter contains a literature review. In Chapter 3 the theory is given. The focus will be how to build, update and maintain the surrogate model. Further, suggestions on how to include available gradients into the model-making process are given in Section 3.4. Section 3.5 presents different methods to include constraints into the optimization procedure. The somewhat robustness against noise of derivative-free methods are presented in Section 3.6. Sections 3.7 and 3.8 contains theory on how to measure and improve the quality of the geometry of the set of interpolation points. Section 3.9 presents how to solve find a minimum of the model within the trust-region for the unconstrained case, and it also suggests how to solve it when constraints are included. The chosen algorithm is presented and commented in Section 3.12. In Chapter 4 the algorithm has been tested. Different constraints have been included into the optimization procedure and also different amounts of available gradient information have been explored. At the end of the chapter, a small test on a oil reservoir simulator has been performed. The last chapter is the conclusion which also suggests further work. There are two appendices. Appendix A contains graphs and tables from the testing of the algorithm and Appendix B contains the source code.

Chapter 2

Literature Review

During the last years there have been a lot of advances in derivative-free optimization. In 2003, some of the leading researchers in the field published a book [1] on the subject. This book is the first of its kind, which explains the theory of different kinds of derivative-free optimization. The available theory on derivative-free optimization with constraints were very limited at that time (and still is), hence, the book focuses on the unconstrained optimization problem.

The main difference between gradient-based and derivative-free optimization is that derivative-free methods does not use any derivative information of the true function, nor of the constraints. The lack of such information implies that the same performance as of gradient-based methods cannot be expected. Derivative-free methods can solve problems that consists of a few hundred variables. In gradient-based methods, the first-order necessary conditions says that the gradients are zero at the solution. This convenient information to help setting up a termination criterion cannot be used in derivative-free optimization as these gradients are not available. In the case of an expensive and/or a noisy true function, designing a stopping criterion is challenging. In the case of model-based derivative-free optimization where the the surrogate model is a (fully or under-determined) second order polynomial, early termination might be the preferred choice. The reason is that the progress of the solver, as we are getting closer to a minimum, will slow down because both the trust-region radius and the step size are converging to zero. This implies that the model must be updated frequently with many new points, unless the true function looks like a quadratic function around the minimum. When the function evaluations are time-consuming, then the extra time spent trying to find a better solution might not be worth it. The expected convergence is typically closer to linear than to quadratic. Hence, a stopping criterion that makes the solver quit early could be preferred. [1]

The book starts by describing ”Direct Search” methods. These methods are very simple procedures in principle: choose a starting point, and sample the function around that point. Move either in a direction as soon as you have found a decrease in the objective

function, or do a full search around the point and then choose the best point as your new best temporarily optimum. Then continue the process until you cannot find a better point. These methods does not use any derivative information in any way, but rather infer a direction of decrease based on direct sampling of the solution space[1]. Moreover, the fact that they are simple to implement, and that many of them are (somewhat) robust against noisy cost functions, makes them attractive[1]. However, if evaluating the function is expensive, these methods will not be advantageous, because they usually require a larger number of function evaluations for convergence compared to a model-based trust-region version. In [1] a comparison between four derivative-free methods on two examples are given, where the score is based upon the amount of function evaluations needed for convergence. They conclude that model-based trust-region methods are more efficient.

The book[1] also covers derivative-free model-based trust-region methods. If one compares the model-based with the direct-search methods, it can be seen that these two approaches are completely different. The theory of the model-based methods is quite complex, whereas the theory of the direct-search methods is a lot easier. The different parts needed to create a globally convergent derivative-free model-based trust-region method is presented in the book. In fact, it gives a framework which can be used to design globally convergent model-based trust-region methods. The main ingredients are: (i) how to build and update a surrogate model, (ii) how to maintain a well suited set of interpolation points and (iii) how to find the minimum of the model within the trust-region.

In [1] both the cases where you have a over-determined model (i.e., more interpolation points than needed, use regression) and the case where you have a fully determined model are covered. However, the case when you have an under-determined model is mentioned only shortly. Because this project is concerned about the use-case where the black-box is an oil reservoir simulator, which is a very time consuming computation, we are interested in exploring the under-determined models within our derivative-free optimization procedure. The question is then whether an under-determined second order polynomial model will provide enough curvature information and replicate the true function well enough for it to be useful. We therefore turn our attention to the work of M. J. D. Powell, who has done a lot of work on both under and fully determined quadratic approximation models for derivative-free optimization procedures. In [7], Powell compares different types of approximations:

- Linear. I.e., the Hessian matrix is the zero matrix.
- Fully quadratic. I.e., all elements of the Hessian are fully determined by the interpolation conditions.
- Diagonal quadratic. The Hessian is imposed a diagonal structure. All off-diagonal terms are set to zero.
- Sparse quadratic. The Hessian is imposed a predetermined structure.

The fully quadratic and the linear models are models that are uniquely determined by the interpolation points alone without imposing other constraints.

The numerical experiments presented in [7] suggest, not surprisingly, that including curvature information (i.e., a Hessian) in the model makes it easier to find the minimum of the function. Depending on the test functions, it varies which of the quadratic models (full, sparse, diagonal) that does the best job. Here ”good” means that the total amount of function evaluations, during the minimization of the function, is low. However, the sparse quadratic models seem to perform best. That the sparse quadratic models perform better than the diagonal models is expected because some known information of the sparsity of the Hessian of the true function is applied. This gives an “unfair” advantage to the sparse models. For the current project, the most important part of this article is the new way of updating the models, which requires fewer operations. This is done using an under-determined quadratic model (could be fully determined). Instead of forcing any kind of sparsity pattern on the quadratic model to take up the remaining freedom (i.e., those that are not specified by the interpolation conditions), the freedom is eliminated by minimizing the Frobenius norm of the change of the second order derivative matrix from one iteration to the next. Powell also presents some promising numerical results. The fact that we don’t need to impose any known structure on the Hessian is a very important feature of this method because this kind of structure is not necessarily known. Thus, it makes the modelling procedure more general and easier to use.

This work was continued by Powell in a paper titled “Least Frobenius norm updating of quadratic models that satisfy interpolation conditions”[8]. In this paper, the method of updating a possibly under-determined quadratic model is presented in great detail. The remaining freedom is, as mentioned, taken up by minimizing the Frobenius norm of the change of the second derivative matrix of the model. The model is uniquely defined due to the Frobenius norm being strictly convex. An efficient way of updating the system of equations when one of the interpolation points is replaced with a new one is also given. Not only is this method efficient, but it also keeps the Lagrange polynomials available, which is sought after as they are often used to maintain a good geometry of the set of interpolation points.

There is one critical demand that must be fulfilled when replacing an interpolation point, which is related to which point to remove when adding a new one. Namely, that the Lagrange polynomial corresponding to the old interpolation point must be nonzero evaluated at the new point. This demand is easily fulfilled and does not give rise to any problems, and is discussed later in this report. To keep the amount of operations low at each updating, a decomposition of the Hessian is stored instead of an explicit version. This suggests that using a (truncated) conjugate gradient method to solve the subproblem is preferable. Further explanation of this is also given later in the report.

Until now, there has been no mentioning of the center point of the model. Due to floating point arithmetic it is advantageous to change the center point of the model when the distance between the current best point and the current center point becomes “too large”. Fortunately, Powell provides convenient formulas for this in the same paper [8]. The cost of doing this is high and should therefore only be done when strictly necessary. Changing the center point to the current best point each time a new optimum is found is therefore not

recommended. A method to express the Lagrangian polynomials without their constant terms is presented in [8]. Each Lagrange polynomial is a quadratic polynomial, and the constant term is removed by using a difference of the Lagrange polynomials instead of the polynomials themselves. The reason to do this is purely related to the natural limitation of floating point arithmetic. All the needed changes to the mentioned formulas are provided. The updating formulas are supposed to automatically correct for accumulated errors. Numerical experiments show that this is the case, and particularly the one without the constant term is better. However, both versions do a good job. The tests also show that shifting the center point is crucial to avoid an unacceptable amount of loss of accuracy.

Even though [8] provides promising results, the methodology is not robust enough against numerical errors. A solution which reduces such errors is given by Powell in [9]. Without going into too many details, as this will be presented in the theory part of this project, the main result is as follows. If a factorization of a submatrix is stored instead of the matrix itself, one can force the rank of the submatrix to be correct. A method to update this factorization is given. The rest of the matrix is updated as in [8]. To understand which matrices, please see the theory chapter.

Powell has released two algorithms: NEWOUA[10] and BOBYQA [11], which implement the methodologies for model building and updating described so far. NEWUOA is a software (implemented in Fortran 77) that tries to find the smallest value of a function using a derivative-free model-based trust-region optimization technique. BOBYQA can be viewed upon as an extension of NEWUOA, where bounds on the decision variables can be set (i.e., $\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}$, where $\mathbf{a}$ and $\mathbf{b}$ are constant vectors), which is of great practical use for further work on this project. Both papers provide a good practical approach of how to implement the methods just mentioned. Initialization procedures are given, including how to select the first set of interpolation points and how to set up the first model. Because this project will use the framework given in [1], a lot of the content of these two papers are not directly relevant.

Up to now, we have presented references discussing initializing, maintaining and updating the model and the interpolation set. The next important part of a derivative-free model-based trust-region algorithm is to find a way of finding the minimum of the model within the trust-region. The following review is concerned about the case when there is no constraints. Some algorithms to do this are provided in [1]. To have a global convergent algorithm, it is crucial that the found minimum is as least as good as something related to the steepest descent. The Cauchy Step is that "something", and is the step towards the minimum of the model along the steepest descent direction within the trust-region. Actually, as long as the step is a fraction of the Cauchy step, global convergence can still be achieved. However, the mentioned global convergence is only to first-order critical points (i.e., the gradient of the model is zero). If global convergence to second-order critical-points is desired, then the step can be found along a direction related to the biggest negative curvature. The Eigenstep is defined as a scaled and possibly sign-switched eigenvector of the Hessian corresponding to the most negative eigenvalue. The Eigenstep serves the same purpose as the Cauchy step in this situation. If negative curvature is present, then both these steps are

calculated, and the one that provides the better solution (i.e., lowest value of the model) is chosen. [1]

However, in practise using the Cauchy step normally results in a slowly convergent method, while asymptotic rate of convergence is possible[12]. An algorithm to find the exact or nearly exact solution is given in [12]. This algorithm is, to quote the book, "[...] most definitely effective, it is not necessarily efficient". Meaning that it will find a good minimizer, but the computational effort might be very high, depending on the sparsity of the Hessian. The method can be used in the case of an indefinite (both negative and positive eigenvalues) Hessian matrix. Very simplified, the algorithm tries to find the minimum in an iterative process by solving a nonlinear equation using Newton's method. Considering the high computational effort, alternative methods that give an approximate solution could be attractive.

The Dogleg method by Powell and the Double-dogleg method by Dennis and Mei are two such algorithms. Both algorithms need the Hessian matrix to be positive semi-definite, which is not tolerable for this project. However, a modified version of the Dogleg method, which doesn't have that criterion, is given in [13], whereas numerical experiments of this method is performed in [14]. The first paper, [13], shows that using this algorithm in an unconstrained optimization leads to the same theoretical global and local convergence properties. The algorithm narrows down the search of the minimizer in a subspace spanned by selected vectors. This algorithm is appropriately named The Indefinite Dogleg Method.

All the mentioned approximate methods (including the nearly exact version from [12]), have all one common disadvantage, and that is that they all use Cholesky factorization of the Hessian (or of a sum including the Hessian). This factorization can be computationally expensive as the number of variables increases. In addition, they all need the Hessian in its explicit form. (Remember that the method of Powell in [8] only stores a factorization of the matrix.) Thus, using an optimization technique that multiplies the Hessian with a vector will be advantageous regarding number of operations needed.

A method which solves both of these issues is the conjugate gradient method[15]. The conjugate gradient method can be thought of as an alternative to the steepest descent. In the steepest descent, a step might undo some of the improvement made in a previously taken step. This is avoided in the conjugate method by always only taking steps that are Q-orthogonal (or Q-conjugate) to all the previous steps, and not just the previous one, as is the case for the steepest descent. However, this method, in its original form, cannot directly be used to find the minimum inside of a trust-region since the problem must be unconstrained. In addition, the Hessian of the model must be positive definite. Both of these two limitations are unacceptable in this project.

Fortunately, several improved versions of the conjugate gradient algorithm have been published. The truncated conjugate gradient method[16], which is "truncated" in the sense that the search area is now limited to a trust-region, is an interesting improvement. In addition, this improved version can handle indefinite Hessians. In the case of a positive def-

inite Hessian, the reduction is at least half of the global minimizer in the trust region[17]. However, no such conclusion can be drawn when the problem is nonconvex[12]. In the nonconvex case, the step produced by the algorithm might be just as bad as the Cauchy direction, and thus lead to a slow, but theoretically globally convergent algorithm, while in practice, the algorithm may barely converge[12].

An algorithm which improves on the performance of the truncated conjugate gradient is the "The generalized Lanczos trust-region method" [18]. The idea behind this algorithm is to look for the solution in one additional subspace, which is produced by the Lanczos algorithm, in the case that the solution of the truncated conjugate gradient method is at the boundary of the trust-region. This additional subspace almost always contains the solution to the subproblem. A recently published article [19], proposes new stopping criterion that will improve the numerical performance of this algorithm.

Algorithm 11.2 in "Introduction to Derivative-Free optimization"[1] will be the focus of this specialization project. Powell's method of building and updating an underdetermined quadratic model is embedded into the previously mentioned framework. There are several differences between this and Powell's algorithms (e.g., NEWUOA [10]), which will be presented later.

Stefan Wild has recently designed a methodology to compare derivative-free optimization algorithms that considers expensive function evaluations[20]. He expands upon something called *performance profiles*. Performance profiles[21] rate solvers based upon how long time they use to achieve a given reduction in the objective function value within a limit of function evaluation. *Data profiles* were designed to give the user information about the percentage of problems that can be solved (for a predetermined tolerance) with a limitation the allowed amount of function evaluations. This information is essential to users with expensive function evaluations. In this paper[20], a small comparison between two direct search solvers and NEWOUA (a model-based trust-region solver) is performed. Also in this test the model-based solver is the winner.

There is a lack of convergence theory of constrained derivative-free model-based trust-region optimization. However, the book [1] presents some ideas. Depending on the type of constraints, the algorithm may or may not still be globally convergent. If there are only bounds and linear constraints, the convergence theory of the book should be fairly easy to adapt[1]. There is one type of constraints that could be included while the global convergent property remains the same. The idea is to add some penalty corresponding to the amount that the constraints are being violated to the objective function value, whereas nothing else is changed. If it is possible to merge the objective function and the penalty terms into a merit function, then this method can be applied. If this method is used, one has to be careful and make sure that the magnitudes of the terms are about the same. E.g., having an objective function varying in the range of -0.001 to 0.001 whereas the penalty term is in the range of -10^9 to 10^9, the objective function becomes negligible. In addition, it is not guaranteed that the final solution satisfies the constraint(s). Some ideas when it comes to nonlinear constraints are suggested, and they are all presented later in the theory chapter.

The available theory on including gradients into the derivative-free algorithms is limited, and close to zero (or zero) when it comes to model-based trust-region algorithms. This makes sense, because if reliable gradients are available, gradient based methods should be applied. Further, if only some of the derivatives are known, then those that are missing can be estimated. In this project, we are trying to "go the other way". Instead of estimating the remaining gradients, we simply include those that we have into the derivative-free method.

The paper [22] presents a generalized pattern search algorithm which includes gradients, if available, to help select better search directions for the *poll step*. The poll step is the step when the algorithm evaluates point around the current iterate to find a direction which offers decrease in the function value. The paper reports a significant reduction in the maximum amount of function evaluations that are needed in the poll steps.

This review has presented the relevant literature sources needed to conduct this Master's thesis. First, a note on other types of derivative-free optimization was given. How to build, update and maintain an under-determined quadratic model have been discussed which are an important part of the overall algorithm because we need the model to at least replicate (some of) the main curvature of the true function. If the model is too bad, we might end up taking a lot of steps in the wrong direction and the algorithm might run forever. Further, methods for finding a minimum of the model within the trust-region were briefly presented. A short note on how to handle constraints was also given. Next, the theory of some of the presented topics will be given.

Chapter 3

Theory

This chapter begins with a simplified explanation and overview of the derivative-free model-based trust-region optimization algorithm that will be used in this project. A flowchart of the algorithm can be seen in Figure 3.1. The goal of this section is to make it easier for the reader to follow along and to understand why the subsequent theory is presented.

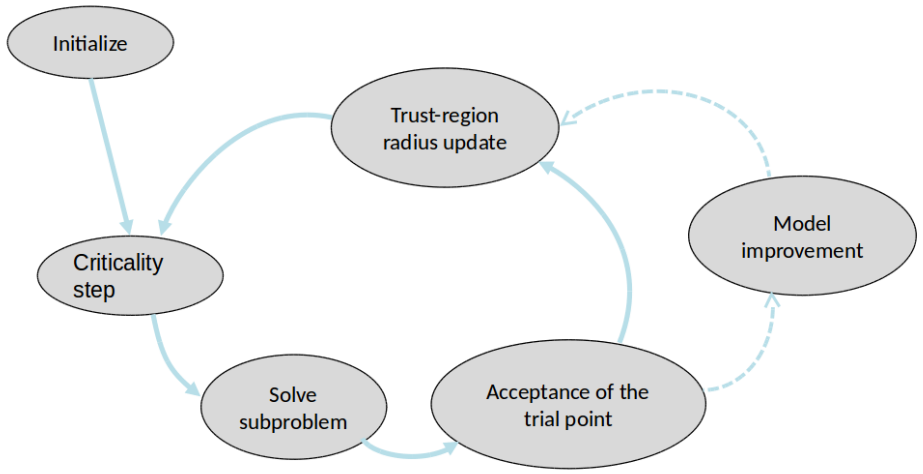


Figure 3.1: Overview of the different step in the derivative-free model-based trust-region optimization method. This diagram is based upon Algorithm 11.2 in Introduction to derivative-free optimization[1].

The algorithm starts with the *Initialize* step. The number of interpolation points used to create the surrogate model is chosen, as well as the initial guess and the initial trust-region radius. The starting point, which is provided by the user, is perturbed to get the selected

amount of sample points. These points are evaluated and used to create the first surrogate model.

In the next step, some measurement of optimality is required. We have chosen the gradient of the Lagrangian because the optimization problem will have constraints. In the unconstrained case, the gradient of the surrogate model suffices. This step makes sure that the surrogate model becomes more accurate when the gradient (of the Lagrangian) is close to zero. This will prevent the scenario where the gradient is too small compared to the trust-region radius. If the gradient is small compared to the radius, then often a small step within the trust-region is taken. The reason why it is not preferable to take these small steps is because the model is most likely a bad predictor of the true function when the steps become too small. Remember that the surrogate model is constructed based upon points spread around the inside of the trust-region, meaning that maybe only one or two points will contribute with information close to the current iterate. Thus, the chance that the model mimics the true function in detail around the current iterate, is close to zero.

The *Criticality step* increases the accuracy of the model by reducing the trust-region radius. This implies that the radius is a natural termination criterion for the algorithm. After the radius is reduced, new points will be selected until the gradient of the Lagrangian satisfies some requirement. When the radius is decreased, some of the old points might leave the trust-region, forcing these points to be replaced by points inside the trust-region. Note that the replacement of these points might make the geometry of the new set of points worse, and, hence, more replacements must be made. To elaborate, when one point outside the trust region is going to be replaced by a point inside, it is not necessarily possibly to place that new point such that the geometry of all the points in the set is good. Maybe some of the old ones must also be moved.

At this point of the algorithm, the surrogate model satisfies a criterion and hopefully it is a good predictor of the true function. A minimum of this surrogate model, which obeys the constraints of the overall problem, is found. This optimization problem is known as the *subproblem*. The found minimum is evaluated.

After the point has been evaluated, the next step is the *Acceptance of the trial point* step. Here it is decided if the point may be accepted into the set of interpolation points. One out of three cases might happen. (i) The point is the best found point so far and it is included in the set of interpolation points before going to the *Trust-region radius update* step. (ii) The point is disregarded all together. (iii) The point is included, but it is not the best point found thus far. For both (ii) and (iii) the *Model improvement* step will be entered because, clearly, the model is not good enough.

In the *Model improvement* step, one point of the interpolation set is attempted replaced solely with the goal of making the model better. This step may or may not be able to do a replacement. Afterwards the algorithm enters the *Trust-region radius update* step.

In the *Trust-region radius update* step the trust-region radius is updated based upon

how well the model predicted in the *Acceptance of the trial point* step. All the different steps have been entered, and the algorithm is back to the *Criticality step*.

3.1 Notations

Before we dive into the theory, the most used notation will be introduced. Let n be the number of inputs to the surrogate model, i.e., the number of decision variables the objective function will depend upon. The number of interpolation points is denoted by m, and this number is fixed throughout the entire optimization procedure. $\mathbf{y}_i$ is the i-th interpolation point, for $i = 1, 2, \ldots, m$. Further, let $\mathbf{y}_0$ be the center point of the model, and $\mathbf{y_b}$ be the best point found thus far. $f(\mathbf{x})$ represents the true function, while $\mathbf{x}$ is of the same dimension as $\mathbf{y}_i$. The $\mathbf{y}_i$'s vectors represent decision vectors that we have found values for, while for all other vectors, $\mathbf{x}$ is used. All the m interpolation points minus the center point, i.e., $(\mathbf{y}_i - \mathbf{y}_0)$, are stacked together in a big, n-by-m, matrix $\hat{\mathbf{Y}}$:

$$\hat{\mathbf{Y}} = \begin{bmatrix} (\mathbf{y}_1 - \mathbf{y}_0) & (\mathbf{y}_2 - \mathbf{y}_0) & \ldots & (\mathbf{y}_m - \mathbf{y}_0) \end{bmatrix}$$

Note that $\mathbf{y}_0$ does not necessarily have to be one of the $\mathbf{y}_i$'s vectors, but it may be.

3.2 The surrogate model

This section presents all the relevant theory for how to handle the surrogate model. First, the chosen model is presented and the derivation is shown. Then a method to update the model is given. Further, an extension of the updating method is presented which will make it more robust against floating point errors. How to include gradient information into the model building and updating procedures are derived. Afterwards, one method to shift the center point of the model and one method to increase the geometry of the set of interpolation points are given.

In the field of optimization using derivative-free model-based trust-region methods, it is common to use a polynomial of first or second order as the interpolation model. The chosen model is therefore a second order polynomial. Some reasons why these kind of models are commonly used are that they are easy to understand and they are easy to work with. By easy to work with, it is meant that they are easy to differentiate and they are easy to create based upon interpolation points. Moreover, if one could choose between finding the minimum of a low order polynomial and a complicated function, most people would probably choose the former.

A second order polynomial can be written on the following form:

$$Q(\mathbf{x}) = c + \mathbf{g}^\intercal(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^\intercal \mathbf{G}(\mathbf{x} - \mathbf{y}_0) \tag{3.1}$$

where c is a scalar, $\mathbf{g} \in \mathcal{R}^n$ is the gradient at $\mathbf{y}_0$ and $\mathbf{G} = \mathbf{G}^\intercal \in \mathcal{R}^{n \times n}$ is the Hessian of the model. Because $\mathbf{G}$ is symmetric this function has $(n + 1)(n + 2)/2$ coefficients.

If these coefficients should be determined solely based upon the interpolation conditions, then $(n+1)(n+2)/2$ sample points would have been needed. To clarify, by *interpolation conditions* it is meant

$$Q(\mathbf{y}_i) = f(\mathbf{y}_i), \quad i = 1, 2, \ldots, m$$

and these will remove m degrees of freedom as long as the points are linearly independent. Q is the surrogate model and f is the true function.

The true function (i.e., the oil reservoir simulator) is computationally expensive to evaluate, thus, using fewer points would be desirable. One possibility is to simply use a model with less coefficients such as a linear model. However, a linear model does not possess any curvature information whatsoever because the Hessian matrix is the zero matrix. There are algorithms out there which follow this approach such as COBYLA[23]. The selected approach is to include some curvature information which, hopefully, will make the algorithm converge faster.

The surrogate model will be on the form (3.1). As mentioned in the literature review, there exists different approaches to decide the remaining degrees of freedom. The selected approach is the one that is suggested in [7] which solves a minimization problem. The method is based upon the idea that the old model is a good approximation to the new model. This is (often) a reasonable thought as they are both interpolation models which shares $m-1$ points. After the interpolation conditions are satisfied, some change between the old and the new model is minimized. The updating is required to be independent of the center point of the model. The reason is not specified in the paper, but a unique model is required and considering that the center point could be any point, including it into the updating would make it dependent on the that point, and thus not unique. This leads the attention to the Hessian matrix which is the only part of (3.1) which is invariant to shift of the center point.

There are different representations to choose from when dealing with a second order polynomial such as the monomial basis, the radial basis and the Lagrange polynomial basis. The latter is chosen[1]. In fact, when it comes to the model itself, it doesn't matter which representation is chosen. However, the Lagrange polynomial basis is important for other reasons, which will be revealed later, and therefore it must be available. This basis will be available almost for free when the approach given in [7] is used. A second order polynomial (where $m = (n+1)(n+2)/2$) may be written in terms of the Lagrange polynomial basis:

$$Q(\mathbf{x}) = \sum_{j=1}^{m} f(\mathbf{y}_j)\ell_j(\mathbf{x}) \tag{3.2}$$

where the ℓ_j's are the Lagrange polynomials. These must satisfy the Kronecker delta property: $\ell_j(\mathbf{y}_i) = 1$ if $j = i$ and is zero otherwise. $f(\mathbf{y}_j)$ is the true function evaluated at $\mathbf{y}_j$.

Now that some general information around the model we are using have been established, the next topic can be presented: how to create and update the surrogate model.

[1] In fact, a modified version of the Lagrange polynomial basis will be used.

3.3 Updating of the interpolation model

In this part it is derived how the coefficients of the interpolation model in (3.1) can be uniquely determined when one interpolation points is replaced with a new one. The goal of this section is to find the quadratic polynomial, $Q^{\#}$, that must be added to the old model, Q, to get the new model, Q^{+}. I.e., find the model $Q^{\#}$ such that $Q + Q^{\#} = Q^{+}$. The old and the new model have all the same interpolation conditions but one. That means that the old model will have the exact same value at all of the interpolation points but the one that has been replaced. The superscripts $\#$ and $+$ will denote that a component belongs to the difference model, $Q^{\#}$, or the new model, Q^{+}, respectively. E.g., $\mathbf{y}_i^{+}$ will be part of the new set of interpolation points, whereas $\mathbf{y}_i$ will belong to the old set.

This is the representation of the new model:

$$Q^{+}(\mathbf{x}) = c^{+} + \mathbf{g}^{+\intercal}(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^{\intercal}\mathbf{G}^{+}(\mathbf{x} - \mathbf{y}_0) \tag{3.3}$$

and this is how the difference model is represented:

$$Q^{\#}(\mathbf{x}) = c^{\#} + \mathbf{g}^{\#\intercal}(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^{\intercal}\mathbf{G}^{\#}(\mathbf{x} - \mathbf{y}_0) \tag{3.4}$$

The corresponding interpolation conditions will be:

$$Q^{+}(\mathbf{y}_i^{+}) = f(\mathbf{y}_i^{+}), \quad i = 1, 2, \ldots, m \tag{3.5}$$

and

$$Q^{\#}(\mathbf{y}_i^{+}) = Q^{+}(\mathbf{y}_i^{+}) - Q(\mathbf{y}_i^{+}) = f(\mathbf{y}_i^{+}) - Q(\mathbf{y}_i^{+}), \quad i = 1, 2, \ldots, m \tag{3.6}$$

Note that all the right hand sides of (3.6) will be zero except for the one corresponding to the point that is being replaced.

Before continuing, the Frobenius norm is introduced in case the reader is not familiar with it. The Frobenius norm of a matrix is the Euclidean norm of the vectorized matrix. Here is a small example for illustration:

$$\left|\left|\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right|\right|_F = \left|\left|\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}\right|\right|_2 = \sqrt{a^2 + b^2 + c^2 + d^2}$$

3.3.1 Derivation of the solution of a convex quadratic program

As previously mentioned, after satisfying the m interpolation conditions (3.5) there is still $(n+1)(n+2)/2 - m$ degrees of freedom left. These will be determined by minimizing the change of the Hessian matrix from the old model to the new one. The Frobenius norm will be used as the measurement of change. This can be formulated as a minimization

problem. Specifically, the squared Frobenius norm of the change of the Hessian is minimized subject to the m interpolation conditions.

$\mathbf{G}^{\#}$, and consequently $\mathbf{G}^{+}$, is selected as the matrix that minimizes the squared Frobenius norm:

$$||\mathbf{G}^{\#}||_F^2 = ||\mathbf{G}^{+} - \mathbf{G}||_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{G}_{ij}^{+} - \mathbf{G}_{ij})^2$$

subject to it being symmetric and subject to the interpolating conditions.

Since the Frobenius norm is strictly convex (just as the Euclidean norm is), $Q^{\#}$ is uniquely defined if the following two properties of the interpolation points are fulfilled[8]:

(P1) If $\mathcal{Q}$ is the space of all quadratic polynomials from $\mathcal{R}^n$ to $\mathcal{R}$ that are zero for the interpolation points, then the dimension of this space must be $\frac{1}{2}(n+1)(n+2)-m$.

(P2) If $p(\mathbf{x})$ is any linear polynomial from $\mathcal{R}^n$ to $\mathcal{R}$ and $p(\mathbf{x}) = 0$ for all the interpolation points, then p must be identically zero.

It is demonstrated why these two properties make $Q^{\#}$ uniquely defined by the following contradiction[8]: (P1) ensures that the model is equal to the true function for all the interpolation points. Let $q(\mathbf{x}) \in \mathcal{Q}$ be the polynomial with the second derivative matrix that minimizes $||\mathbf{G}^{+} - \mathbf{G}||_F^2$. If $\hat{q}(\mathbf{x}) \in \mathcal{Q}$ is another polynomial which has the same double derivative matrix, then $q - \hat{q}$ will be a linear polynomial which is nonzero. However, property (P2) says that such a linear polynomial must be zero, and thus we have a contradiction, and we can conclude that $Q^{\#}$ is uniquely defined. Later, a convenient way to ensure these two properties is presented.

The parameters of $Q^{\#}$ can be found by solving a convex quadratic program:

$$\begin{aligned}
\min_{\mathbf{G}_{ij}^{\#},\mathbf{g}^{\#},c^{\#}} \quad & \frac{1}{2}||\mathbf{G}^{\#}||_F^2 = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}{\mathbf{G}_{ij}^{\#}}^2 \\
\text{s.t.} \quad & \\
& c^{\#} + \mathbf{g}^{\#\mathsf{T}}\mathbf{y}_{d,i} + \frac{1}{2}\mathbf{y}_{d,i}^{\mathsf{T}}\mathbf{G}^{\#}\mathbf{y}_{d,i} = f(\mathbf{y}_i^{+}) - Q(\mathbf{y}_i^{+}) \qquad i = 1, 2, \ldots, m
\end{aligned} \tag{3.7}$$

where $\mathbf{y}_{d,i} = (\mathbf{y}_i^{+} - \mathbf{y}_0)$. Note that $Q(\mathbf{y}_i^{+})$ is the old model evaluated at the new interpolation point. Recall that $\mathbf{G}^{\#}$ is required to be symmetric. However, this requirement is automatically satisfied by (3.8c).

The first order KKT conditions of this minimization problem will now be derived. The

corresponding Lagrangian function is:

$$\mathcal{L}(c^{\#}, \mathbf{g}^{\#}, \mathbf{G}^{\#}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}{\mathbf{G}_{ij}^{\#}}^2 - \sum_{k=1}^{m}\lambda_k\left(c^{\#} + \mathbf{g}^{\#}\mathbf{y}_{d,k} + \frac{1}{2}\mathbf{y}_{d,k}^{\intercal}\mathbf{G}^{\#}\mathbf{y}_{d,k} - (f(\mathbf{y}_k^+) - Q(\mathbf{y}_k^+))\right),$$

where λ_k's are the Lagrange multipliers. We then derive the partial derivatives of $\mathcal{L}$ with respect to variables $c^{\#}$, $\mathbf{g}^{\#}$ and $\mathbf{G}^{\#}$ and set these equal to zero at the solution:

$$\frac{\partial\mathcal{L}}{\partial c^{\#}} = \sum_{k=1}^{m}\lambda_k = 0 \tag{3.8a}$$

$$\frac{\partial\mathcal{L}}{\partial \mathbf{g}^{\#}} = \sum_{k=1}^{m}\lambda_k(\mathbf{y}_k^+ - \mathbf{y}_0) = 0 \tag{3.8b}$$

$$\frac{\partial\mathcal{L}}{\partial \mathbf{G}^{\#}} = \sum_{k=1}^{m}\lambda_k(\mathbf{y}_k^+ - \mathbf{y}_0)(\mathbf{y}_k^+ - \mathbf{y}_0)^{\intercal} - \mathbf{G}^{\#} = 0 \tag{3.8c}$$

The unknown parameters are now uniquely given by (3.8) and the m constraints in (3.7)[8]. Clearly, (3.8c) can be used to eliminate $\mathbf{G}^{\#}$. These equations will be put together as a linear system of equations in matrix form. Let $\mathbf{A}^+$ be the $m \times m$ matrix whose elements are:

$$\mathbf{A}_{i,k}^+ = \frac{1}{2}\left[(\mathbf{y}_i^+ - \mathbf{y}_0)^{\intercal}(\mathbf{y}_k^+ - \mathbf{y}_0)\right]^2, \quad \forall\, i,k \in \{1,2,\ldots,m\} \tag{3.9}$$

Further, let $\mathbf{e}$ be an $m \times 1$ column vector with all elements set to 1, and $\mathbf{F}^{\#}$ be of the same size with element i having the value $f(\mathbf{y}_i^+) - Q(\mathbf{y}_i^+)$. All the λ_k's are stacked together into the column vector λ of appropriate size. The matrix form of the linear system of equations is then:

$$\left(\begin{array}{c|ccc} \mathbf{A}^+ & \mathbf{e} & \vdots & (\hat{\mathbf{Y}}^+)^{\intercal} \\ \hline \mathbf{e}^{\intercal} & & \mathbf{0} & \\ \cdots & & & \\ \hat{\mathbf{Y}}^+ & & & \end{array}\right)\left(\begin{array}{c} \lambda \\ \hline c^{\#} \\ \cdots \\ \mathbf{g}^{\#} \end{array}\right) = \mathbf{W}^+\left(\begin{array}{c} \lambda \\ \hline c^{\#} \\ \cdots \\ \mathbf{g}^{\#} \end{array}\right) = \left(\begin{array}{c} \mathbf{F}^{\#} \\ \hline 0 \\ \cdots \\ \mathbf{0} \end{array}\right), \tag{3.10}$$

where $\mathbf{W}^+$ is a $(m+n+1) \times (m+n+1)$ matrix.

Before we continue with how to do the updating of the model, we will see how we can create the Lagrange polynomials. The reason why this is presented now, is that it is very related to the just found solution of the minimization problem.

3.3.2 Create the Lagrange polynomials

The normal Lagrange polynomials, i.e., those belonging to a second order polynomial which is fully determined by the interpolation conditions alone, are uniquely determined

by the Kronecker-delta property. The Kronecker-delta property is $\ell_j(\mathbf{y}_i) = \delta_{ji} = 1$ if $j = i$ and is zero otherwise. However, now that there are less interpolation points, there are also have less Lagrange polynomials and thus also less Kronecker delta properties to fulfill. This means that the coefficients of the Lagrange polynomials are not uniquely defined. In other words, there will be several bases that will give the same (and correct) output and all of them can satisfy the Kronecker delta property. To overcome this lack of uniqueness, the same approach is taken as the one used for determining $Q^{\#}$. But now the squared Frobenius norm of $\nabla^2 \ell_j^+$ is minimized instead of $\mathbf{G}^{\#}$ and now it is subject to the Kronecker delta property instead of the interpolation conditions. $\nabla^2 \ell_j^+$ is the Hessian belonging to the j-th Lagrange polynomial of the new model. The ℓ_j's are now applicable to our under-determined model. The Kronecker delta property conditions will be as follows:

$$\ell_1^+(\mathbf{y}_j^+) = c_1^+ + (\mathbf{g}_1^+)^\intercal(\mathbf{y}_j^+ - \mathbf{y}_0) + \frac{1}{2}(\mathbf{y}_j^+ - \mathbf{y}_0^+)^\intercal \mathbf{G}_1^+(\mathbf{y}_j^+ - \mathbf{y}_0) = \delta_{1j} \tag{3.11a}$$

$$\ell_2^+(\mathbf{y}_j^+) = c_2^+ + (\mathbf{g}_2^+)^\intercal(\mathbf{y}_j^+ - \mathbf{y}_0) + \frac{1}{2}(\mathbf{y}_j^+ - \mathbf{y}_0)^\intercal \mathbf{G}_2^+(\mathbf{y}_j^+ - \mathbf{y}_0) = \delta_{2j} \tag{3.11b}$$

$$\vdots$$

$$\ell_m^+(\mathbf{y}_j^+) = c_m^+ + (\mathbf{g}_m^+)^\intercal(\mathbf{y}_j^+ - \mathbf{y}_0) + \frac{1}{2}(\mathbf{y}_j^+ - \mathbf{y}_0)^\intercal \mathbf{G}_m^+(\mathbf{y}_j^+ - \mathbf{y}_0) = \delta_{mj} \tag{3.11c}$$

for $j = 1, 2, \ldots, m$.

Let's say that we want to find $\ell_t^+(\mathbf{x})$, which means that all the right hand sides in (3.11) should be zero except for the one belonging to $\ell_t^+(\mathbf{x})$, which will be 1: $\ell_t^+(\mathbf{y}_t) = \delta_{tt} = 1$. If this information is transferred into the matrix formulation of the solution of (3.7), it will be equivalent to setting the right hand side of (3.10) to a unit vector with the element 1 at position t:

$$\mathbf{W}^+ \begin{pmatrix} \lambda_t^+ \\ \hline c_t^+ \\ \cdots \\ \mathbf{g}_t^+ \end{pmatrix} = \begin{pmatrix} \mathbf{e}_t \\ \hline 0 \\ \cdots \\ \mathbf{0} \end{pmatrix} \tag{3.12}$$

If we calculate the Lagrange polynomials directly from (3.12) we would need to solve an almost identical linear system of equations m times, one time for each Lagrange polynomial. A more computational budget friendly method will be presented later. Now we will turn the focus back to the updating procedure.

3.3.3 Simple updating scheme

Before the method that is explained in the paper [8] is presented, a more intuitive method is suggested. This alternative is included because it is more straightforward and it accomplishes the same theoretical result. In addition, hopefully, it will be easier for the reader to

understand what is happening. However, computationally wise, it is terrible in comparison.

Let's assume we already have a model Q and that we want to update it after one of the interpolation points has been replaced, t say. Let $\mathbf{x}^+$ be the new interpolation point, i.e., $\mathbf{x}^+ = \mathbf{y}_t^+$. The $Q^\#$ model will be calculated by using a linear solver on equation (3.10). As mentioned, the $\mathbf{F}^\#$ vector consists of the values $(f(\mathbf{y}_i^+) - Q(\mathbf{y}_i^+))$ for $j = 1, 2, \ldots, m$. This means that $\mathbf{F}^\#$ will be the unit vector $\mathbf{e}_t$ times $f(\mathbf{x}^+) - Q(\mathbf{x}^+)$. Now that we have $Q^\#$ we can simply add it to the old model Q and we are done.

The observant reader might have noticed that $Q^\#$ is actually a scaled version of the t-th Lagrange polynomial of the new model. To see this, compare the right hand side of (3.12) with the $\mathbf{F}^\#$ used here. This means that the model can be updated by this formula:

$$Q^+(\mathbf{x}) = Q(\mathbf{x}) + Q^\#(\mathbf{x}) = Q(\mathbf{x}) + \ell_t^+(\mathbf{x})(f(\mathbf{x}^+) - Q(\mathbf{x}^+))$$

In the scenario that we don't yet have a model, i.e., none of the coefficients of Q is determined. Then this method can be used to create the first model. Simply put all of the coefficients to zero and follow the exact same procedure. The only difference will be that the $\mathbf{F}^\#$ vector will possibly have values in all its slots, and the $Q^\#$ will not be a scaled Lagrange polynomial.

This initialization procedure actually reveals two new ways of using this under-determined model. Until now, it has been assumed that the old model is a good approximation to the new one. But if we want to remove this assumption, we can use this method to reset the stored Hessian. The other idea is to never use the old model, and let the model be determined solely on (3.10) at every iteration. This would lead to a model which has at least a linear part and is extended with some curvature information such that the interpolation conditions are satisfied.

The focus will now be turned to the updating procedure that is given in [8].

3.3.4 Sophisticated updating scheme

The way that the paper [8] deals with the updating of the system with a new point is very different and a lot more complicated than the just presented method. As we have seen, we are solving the (3.10) a lot of times. The idea in this paper is that we will work with the inverse of the $\mathbf{W}$ matrix. This inverse is hereby denoted by $\mathbf{H}$, i.e., $\mathbf{H} = \mathbf{W}^{-1}$. The inverse is never calculated directly, but initialized and then updated directly. First, a procedure to update the entire $\mathbf{H}$ matrix is presented. This procedure is somewhat resistant to errors that occur due to the limited accuracy of floating point arithmetic. However, users of the method reported they had problems and Powell came up with an even more robust updating procedure. This is an extension which is used to update one submatrix of $\mathbf{H}$. The rest of the matrix is still updated with the original formula, which will be presented shortly. The extension will be given afterwards.

Here, the formulas detailing the model updating procedure are presented and commented upon, but the complete derivation is not presented. Full derivations are of less interest to this project, however, the most important parts of the methodology are explained and commented. Please see [8] for full derivations.

Updating the H matrix

One of the most important goals of the model updating procedure is to keep the computational effort of each update low ($\mathcal{O}(m^2 + n^2)$). Moreover, an important feature of the procedure is that it is designed to automatically reduce the effect of numerical errors. The key idea behind the updating procedure, which allows for both of the preceding specifications, is the fact that $\mathbf{W}^+$ and $\mathbf{W}$ only differ in the t-th row and column, which can easily be seen from (3.10). Remember that both $\hat{\mathbf{Y}}$ and $\mathbf{A}$ are determined by the interpolation points alone. Further, $\mathbf{H}^+$ can be calculated from $\mathbf{H}$ and the t-th column of $\mathbf{W}^+$, $\mathbf{w}_t^+$. The fact that $\mathbf{H}^{-1}$ should be equal to $\mathbf{W}$ is also a key factor. The matrix $\mathbf{W}$ is not stored.

Before the results are given, some information regarding how the updating procedure is robust[2] against numerical errors due to floating point arithmetic will be given.

In the beginning of an updating procedure, $\mathbf{H}$ is available and $\mathbf{w}_t^+$ can easily be calculated ($\mathbf{w}_t^+$ is as defined above) from the interpolation points. The following update procedure is by no means effective nor is it used, but the given relationship provides good protection against accumulation of numerical errors due to computer rounding errors.

Let's start by inverting $\mathbf{H}$ to get $\mathbf{W}$. We know that $\mathbf{W}$ is symmetric (see equation (3.10)) and we already have the vector $\mathbf{w}_t^+$. This means that we can calculate the new $\mathbf{W}^+$. Then this new matrix can be inverted to get $\mathbf{H}^+$, and the updating of the inverse matrix is done.

We now assume that any numerical errors from the current iteration can be neglected, and that there might be huge errors in $\mathbf{H}$ due to previous iterations. Most of the errors in the matrix will be inherited by the updated matrix. Let the error be denoted as $\Delta = \mathbf{W}_r - \mathbf{H}^{-1}$, where $\mathbf{W}_r$ is the "real" $\mathbf{W}$ matrix as defined by (3.10). Δ is a matrix of the same size as $\mathbf{W}$. Remember that $\mathbf{W}_r$ is not actually stored, nor created. This error, Δ, will not grow as the number of iterations increases[8]. This statement will now be justified.

Let $\Delta^+ = \mathbf{W}_r^+ - (\mathbf{H}^+)^{-1}$ be the error of the updated system. Remember that Δ represents the accumulation of numerical errors due to previous iterations. Now we will use the relationship given in the previous paragraph, namely, that the t-th row and column of $(\mathbf{H}^+)^{-1}$ is equal to the same row and column of $\mathbf{W}_r^+$. This implies that the t-th row and column of Δ^+ must be zero.

[2] It is not "fully robust", but it helps reducing the effect of numerical errors that occur due to computer rounding errors.

Now, let's use the fact that $\mathbf{W}_r$ and $\mathbf{W}_r^+$ will be identical for all rows and columns except the t-th. We know that $\mathbf{H}^{-1}$ and $(\mathbf{H}^+)^{-1}$ will be identical for all rows and columns except the t-th ones. All this implies that any accumulated error in the t-th row of Δ will be eliminated in Δ^+, whereas the other errors will be inherited by Δ^+. This, in turn, implies that removing an interpolation point that has been in the set for a long time is preferable.

Remember that t is taken from the closed interval $[1, 2, 3, \ldots, m]$ and that $\mathbf{H}$ is a $(m+n+1) \times (m+n+1)$ matrix, thus, any errors in the bottom right $(n+1) \times (n+1)$ matrix will not be removed. This is where the extension of updating method comes in. It will be presented later.

We did assume that we could neglect the contribution of rounding errors due to the current iteration. However, if $|\Delta^+_{i,j}| > |\Delta_{i,j}|$ (where the subscript i, j denotes the i, j element of the matrix), then this difference is due to rounding errors due to the current iteration. The discussion is now done and we proceed by presenting the updating formulas.

Let $\mathbf{x}^+$ be the new interpolation point, and let t be the index of the interpolation point that is going to be replaced. Further, Let $\mathbf{w} \in \mathcal{R}^{m+n+1}$ (not the same vector as $\mathbf{w}_t^+$) be the following vector:

$$\begin{aligned} \mathbf{w}_i &= \frac{1}{2}\{(\mathbf{y}_i - \mathbf{y}_0)^\intercal(\mathbf{x}^+ - \mathbf{y}_0)\}^2, \quad i = 1, 2, \ldots, m \\ \mathbf{w}_{m+1} &= 1 \\ \mathbf{w}_{m+i+1} &= (\mathbf{x}^+ - \mathbf{y}_0), \quad i = 1, 2, \ldots, n \end{aligned}$$

then

$$\begin{aligned} \mathbf{H}^+ = \mathbf{H} + \frac{1}{\sigma_t}[&\alpha_t(\mathbf{e}_t - \mathbf{H}\mathbf{w})(\mathbf{e}_t - \mathbf{H}\mathbf{w})^\intercal - \beta_t \mathbf{H}\mathbf{e}_t\mathbf{e}_t^\intercal\mathbf{H} \\ &+ \tau_t\{\mathbf{H}\mathbf{e}_t(\mathbf{e}_t - \mathbf{H}\mathbf{w})^\intercal + (\mathbf{e}_t - \mathbf{H}\mathbf{w})\mathbf{e}_t^\intercal\mathbf{H}\}] \end{aligned} \tag{3.13}$$

where

$$\begin{aligned} \alpha_t &= \mathbf{e}_t^\intercal\mathbf{H}\mathbf{e}_t = \mathbf{H}_{t,t} \\ \beta_t &= \frac{1}{2}||\mathbf{x}^+ - \mathbf{y}_0||^4 - \mathbf{w}^\intercal\mathbf{H}\mathbf{w} \\ \tau_t &= \ell_t(\mathbf{x}^+) = \mathbf{e}_t^\intercal\mathbf{H}\mathbf{w} \\ \sigma_t &= \alpha_t\beta_t + \tau_t^2 \end{aligned}$$

where $\mathbf{e}_t$ is the t-th unit vector in $\mathcal{R}^{m+n+1}$. The following convenient relationship is also available[3]:

$$\ell_j(\mathbf{x}^+) = \mathbf{e}_j^\intercal\mathbf{H}\mathbf{w}, \quad j = 1, 2, \ldots, m$$

It is proven in exact arithmetic that σ_t will be greater than zero as both $\alpha_t \geq 0$ and $\beta_t \geq 0$. As stated earlier, a convenient way to ensure that the properties (P1) and (P2) are satisfied

[3] In "Extract the Lagrange polynomials" we will see how the Lagrange polynomials can be extracted from $\mathbf{H}$.

would be given. In [8] it is stated that they will be as long as $\ell_t(\mathbf{x}^+)$ is nonzero. The interested reader can find the reasoning of this in the original paper [8]. We can see that this is necessary or else the value $1/\sigma_t$ could potentially explode. In addition, it means that the $\mathbf{W}^+$ of (3.10) for the new interpolation points is nonsingular. If a square matrix is nonsingular, then the matrix is invertible.

The extension of this updating formula will be presented next.

Updating one part of the H

As previously mentioned, a separate updating formula has been invented by Powell to update a submatrix of $\mathbf{H}$ that will reduce the effect of numerical errors even more. See [10] and [9] for practical and theoretical descriptions, respectively. Before the formulas are given, some reasoning of how this factorization helps to reduce the effect of computer rounding errors is given. Let

$$\mathbf{H} = \left(\begin{array}{c|c} \Omega & \Xi^\intercal \\ \hline \Xi & \Upsilon \end{array}\right),$$

where Ω, Ξ and Υ are of the dimension $m \times m$, $(n+1) \times m$ and $(n+1) \times (n+1)$, respectively. Ω is the submatrix that will be updated by this formula. Further, let $\mathbf{W}$ in (3.10) be expressed as:

$$\mathbf{W} = \left(\begin{array}{c|c} \mathbf{A} & \mathbf{X}^\intercal \\ \hline \mathbf{X} & \mathbf{0} \end{array}\right),$$

where

$$\mathbf{X}^\intercal = \left(\mathbf{e} \vdots \hat{\mathbf{Y}}^\intercal\right).$$

and $\mathbf{A}$ is defined as before. The interpolation points are chosen such that $\mathbf{X}$ will have full rank, i.e., the rank is $n+1$. The proceeding updating formula for the factorization is based upon the remark that $\mathbf{X}\Omega = 0$. That this is the case, can easily be shown:

$$\begin{aligned} \mathbf{WH} &= \left(\begin{array}{c|c} \mathbf{A} & \mathbf{X}^\intercal \\ \hline \mathbf{X} & \mathbf{0} \end{array}\right)\left(\begin{array}{c|c} \Omega & \Xi^\intercal \\ \hline \Xi & \Upsilon \end{array}\right) \\ &= \left(\begin{array}{c|c} \mathbf{A}\Omega + \mathbf{X}^\intercal\Xi & \mathbf{A}\Xi^\intercal + \mathbf{X}^\intercal\Upsilon \\ \hline \mathbf{X}\Omega & \mathbf{X}\Xi^\intercal \end{array}\right) = \mathbf{I} \end{aligned}$$

where the last equality is due to $\mathbf{H}^{-1} = \mathbf{W}$. The fact that $\mathbf{X}\Omega$ should be zero and the rank of $\mathbf{X}$ is $n+1$ creates a bound on the rank of Ω. The rank can not be higher than $m-n-1$. If Ω has the rank $m-n-1$, then the bottom right submatrix of dimension $(n+1)\times(n+1)$ of $\mathbf{H}^{-1}$ is zero[9]. Which solves the problem of the accumulated errors in the bottom right submatrix of Δ mentioned earlier. This property is guaranteed by using the factorization of Ω given below and the proceeding updating formulas, which makes sure the rank of Ω

is $m - n - 1$.

The Ω matrix is factorized as shown:

$$\begin{aligned}\Omega &= \sum_{k=1}^{m-n-1} s_k \mathbf{z}_k \mathbf{z}_k^{\mathsf{T}} \\ &= \mathbf{Z}\mathbf{S}\mathbf{Z}^{\mathsf{T}}\end{aligned}$$

where $\mathbf{Z}$ contains the column vectors $\mathbf{z}_k$, and $\mathbf{S}$ is a diagonal matrix of the s_k's. The s_k's are either -1 or $+1$, and it is only -1 if floating point errors have previously occurred. This can be seen by looking at the updating formula for the s_k's, i.e., equation (3.17). As previously mentioned, it is proved that σ_t will be a positive number in exact arithmetic. Thus, if any of the elements of $\mathbf{S}$ is -1 after an update, then errors due to the limitation of precision of floating points have occurred.

If the t-th element of $\mathbf{z}_k$ is zero, then

$$s_k^+ = s_k \tag{3.14}$$
$$\mathbf{z}_k^+ = \mathbf{z}_k \tag{3.15}$$

Let $\mathcal{K}$ be the set containing those indices (the k's). Before the updating is started, an elementary change is made, if necessary, to the terms of the sum, which forces the cardinality of $\mathcal{K}$ to be greater or equal $m - n - 3$. If $s_i = s_j$, then the following orthogonal rotation can be done to $\mathbf{z}_i$ and $\mathbf{z}_j$ without changing the value of Ω:

$$\mathbf{z}_i := \cos(\theta)\mathbf{z}_i + \sin(\theta)\mathbf{z}_j \tag{3.16a}$$
$$\mathbf{z}_j := -\sin(\theta)\mathbf{z}_i + \cos(\theta)\mathbf{z}_j \tag{3.16b}$$

for any $\theta \in [0, 2\pi]$. Thus, either i or j can be put into $\mathcal{K}$ by forcing the t-th element of either of the $\mathbf{z}$'s to be zero. This means that a maximum of two new vectors must be calculated after retaining the values in (3.15). If the cardinality of $\mathcal{K}$ is $m - n - 2$, then the following updating formula is used:

$$s_u^+ = \text{sign}(\sigma_t) s_u \tag{3.17}$$
$$\mathbf{z}_u^+ = |\sigma_t|^{-1/2}\{\tau_t \mathbf{z}_u + \mathbf{Z}_{t,u}\text{chop}(\mathbf{e}_t - \mathbf{H}\mathbf{w})\} \tag{3.18}$$

where $\mathbf{z}_u$ denotes the only $\mathbf{z}$ vector that has a nonzero element in the t-th component, $\mathbf{Z}_{t,u}$ is the t-th element of $\mathbf{z}_u$, chop() keeps the first m elements of the vector and the other paramerets are as before.

In the other case, when the cardinality of $\mathcal{K}$ is $m - n - 3$ (i.e., one of the s_k's is -1), then the updating formula depends upon the sign of β_t. Let $\mathbf{z}_u$ and $\mathbf{z}_v$ be the two vectors

that are not given by (3.15), and $s_u = 1, s_v = -1$. If β_t is non-negative:

$$\begin{aligned}
\zeta &= \tau_t^2 + \beta_t \mathbf{Z}_{t,u}^2 \\
s_u^+ &= s_u = 1 \\
s_v^+ &= \text{sign}(\sigma_t) s_v = -\text{sign}(\sigma_t) \\
\mathbf{z}_u^+ &= |\zeta|^{-1/2} \{\tau_t \mathbf{z}_u + \mathbf{Z}_{t,u} \text{chop}(\mathbf{e}_t - \mathbf{H}\mathbf{w})\} \\
\mathbf{z}_v^+ &= |\zeta\sigma|^{-1/2} \{-\beta_t \mathbf{Z}_{t,u} \mathbf{Z}_{t,v} \mathbf{z}_u + \zeta \mathbf{z}_v + \tau_t \mathbf{Z}_{t,v} \text{chop}(\mathbf{e}_t - \mathbf{H}\mathbf{w})\},
\end{aligned}$$

while if $\beta_t < 0$:

$$\begin{aligned}
\zeta &= \tau_t^2 - \beta_t \mathbf{Z}_{t,v}^2 \\
s_u^+ &= \text{sign}(\sigma_t) s_u = \text{sign}(\sigma_t) \\
s_v^+ &= s_v = -1 \\
\mathbf{z}_1^+ &= |\zeta\sigma_t|^{-1/2} \{\zeta \mathbf{z}_u + \beta_t \mathbf{Z}_{t,u} \mathbf{Z}_{t,v} \mathbf{z}_v + \tau_t \mathbf{Z}_{t,u} \text{chop}(\mathbf{e}_t - \mathbf{H}\mathbf{w}) \\
\mathbf{z}_v^+ &= |\zeta|^{-1/2} \{\tau_t \mathbf{z}_v + \mathbf{Z}_{t,v} \text{chop}(\mathbf{e}_t - \mathbf{H}\mathbf{w})\}.
\end{aligned}$$

To conclude, equation (3.13) is still used to update $\boldsymbol{\Xi}$ and $\boldsymbol{\Upsilon}$, while the just stated formulas are used to update the factorization of Ω. Before we continue, some information about how to decide the θ is provided. There is atleast two ways to find the value. The first one is the easiest and most straight forward method.

Let $\mathbf{z}_k^i$ and $\mathbf{z}_k^j$ be the k-th element of the two vectors. atan2$(\mathbf{y}, \mathbf{x})$ is the inverse tangent that takes into the consideration where the points are placed and gives a value between $\pm\pi$. Now, let $\tilde{\theta} = \text{atan2}(\mathbf{z}_k^i, \mathbf{z}_k^j)$. If $\tilde{\theta} < 0$, then $\theta = \tilde{\theta} + \pi$. Else, $\theta = \tilde{\theta}$. The new vectors, are then given by:

$$\begin{aligned}
\mathbf{z}_{\text{new}}^i &:= \cos(\theta)\mathbf{z}^i + \sin(\theta)\mathbf{z}^j \\
\mathbf{z}_{\text{new}}^j &:= -\sin(\theta)\mathbf{z}^i + \cos(\theta)\mathbf{z}^j
\end{aligned}$$

The other method relies on the definition of $\sin$ and $\cos$ and is the preferred method as it doesn't require the calculation of θ.

$$\sin(\theta) = \frac{\mathbf{z}_k^j}{\sqrt{(\mathbf{z}_k^j)^2 + (\mathbf{z}_k^i)^2}} \qquad \cos(\theta) = \frac{\mathbf{z}_k^i}{\sqrt{(\mathbf{z}_k^j)^2 + (\mathbf{z}_k^i)^2}}$$

Now the updating is straight forward as both the unknowns are determined. Both of these methods will put a zero at the k-th element of $\mathbf{z}^j$. The last method is the method that Powell used in his implementation of NEWOUA[10], and the solution was found by inspecting his code. The code is available (pr. 29.06.2018) at `http://mat.uc.pt/~zhang/software/newuoa.zip`.

All the needed details to update the inverse H matrix are given and we can now proceed with a computational budget friendly method to extract the Lagrange polynomials.

Extract the Lagrange polynomials

Here we will show how to create the Lagrange polynomials almost for free. Note that the $\mathbf{W}$ matrix is solely determined by the interpolation points. This means that the inverse of that matrix also does not depend upon the right hand side of (3.10). In other words, we can extract the Lagrange polynomials directly out of the matrix because multiplying the inverse matrix with a unit vector is the same as extracting the corresponding column of that matrix. Thus, all information belonging to the t-th Lagrange polynomial are stored in column t of $\mathbf{H}$.

$$\mathbf{G}_t = \nabla^2 \ell_t = \sum_{k=1}^{m} \mathbf{H}_{k,t}(\mathbf{y}_k - \mathbf{y}_0)(\mathbf{y}_k - \mathbf{y}_0)^\intercal \tag{3.20a}$$

$$c_t = \mathbf{H}_{m+1,t} \tag{3.20b}$$

$$\mathbf{g}_t = \begin{pmatrix} \mathbf{H}_{m+2,t} \\ \mathbf{H}_{m+3,t} \\ \vdots \\ \mathbf{H}_{m+n+1,t} \end{pmatrix} \tag{3.20c}$$

$$\ell_t(\mathbf{x}) = c_t + \mathbf{g}_t^\intercal(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^\intercal \mathbf{G}_t(\mathbf{x} - \mathbf{y}_0) \tag{3.20d}$$

To keep the computational effort at a minimum when calculating $\ell_t(\mathbf{x}^+)$, Powell suggests to use the following procedure. First calculate:

$$\sigma_k = (\mathbf{y}_k - \mathbf{y}_0)^\intercal(\mathbf{x}^+ - \mathbf{y}_0), \quad k = 1, 2, \ldots, m$$

and then compute

$$\ell_j(\mathbf{x}^+) = c_j + \mathbf{g}_j^\intercal(\mathbf{x}^+ - \mathbf{y}_0) + \frac{1}{2}\sum_{k=1}^{m} \mathbf{H}_{k,j}\sigma_k^2, \quad j = 1, 2, \ldots, m$$

Next we will show how to efficiently store and update the quadratic model.

Update the stored quadratic model

After the $\mathbf{H}^+$ matrix is created, the model needs to be updated too. As briefly mentioned, $Q^\#$ is a multiple of $\ell_t^+(\mathbf{x})$. Combining this with the formulas in (3.20), but with $\mathbf{H}^+$ instead of $\mathbf{H}$, the following updating formulas are obtained[8]:

$$c^+ = c + (f(\mathbf{x}^+) - Q(\mathbf{x}^+))\mathbf{H}_{m+1,t}^+ \tag{3.21a}$$

$$\mathbf{g}_j^+ = \mathbf{g}_j + (f(\mathbf{x}^+) - Q(\mathbf{x}^+))\mathbf{H}_{m+j+1,t}^+, \quad j = 1, 2, \ldots, n \tag{3.21b}$$

$$\mathbf{G}^+ = \mathbf{G} + (f(\mathbf{x}^+) - Q(\mathbf{x}^+))\sum_{j=1}^{m} \mathbf{H}_{k,t}^+(\mathbf{y}_k^+ - \mathbf{y}_0)(\mathbf{y}_k^+ - \mathbf{y}_0)^\intercal \tag{3.21c}$$

Using the last formula for updating $\mathbf{G}^+$ is possible, however, the computational cost will be higher than appreciated, and thus an alternative method is proposed.

$$\mathbf{G} = \Gamma + \sum_{k=1}^{m} \gamma_k (\mathbf{y}_k - \mathbf{y}_0)(\mathbf{y}_k - \mathbf{y}_0)^\intercal \tag{3.22a}$$

$$\Gamma^+ = \Gamma + \gamma_t (\mathbf{y}_t - \mathbf{y}_0)(\mathbf{y}_t - \mathbf{y}_0)^\intercal \tag{3.22b}$$

$$\gamma_k^+ = \gamma_k (1 - \delta_{kt}) + (f(\mathbf{x}^+) - Q(\mathbf{x}^+))\mathbf{H}_{k,t}^+, \quad k = 1, 2, \ldots, m \tag{3.22c}$$

where we only store and update the γ_k's and Γ. If the second derivative matrix is needed, it can be calculated using the above formula. However, the idea is to never calculate it explicitly. While solving the subproblem, the Hessian matrix should be used. If the optimization procedure uses an algorithm which multiplies the Hessian by a vector, the computational cost is reduced (compared to first creating the entire matrix and then do the multiplication by the vector). This suggests that using an optimization algorithm which does not rely on computing eigenvalues of the Hessian is a smart choice, e.g., some version of the (truncated) conjugate gradient method could be used.

The goal for the total computational effort of the overall updating algorithm is $\mathcal{O}(m^2)$. If the updating formula for $\mathbf{G}$ in (3.21c) was used, then the cost would be $\mathcal{O}(mn^2)$ because m matrices of rank one would be added to $\mathbf{G}$. The alternative method obeys the goal of a cost of no more than $\mathcal{O}(m^2)$. Multiplying the expression (3.22a) by a vector, as will be done in a conjugate gradient method, is done in $\mathcal{O}(mn)$ which is equal to $\mathcal{O}(m^2)$ in the case of $m = 2n + 1$, which is Powell's recommended value for m.

3.3.5 Formulas for shifting the center point

Also here an in-depth derivation is not of immediate interest. I refer the interested reader to the original paper [8]. However, a brief description will be given. The computationally effort of changing the center point by the method given below, is no less than to just simply calculate the inverse of the new $\mathbf{W}$ matrix. However, Powell states that using the presented method has some advantages, where one of them is that the Ω matrix (and its decomposition) will be unaffected by the change of the center point. (The paper proofs that the Ω submatrix is independent of the center point.) The shifting is only necessary because of the limitation of precision of the computers. Doing a shift is computationally expensive $\mathcal{O}((n+m)^3)$ and should only be applied when needed, remember that the cost of updating is no more than $\mathcal{O}((n+m)^2)$. Deciding when it is needed is not straight forward, but Powell suggest to perform a shift of the center point when the ratio $||\mathbf{y}_0 - \mathbf{y}_b||/\Delta_k$ becomes "large", where $\mathbf{y}_b$ is the yet best found point and Δ_k is the current trust-region

radius. Let Ω_X and Ω_A be the following $(m+n+1)\times(m+n+1)$ matrices:

$$\Omega_X = \left(\begin{array}{c|c|c} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 1 & \mathbf{0} \\ \hline \mathbf{0} & -\frac{1}{2}\mathbf{s} & \mathbf{I} \end{array}\right), \quad (\Omega_X^\intercal)^{-1} = \left(\begin{array}{c|c|c} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 1 & \frac{1}{2}\mathbf{s}^\intercal \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{I} \end{array}\right)$$

$$\Omega_A = \left(\begin{array}{c|c|c} \mathbf{I} & \mathbf{0} & -\mathbf{P}^\intercal \\ \hline \mathbf{0} & 1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{I} \end{array}\right), \quad (\Omega_A^\intercal)^{-1} = \left(\begin{array}{c|c|c} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & 1 & \mathbf{0} \\ \hline \mathbf{P} & \mathbf{0} & \mathbf{I} \end{array}\right)$$

where $\mathbf{s}$ is the distance between the new, $\mathbf{y}_0^+$, and old center point: $\mathbf{s} = \mathbf{y}_0^+ - \mathbf{y}_0$ and $\mathbf{P}$ is the $n \times m$ matrix containing the following $\mathbf{p}_k$ column vectors:

$$\mathbf{u}_k = \mathbf{y}_k - \mathbf{y}_0 - \frac{1}{2}\mathbf{s} \tag{3.23a}$$

$$\mathbf{p}_k = (\mathbf{s}^\intercal \mathbf{u}_k)\mathbf{u}_k + \frac{1}{4}||\mathbf{s}||^2\mathbf{s}, \quad k = 1, 2, \ldots, m \tag{3.23b}$$

The reason why the vectors in (3.23) are chosen is because they give the convenient expression in (3.24). The vectors provides the following relationship between the difference of the old and new $\mathbf{A}$ matrices [8].

$$\begin{aligned} &\mathbf{A}_{ik}^{\text{new}} - \mathbf{A}_{ik}^{\text{old}} \\ &\quad= \frac{1}{2}\{(\mathbf{y}_i - \mathbf{y}_0 - \mathbf{s})^\intercal(\mathbf{y}_k - \mathbf{y}_0 - \mathbf{s})\}^2 - \frac{1}{2}\{(\mathbf{y}_i - \mathbf{y}_0)^\intercal((\mathbf{y}_k - \mathbf{y}_0))\}^2 \\ &\quad= \frac{1}{2}\{(\mathbf{u}_i - \frac{1}{2}\mathbf{s})^\intercal(\mathbf{u}_k - \frac{1}{2}\mathbf{s})\}^2 - \frac{1}{2}\{(\mathbf{u}_i + \frac{1}{2}\mathbf{s})^\intercal(\mathbf{u}_k + \frac{1}{2}\mathbf{s})\}^2 \\ &\quad= \frac{1}{2}\{-\mathbf{s}^\intercal\mathbf{u}_k - \mathbf{s}^\intercal\mathbf{u}_i\}\{2\mathbf{u}_i^\intercal\mathbf{u}_k + \frac{1}{2}||\mathbf{s}||^2\} \\ &\quad= -\mathbf{p}_k^\intercal\mathbf{u}_i - \mathbf{p}_i^\intercal\mathbf{u}_k, \qquad \text{for } 1 \le i, k \le m. \end{aligned}$$

This means that the new $\mathbf{W}$ matrix, $\mathbf{W}^+$, can be calculated by the following multiplications:

$$\mathbf{W}^+ = \Omega_X\Omega_A\Omega_X\mathbf{W}\ \Omega_X^\intercal\Omega_A^\intercal\Omega_X^\intercal$$

Which in turn implies the end results given below. The $\mathbf{H}$ matrix after the shift, $\mathbf{H}^+$, can be calculated as:

$$\mathbf{H}^+ = (\Omega_X^\intercal)^{-1}(\Omega_A^\intercal)^{-1}(\Omega_X^\intercal)^{-1}\mathbf{H}\ \Omega_X^{-1}\Omega_A^{-1}\Omega_X^{-1} \tag{3.24}$$

A practical way of doing this multiplication is given in [8]. After the shift, some changes to the model parameters must also be done. The changes are found in [10].

$$\begin{aligned} \mathbf{g}^+ &= \mathbf{g} + \mathbf{G}\mathbf{s} \\ \mathbf{\Gamma}^+ &= \mathbf{\Gamma} + \mathbf{v}\mathbf{s}^\intercal + \mathbf{s}\mathbf{v}^\intercal \end{aligned}$$

where $\mathbf{v} = \sum_{j=1}^{m} \gamma_j(\mathbf{x}_j - \mathbf{y}_0^+ + \frac{1}{2}\mathbf{s})$. It might look like that the second derivative matrix is altered, but it is not. That matrix is independent of shifts. The performed changes must be done because of how the matrix is stored (i.e., how it is factorized). The changes to the constant of the model are not given in that paper as Powell is using the Lagrangian polynomials without their constant terms. "Ignoring" the term (by using a difference instead of the polynomial itself) will provide better floating point arithmetic. However, the model improvement algorithm given later uses the polynomials as they are. Using a difference is perhaps possible, but this idea is not further explored. Therefore, the changes to the constant are derived here. Let Q_1 and Q_2 be the old and new model, respectively.

$$Q_1(\mathbf{x}) = c_1 + \mathbf{g}_1^\intercal(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^\intercal \mathbf{G}_1(\mathbf{x} - \mathbf{y}_0)$$
$$Q_2(\mathbf{x}) = c_2 + \mathbf{g}_2^\intercal(\mathbf{x} - \mathbf{y}_0 - \mathbf{s}) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0 - \mathbf{s})^\intercal \mathbf{G}_2(\mathbf{x} - \mathbf{y}_0 - \mathbf{s})$$

Both models should provide the same output for the same input.

$$Q_1(\mathbf{y}_0 + \mathbf{s}) = c_1 + \mathbf{g}_1^\intercal \mathbf{s} + \frac{1}{2}\mathbf{s}^\intercal \mathbf{G}_1 \mathbf{s}$$
$$Q_2(\mathbf{y}_0 + \mathbf{s}) = c_2$$
$$\implies c_2 = c_1 + \mathbf{g}_1^\intercal \mathbf{s} + \frac{1}{2}\mathbf{s}^\intercal \mathbf{G}_1 \mathbf{s}$$

3.4 Gradient enhanced interpolation model

This section will suggest two ideas on how to include the available gradients into the model-making process. The first part will suggest how to set the gradients at the center point of the model, whereas the last idea will be able to set elements of the Hessian matrix.

3.4.1 Set the true gradients at the center point of the model

Let's now assume that the function we are trying to minimize also provides some partial derivatives. Say that it provides the partial derivative of f with respect to the n_g last elements of $\mathbf{x}$. The goal is to include this information in a similar system of equations as the one just presented, i.e., as the one in 3.10. This method does not fully take advantage of the available derivatives, as it only uses this information to set the gradient of the model. The derivatives are only used if they belong to the center point of the model, $\mathbf{y}_0$. Using the available derivatives from the other sample points should be possible to help determine the Hessian matrix. However, doing so in Powell's updating scheme in [8] is not straight forward.

Let

$$\mathbf{x} = \begin{pmatrix} \tilde{\mathbf{x}} \\ \mathbf{x}_g \end{pmatrix} \quad \text{and} \quad \mathbf{g} = \begin{pmatrix} \tilde{\mathbf{g}} \\ \mathbf{g}_g \end{pmatrix}$$

where $\mathbf{g}_g$ and $\mathbf{x}_g$ are vectors of length n_g. $\mathbf{g}_g$ contains the partial derivatives (with respect to $\mathbf{x}_g$) that are provided by the objective function. The following derivation is almost the

same as the one given in the last section. However, it is included to prove that this new system can be put into the same system of equations as the other one. This let's us use Powell's method of updating and handling the inverse matrix (the $\mathbf{H}$ matrix). In addition, the derivation nor the result are, as far as the author knows, not available elsewhere.

The interpolation conditions are as before:

$$c + \mathbf{g}^\intercal(\mathbf{y}_i - \mathbf{y}_0) + \frac{1}{2}(\mathbf{y}_i - \mathbf{y}_0)^\intercal \mathbf{G}(\mathbf{y}_i - \mathbf{y}_0) = f(\mathbf{y}_i) \tag{3.25}$$

for $i = 1, 2, \ldots, m$. However, this time the last n_g elements of $\mathbf{g}$ is determined by some additional information. This means that we must eliminate those as variables in the optimization problem[4]. Let $\mathbf{y}_0$ be the zero-vector for now. This is only done to ease the notation. It will be reintroduced shortly. Here we take (3.25) and insert the known information:

$$c + \mathbf{g}^\intercal \mathbf{y}_i + \frac{1}{2}\mathbf{y}_i^\intercal \mathbf{G}\mathbf{y}_i = f(\mathbf{y}_i) \tag{3.26}$$

$$c + \begin{pmatrix} \tilde{\mathbf{g}} \\ \mathbf{g}_g \end{pmatrix}^\intercal \begin{pmatrix} \tilde{\mathbf{y}_i} \\ \mathbf{y}_g \end{pmatrix} + \frac{1}{2}\mathbf{y}_i^\intercal \mathbf{G}\mathbf{y}_i = f(\mathbf{y}_i) \tag{3.27}$$

$$c + \tilde{\mathbf{g}}^\intercal \tilde{\mathbf{y}_i} + \frac{1}{2}\mathbf{y}_i^\intercal \mathbf{G}\mathbf{y}_i = f(\mathbf{y}_i) - \mathbf{g}_g^\intercal \mathbf{y}_g \tag{3.28}$$

The same optimization problem is formulated, where the change in the squared Frobenius norm of the Hessian matrix is minimized. To avoid having too many super- and subscripts, we will assume that all the coefficients of the old model were zeros, such that all the #'s can be removed. In addition, we skip using the + superscript because we do not have both an old and a new model to separate between, we only have the new model. This assumption will not impact anything other than the right hand side of the interpolation conditions. To remove this assumption later on, all that is needed is to subtract the output of the old model evaluated at the interpolation points from the right hand side of the interpolation conditions.

$$\begin{aligned} \min_{\mathbf{G}_{ij}, \mathbf{g}, c} \quad & \frac{1}{2}||\mathbf{G}||_F^2 = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} {\mathbf{G}_{ij}}^2 \\ \text{s.t.} \quad & \\ & c + \mathbf{g}^\intercal \mathbf{y}_i + \frac{1}{2}\mathbf{y}_i^\intercal \mathbf{G}\mathbf{y}_i = f(\mathbf{y}_i) \qquad i = 1, 2, \ldots, m \end{aligned} \tag{3.29}$$

The Lagrangian function is the same as before, except for that the number of variables is

[4] They could be included as constraints instead.

lower.

$$\begin{aligned}\mathcal{L}(c, \tilde{\mathbf{g}}, \mathbf{G}) &= \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}{\mathbf{G}_{ij}}^2 \\ &\quad - \sum_{k=1}^{m}\lambda_k\left(c + \mathbf{g}^\intercal\mathbf{y}_k + \frac{1}{2}\mathbf{y}_k^\intercal\mathbf{G}\mathbf{y}_k - f(\mathbf{y}_k)\right) \\ &= \frac{1}{4}\sum_{i=1}^{n}\sum_{j=1}^{n}{\mathbf{G}_{ij}}^2 \\ &\quad - \sum_{k=1}^{m}\lambda_k\left(c + \tilde{\mathbf{g}}^\intercal\tilde{\mathbf{y}}_k + \frac{1}{2}\mathbf{y}_k^\intercal\mathbf{G}\mathbf{y}_k - (f(\mathbf{y}_k) - \mathbf{g}_g^\intercal\mathbf{y}_{g,k})\right)\end{aligned}$$

Where we have used equations (3.26) and (3.28) in the last equality. The partial derivatives are found:

$$\frac{\partial\mathcal{L}}{\partial c} = \sum_{k=1}^{m}\lambda_k = 0 \tag{3.30a}$$

$$\frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{g}}} = \sum_{k=1}^{m}\lambda_k\tilde{\mathbf{y}}_k = 0 \tag{3.30b}$$

$$\frac{\partial\mathcal{L}}{\partial\mathbf{G}} = \sum_{k=1}^{m}\lambda_k\mathbf{y}_k\mathbf{y}_k^\intercal - \mathbf{G} = 0 \tag{3.30c}$$

The only change is in (3.30b) compared to (3.8), where now only the upper $n - n_g$ part of $\mathbf{y}_k$ is present. As before, this can be formulated with matrices. The center point of the model is now allowed to be any vector. Let

$$\hat{\mathbf{Y}} = \begin{bmatrix}\tilde{\mathbf{Y}} \\ \mathbf{Y}_g\end{bmatrix}$$

such that all variables that have gradients readily available are stored in the $n_g \times m$ matrix, $\mathbf{Y}_g$. Remember that the center point is subtracted from each column vector in $\hat{\mathbf{Y}}$.

The matrix form of the linear system of equations is then:

$$\left(\begin{array}{c|ccc} \mathbf{A} & \mathbf{e} & \vdots & \tilde{\mathbf{Y}}^\intercal \\ \hline \mathbf{e}^\intercal & & & \\ \cdots & & \mathbf{0} & \\ \tilde{\mathbf{Y}} & & & \end{array}\right)\left(\begin{array}{c} \lambda \\ \hline c \\ \cdots \\ \tilde{\mathbf{g}} \end{array}\right) = \left(\begin{array}{c} \mathbf{F} - {\mathbf{g}_g}^\intercal\mathbf{Y}_g \\ \hline 0 \\ \cdots \\ \mathbf{0} \end{array}\right), \tag{3.31}$$

The $\mathbf{A}$ matrix remains unchanged. The $\mathbf{F}$ vector is as defined earlier, namely it contains the output of the true function evaluated at the interpolation points subtracted the old model

evaluated at the same points. In the case of the model having all zero coefficients, simply subtract 0 as the evaluated model value.

We see that (3.31) is on the same form as (3.10). That means that it should be possible to use the same robust updating method of the inverse matrix. However, the matrix dimension is n_g rows and columns smaller, because $\tilde{\mathbf{Y}}$ is used instead of $\hat{\mathbf{Y}}$. This means that for example we can not extract the Lagrange polynomials as before because the n_g last elements of the gradients will not be specified by that system. Nontheless, the method should be able to provide you with the quadratic model information. One possibility is to simply have two set of systems of equations, one which will provide the Lagrange polynomials and the regular quadratic model, and one set which will provide the gradient enhanced model. We will not dwell into the details about how to do the updating of the $\mathbf{H}$ matrix in the case when some gradients are specified because another method to deal with gradients will be presented shortly. However, the method in [8] is applicable if we take some care, and go one step back and make sure that the steps are still valid. For example, care must be taken when some formulas are simplified to using Lagrange polynomials, these simplifications should not be done.

3.4.2 Including the remaining available gradients

As already mentioned, the previous method only takes advantage of the gradient at the center point of the model. There is still m[5] gradients available which is never used. It would be interesting to try to include this information into the model-making process. Unless we are extremely lucky with the directions of the gradients we cannot create a model that will satisfy all the interpolation conditions and all the gradients. This fact leads us towards the idea of solving some kind of minimization problem. Let $\mathbf{d}^i$ represent the vector $(\mathbf{y}^i - \mathbf{y}_0)$, where i is the interpolation point number.

$$Q(\mathbf{x}) = c + \mathbf{g}^\intercal(\mathbf{x} - \mathbf{y}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{y}_0)^\intercal \mathbf{H}(\mathbf{x} - \mathbf{y}_0) \tag{3.32}$$

$$\nabla_x Q(\mathbf{x}) = \mathbf{g} + \mathbf{H}(\mathbf{x} - \mathbf{y}_0) = \nabla_x f(\mathbf{x}) \tag{3.33}$$

$$\nabla_x Q(\mathbf{d}) = \mathbf{g} + \mathbf{H}\mathbf{d} = \nabla_x f(\mathbf{x}) \tag{3.34}$$

Where $\nabla_x f(\mathbf{x})$ is the gradient of the true function. If we had all the derivatives of the objective function (i.e., $n_g = n$), we would have m equations of the form (3.34). Let's write this equation explicitly for the case when $n = 5$.

$$\begin{pmatrix} \nabla_x f_1 \\ \nabla_x f_2 \\ \nabla_x f_3 \\ \nabla_x f_4 \\ \nabla_x f_5 \end{pmatrix} = \begin{pmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \end{pmatrix} + \begin{pmatrix} h_{11} & h_{21} & h_{31} & h_{41} & h_{51} \\ h_{21} & h_{22} & h_{32} & h_{42} & h_{52} \\ h_{31} & h_{32} & h_{33} & h_{43} & h_{53} \\ h_{41} & h_{42} & h_{43} & h_{44} & h_{54} \\ h_{51} & h_{52} & h_{53} & h_{54} & h_{55} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix} \tag{3.35}$$

[5] or $(m-1)$. It depends upon if the center point is included in the set of interpolation points. We will hereby assume that it is not included in the set.

Let's say we have 5 variables, and gradient information is available for the 4 last of them. Using the method in the previous section will set the gradient at the center point of the model. Here we have inserted the first sample point, $i = 1$, into the equation (3.35) and only extracted the equations which have a new gradient (i.e., $\nabla_x f_2^i$ is available) which we want to incorporate into the model-making process:

$$\begin{pmatrix} d_1 & d_2 & d_3 & d_4 & d_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_5 & 0 & d_1 & d_2 & d_3 & d_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_5 & 0 & 0 & 0 & 0 & d_4 & 0 & d_1 & d_2 & d_3 & 0 & 0 \\ 0 & d_5 & 0 & 0 & 0 & 0 & d_4 & 0 & 0 & 0 & d_3 & 0 & d_1 & d_2 \end{pmatrix} \begin{pmatrix} h_{51} \\ h_{52} \\ h_{53} \\ h_{54} \\ h_{55} \\ h_{41} \\ h_{42} \\ h_{43} \\ h_{44} \\ h_{31} \\ h_{32} \\ h_{33} \\ h_{21} \\ h_{22} \end{pmatrix} = \begin{pmatrix} \nabla_x f_2^1 - g_2 \\ \nabla_x f_3^1 - g_3 \\ \nabla_x f_4^1 - g_4 \\ \nabla_x f_5^1 - g_5 \end{pmatrix}$$

If the system of equations is expanded with the second, $i = 2$, interpolation point, it becomes:

$$\begin{pmatrix} d_1^1 & d_2^1 & d_3^1 & d_4^1 & d_5^1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ d_1^2 & d_2^2 & d_3^2 & d_4^2 & d_5^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_5^1 & 0 & d_1^1 & d_2^1 & d_3^1 & d_4^1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_5^2 & 0 & d_1^2 & d_2^2 & d_3^2 & d_4^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_5^1 & 0 & 0 & 0 & 0 & d_4^1 & 0 & d_1^1 & d_2^1 & d_3^1 & 0 & 0 \\ 0 & 0 & d_5^2 & 0 & 0 & 0 & 0 & d_4^2 & 0 & d_1^2 & d_2^2 & d_3^2 & 0 & 0 \\ 0 & d_5^1 & 0 & 0 & 0 & 0 & d_4^1 & 0 & 0 & 0 & d_3^1 & 0 & d_1^1 & d_2^1 \\ 0 & d_5^2 & 0 & 0 & 0 & 0 & d_4^2 & 0 & 0 & 0 & d_3^2 & 0 & d_1^2 & d_2^2 \end{pmatrix} \begin{pmatrix} h_{51} \\ h_{52} \\ h_{53} \\ h_{54} \\ h_{55} \\ h_{41} \\ h_{42} \\ h_{43} \\ h_{44} \\ h_{31} \\ h_{32} \\ h_{33} \\ h_{21} \\ h_{22} \end{pmatrix} = \begin{pmatrix} \nabla_x f_2^1 - g_2 \\ \nabla_x f_2^2 - g_2 \\ \nabla_x f_3^1 - g_3 \\ \nabla_x f_3^2 - g_3 \\ \nabla_x f_4^1 - g_4 \\ \nabla_x f_4^2 - g_4 \\ \nabla_x f_5^1 - g_5 \\ \nabla_x f_5^2 - g_5 \end{pmatrix}$$

This expansion can be done for all of the m points. Let's denote the system which include all of the points as $\mathbf{D}$. Further, let $\mathbf{h}$ be the vector containing the elements of the Hessian which are relevant (i.e., those elements that can be set by the available gradient information. In other words, those that are part of the n_g last equations of (3.35)), and let $\mathbf{v}$ be the right hand side of the equation.

The idea is to minimize the squared norm of this difference, i.e, $||\mathbf{Dh} - \mathbf{v}||_2^2$. This objective can be included straight into the minimization problem (3.29) as follows:

$$\begin{aligned} \min_{\mathbf{G}_{ij},\mathbf{g},c} \quad & \alpha\frac{1}{2}||\mathbf{G}||_F^2 + (1-\alpha)\frac{1}{2}||\mathbf{Dh} - \mathbf{v}||_2^2 \\ \text{s.t.} \quad & \\ & c + \mathbf{g}^\intercal\mathbf{y}_i + \frac{1}{2}\mathbf{y}_i^\intercal\mathbf{G}\mathbf{y}_i = f(\mathbf{y}_i) \qquad i = 1, 2, \ldots, m \end{aligned} \tag{3.36}$$

where α is a weighting factor between 0 and 1. The problem remains convex as long as α is in the open interval $(0, 1]$ because the additional term is also convex[6]. It is important that $\alpha > 0$ unless $n = n_g$ because if $\alpha = 0$ and $n \neq n_g$ then some of the elements of the Hessian won't be included. The problem in (3.36) can not (as far as the author can see) be transformed into a system of equations of the form (3.31).

If the Lagrangian is set up and differentiated as before, the only change will be in the $\frac{\partial\mathcal{L}}{\partial\mathbf{G}}$ term (compared to (3.30a)). For those elements of $\mathbf{G}$ that are found in $\mathbf{h}$ will add another term. The symmetry of $\mathbf{G}$ that was automatically achieved due to (3.8c) is no longer present. Thus, the symmetry must be imposed otherwise. In addition, because of the change in the term $\frac{\partial\mathcal{L}}{\partial\mathbf{G}} = 0$ the $\mathbf{G}$ matrix can not be replaced by the λ's.

The idea is to accept that it will be slightly more complicated, and simply put all the equations that arises from the derivatives of the Lagrangian and in the interpolation conditions into a linear system of equations. There will be more equations as we will need to determine the Lagrangian multipliers in addition to the Hessian matrix. The exact setup is not shown here because it involves some uninteresting index arithmetic. The way the symmetry is imposed is by "removing" the lower left subdiagonal of the Hessian, and every time those variables are supposed to be used, the one from the upper right part is used instead.

This concludes the discussion about how to include the gradients into the model-making process. In Section 3.6, some ideas regarding how efficient it will be to include gradients will be shared. In the next Section we will see how constraints can be included into the derivative-free model-based trust-region algorithm.

3.5 Constraint handling

A desired functionality when dealing with optimization is the capability of handling constraints. If we were lucky, we could solve the well-placement challenge without specifying bounds. However, this implies that we must always (by luck) choose feasible points to evaluate and that the (local) solution of the unconstrained problem is the same as of the constrained problem.

[6]It is the exact same type as the previous term. The sum of two convex functions is a convex function.

3.5.1 Incorporate constraints into the algorithm

There are different approaches one can take to include constraints into a derivative-free trust-region model-based optimization method. Constraints can be split into two categories *hard* and *soft* (or *unrelaxable* and *relaxable*). A hard constraint is a restriction that cannot be violated. In other words, the constraint must be satisfied at all times. E.g., the bounds of the reservoir are hard constraints. A soft constraint is a restriction that is desired, but it can be violated without any catastrophically consequences. E.g., the desired regions for the different wells or the restriction of allowable change in the bottom hole pressure.

Let's assume we only have hard constraints with available gradients[7]. The perhaps most straight forward idea is to simply intersect the feasible area defined by the constraints with the trust-region area. This will allow us to only generate feasible sample points. However, both solving the subproblem and doing the optimization of the Lagrange polynomials might become a lot more difficult. The Lagrange polynomials are optimized in the process of improving the geometry of the set of interpolation points, as we will see in a later section. Nonetheless, if we only have bounds and linear constraints the authors of [1] states that the convergence theory should be applicable after some adaptation.

Soft constraints without available gradients must be handled differently. Because they are soft, they will have no impact on the geometry of the interpolation set[1]. This is because the points that violate the soft constraint are just "less desired", but still feasible. An easy and straightforward way of handling this type is as follows: Combine the objective function and the soft constraints into a merit function and use that as the input to the derivative-free optimization algorithm. Following that approach, the convergence theory for the unconstrained case can be used without any modifications[1]. Sometimes the constraints are in fact additional objective functions and the user has some knowledge on how to combine them into one penalty function, which makes this approach desirable. Except for the complication of having to find a sensible penalty function, this method might seem like a good option. However, the final solution of the algorithm might violate the soft constraints.

We might want to add some constraints that depends upon the simulator. In other words, adding restrictions on one or more of the outputs of the simulator could be of interest. An example is that we would like to put a limitation on how much gas we are allowed to produce. To deal with these kind of constraints, we could model them using an under-determined second order polynomial or maybe even a linear model, just as we do we the true function. If we want to use the under-determined model approach, we already have the needed Lagrange polynomial basis. Thus, including a constraint modelling approach into the algorithm demands almost no extra effort. However, the subproblem becomes significantly harder to solve, but the constraint representation becomes more accurate compared to just adding a penalty in a merit function[1]. If evaluating the black box is expensive, then this trade off may be reasonable.

[7] Available gradients means that we can calculate the derivatives of the constraints

3.5.2 Constraints in the well-placement challenge

The most basic constraints are the bounds on the decision variables. In the well-placement challenge, these bounds will allow us to specify the bounds of the oil reservoir. Moreover, if we would like to specify such that each well has a predefined, separate region, these kind of constraints would allow us to do so.

Basic restrictions on the decision variables are a very useful tool, however, it does have its limitation. One limitation is that, in the case that we have overlapping regions, there is nothing keeping the wells distanced from each other and, of course, we would not like to place two wells in the same place. Another limitation is that, even though we have separated regions, we might want a minimum distance between two wells which is bigger than the distance between the regions. One solution is to shrink the regions, however, for obvious reasons, this is not desired.

Linear constraints will, for example, allow us to set a restriction on how fast the set-point for the bottom (or top) hole pressure may change.

Restrictions on the length of the wells are common to include into the optimization procedure. This makes sure that the wells are not too short nor too long. A too short well might be not be possible to actually drill, whereas a too long one might be too expensive or the drilling capabilities might be restricted. Let's say that we would like to specify the minimum length of a well. A well is defined as a spline (i.e., two points and the line between the points). A minimum distance constraint (e.g., $||\mathbf{x} - \mathbf{y}||_2 > D_{\text{min}}$) is in general a hard nonconvex nonlinear constraint. Directly including such a constraint is not preferable. One approach is to convexify the constraint, which is done in [24]. The result is given below.

Say that we have N different vectors which we want to separate by a minimum distance, D_{min}. Let $\mathbf{a}_{ij}$ be a column vector of the same length as the vectors that we want to keep separated. These vectors must satisfy $||\mathbf{a}_{ij}||_2 = 1$ for $i < j$, and $j = 1, 2, \ldots, N$.

$$\mathbf{a}_{ij}^{\mathsf{T}}(\mathbf{x}_i - \mathbf{x}_j) \geq D_{\text{min}}, \qquad i < j, \quad j = 1, 2, \ldots, N$$

There are many good heuristics to choose the $\mathbf{a}_{ij}$'s [24]. One approach is to start with an approximate solution of the optimization problem which doesn't need to satisfy the minimum norm constraint. In other words, one can ignore the constraint when finding the approximate solution. Denote the approximate solution $\hat{\mathbf{x}} = [\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \ldots, \hat{\mathbf{x}}_N]$. We then choose $\mathbf{a}_{ij} = (\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_j)/||\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_j||_2$. [24]

The given reformulation will make the constraint convex, but it will also decrease the size of the feasible area. This means that solving this more restricted problem is easier, and any solution found will be guaranteed to be feasible for the nonconvex problem. An example of this trick is given in Figure 3.2. The $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$ represents an approximate solution of the optimization problem of interest, but where the minimum distance constraint has been ignored. The required minimum distance between the two points is $D_{\text{min}} = 2$.

To illustrate the difference between the nonconvex and the convex constraint, we have assumed that we keep $\hat{\mathbf{x}}_1$ and only move $\hat{\mathbf{x}}_2$. The circle represents the nonlinear constraint, and all the area outside the circle are the feasible area. For the convex constraint, we can clearly see that the feasible area is severly decreased. The only area that is still feasible is the area right of the green line.

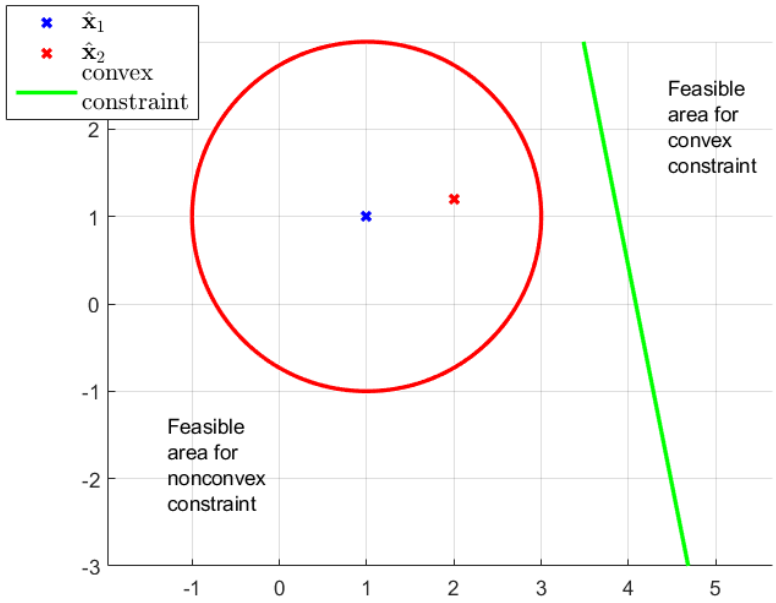


Figure 3.2: An illustration of the difference between the nonconvex and the convex minimum distance constraint. We have assumed that we keep $\hat{\mathbf{x}}_1$ and only move $\hat{\mathbf{x}}_2$.

This procedure can clearly be applied to keep the lengths of the wells above a minimum value. As mentioned earlier, we would also like to restrict the maximum length of a well (e.g., $||\mathbf{x} - \mathbf{y}||_2 \leq D_{\text{max}}$). This is just the Euclidean norm which forms a closed, convex feasible area because we have to stay inside the circle. However, if the optimization algorithm doesn't allow using nonlinear constraints, we may perform a linearization of it, as is done in [3]. The result of the linearization is shown in Figure 3.3. However, the linearization becomes more tedious when we add another dimension. The outline of the method is to uniformly sample the n dimensional sphere and create linear constraints. Moreover, we already have a constraint of the exact same type ($||\mathbf{x}-\mathbf{y}||_2 \leq D_{\text{max}}$), namely, the trust-region constraint. Thus, we will keep this convex constraint for now.

There is one type of constraints left that we haven't yet discussed, namely, the minimum inter-well distance constraint. This might look a lot like the minimum length restriction, however, it is far more complex. If the minimum distance restriction is applied without thinking, it would have to be applied an exponential amount of times. Let's say

that there are only two wells, then we must assert that the length between every point on each of the wells are separated by the minimum amount. Considering that the variables are continuous, this gets even worse.

Another option is to first find the two points that are closest, and then apply the constraint for those two points. However, this kind of setup leads to a more complicated problem to solve. It would be of interest to try to include that directly into the subproblem and see how it goes, but we should have another possible option for including this type of constraints. The inter-well distance constraint can be thought of as a soft constraint. Hence, it doesn't matter if it is violated by "some" amount. This means that we can avoid dealing with them directly while solving the subproblem. Instead a penalty term can be added to the objective function and a merit function can be used as the true function in the optimization algorithm. This, alone, will make the true function nonlinear, however, considering we are already dealing with an oil reservoir simulator, the objective function is highly nonlinear already. To be more precise, we will find the minimum distance between the wells and then use this value as an input to a penalty function such that the penalty will be high if the constraint is violated too much. This term will be added to the output of the simulator.

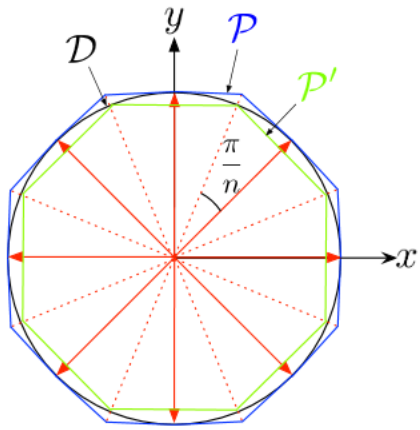


Figure 3.3: Linearization of the Euclidean norm in 2D. The green polygon is the one of interest, as it only contains feasible points. The image is taken from [3]

.

This concludes the constraint handling process. Now we have methods to include all the different types of constraints of interest: minimum length, maximum length, minimum inter-well distance and regions for the different wells. In other words, we can specify bounds, linear constraints and nonlinear constraints.

3.6 Robustness against noise

As mentioned in the introductory chapter, derivative-free methods have a tendency to find a "good" local minimum. The multiple minimums can either arise due to noise or the function of interest might simply have several minimums. The origin of the multiple minimums is not important. In this section, an illustration will be given of why derivative-free methods possess this property.

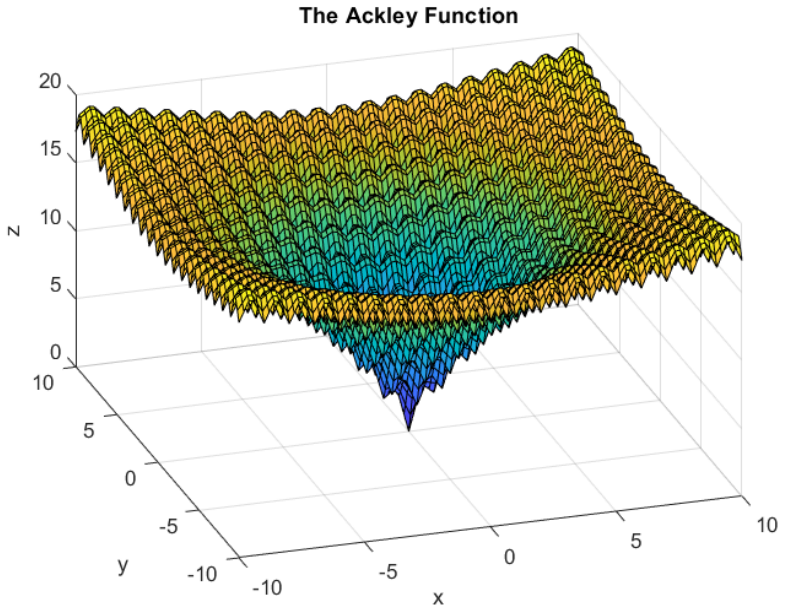


Figure 3.4: The Ackley function with multiple local optimums.

We will be using the Ackley function to demonstrate the robustness against noise. This function was proposed by David Ackley[25] and is commonly used for testing of optimization algorithms.

$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{n}\mathbf{x}^\intercal\mathbf{x}}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos cx_i\right) + a + \exp(1) \tag{3.37}$$

With the values $a = 20$, $b = 0.2$, $c = 2\pi$ and $n = 2$.

The equation is given in (3.37), whereas the plot is shown in Figure 3.4. We see that it has plenty of local optimums. For the illustration, we will simply choose a center point, a trust-region radius, sample the function and create the interpolation model. In Figure 3.5 we clearly see the how the algorithm is "blind" to everything that is in between the points. Despite the fact that there are lots and lots of local optima in between the sample points, the optimization procedure will be able to find a decent solution. The center point of the model has been set to be at the global optimum and the trust-region radius is set to

5 whereas the number of interpolation points is $m = 2n + 1 = 5$. This figure gives us a clue about the initial value for the trust-region radius. If the initial value is too small, we will easily be trapped in a local optimum. Thus, in the case that noise is present, it is very important that the initial value is set accordingly. Further, if possible, it should be chosen such that there is a good chance that the global optimum is within the initial trust-region.

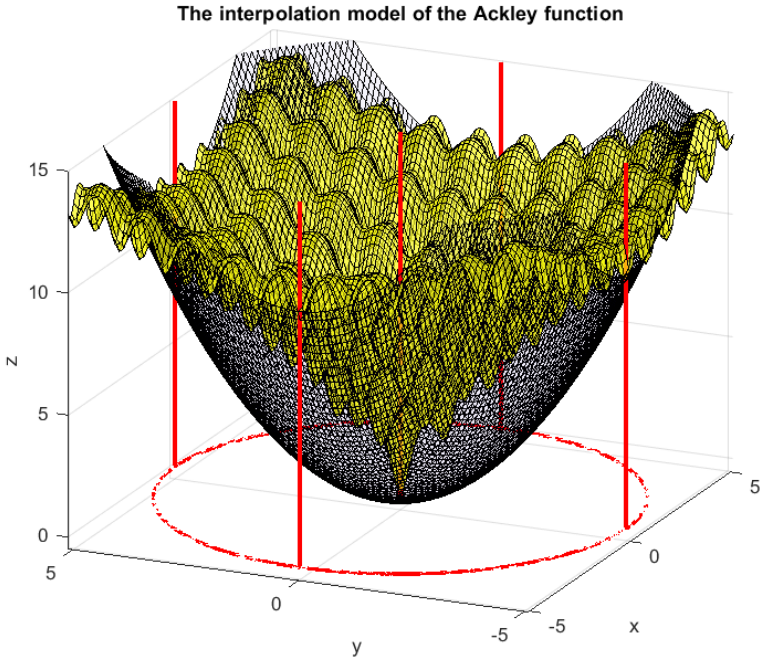


Figure 3.5: The Ackley function plotted together with the interpolation model. The circle represents the boundary of the trust-region. The red lines are the sample points. Center point is $[0, 0]$ and the trust-region radius is 5. $m = 2n + 1 = 5$.

The effectiveness of this near blindness might have been a bit over-represented in the figure because the global optimum is at the center point of the model. However, the property is still present even if the model is centered elsewhere. This is shown in Figure 3.6 where the center point is placed at $[2.1, 3]$. The global optimum is located at $[0, 0]$, and the solution of the subproblem would have been $x = 0.2892$ and $y = -1.0581$. In other words, the algorithm are converging against a good local (possible global) optimum.

3.6.1 Scenario: Including gradients

The reason why gradients are included into the model-making process in the first place is to either allow the number of required sample points (i.e., m) to be lowered, or keep the same amount and use the gradient information to improve the model. In this section, a

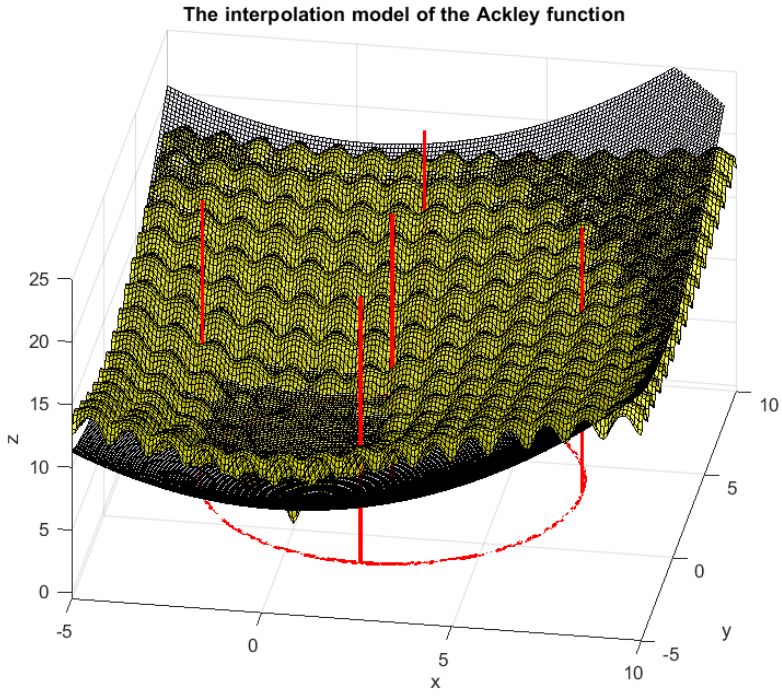
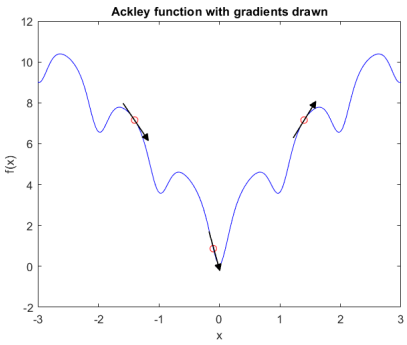


Figure 3.6: The Ackley function plotted together with the interpolation model. The circle represents the boundary of the trust-region. The red lines are the sample points. Center point is $[2.1, 3]$ and the trust-region radius is 5. $m = 2n + 1 = 5$

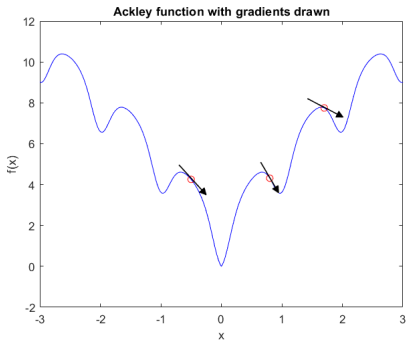
strong limitation on the objective function will be derived. To keep the illustrations simple, the Ackley function (3.37) in 1D will be used. In addition, we will cheat a bit and use $m = (n+1)(n+2)/2 = 3$ sample points. In theory, this would define the quadratic model (i.e., the constant, the gradient and the 1-by-1 Hessian matrix) and we would not need to include any gradients. However, this doesn't influence the purpose of the illustration.

If we are lucky with the sample points, including gradient information will make the model a lot better. Figure 3.7a shows this scenario. However, if we are not that lucky with our sample points, including the gradients will make the model a lot worse than if the gradients were neglected altogether. Figure 3.7b shows how two out of three of the gradients point in completely wrong directions and we do definitely not want to "enhance" the model with this information. Unless the modelling process should be based on luck, the gradients shouldn't be included. If the true function is a convex quadratic function, the gradients can be included. However, if this is the case, then a regular gradient-based approach can be used. Nonetheless, results of testing this idea will be shown in Chapter 4.

The reason why the derivative-free methods have a tendency to find good local opti-



(a) Here good sample points have been chosen by luck.



(b) Here bad sample points have been chosen by bad luck.

Figure 3.7: The red circles are sample points. The direction of the gradients are drawn, the size of the arrow is not representative. The figures illustrates why including gradient information might be a very bad idea.

mums is the fact that they only use the objective function value and not the gradients. Thus, including the gradients into the model-making process defeats its own purpose. Instead of enhancing the model, it will make it worse (unless we are lucky). The only achievement that is still obtained is that the model can be made with very few points. However, there exists better methods. E.g., use a linear model or use the under-determined model we already have discussed in-depth detail.

3.7 Poisedness - Geometry of the interpolation points

Until now, the phrase "geometry of the interpolation points" has been used, and it has been implied that the geometry can be both good and bad. A term which is used to describe the geometry characteristics is *poisedness*, which is used in [1]. Poisedness is a measurement of how well the points are spread. In other words, how likely it is that the points will be able to capture information. E.g., if $n = 2$ and the m points are all on the first axis, the geometry will be very bad. However, if one point is at the origin, and the other ones are $[1, 0], [0, 1], [-1, 0]$ and $[0, -1]$ the geometry will be good, or we can say the points are *poised*. Actually, the geometry will only be good if the area that are being considered is close to the points (say, a ball centered at the origin with radius 1). If we have a radius of 10,000,000 these points are no longer suitable.

In our work, the Lagrange polynomials have been chosen as the basis for the model. It is a natural choice because the Lagrange polynomials are readily available when Powell's updating method is used, but, as mentioned, we could have chosen another one if that was desired. However, this basis is convenient because it has several good properties. Some of these properties are kept even though we use a modified version of the classical Lagrangian basis. Remember that our basis is not uniquely defined by the Kronecker delta

property alone, whereas the classical Lagrangian basis would have been. Firstly, this basis gives a convenient way to measure the *poisedness* of the interpolation set and is also the most commonly used measurement of it [1]. Finally, the basis scales automatically, i.e., all the coefficients are always in the set $[-1, 1]$, which means that we do not need to add any additional normalization procedures to keep it numerically sound.

The poisedness of the interpolation set is denoted Λ and is defined as[1]:

$$\Lambda = \max_{1 \leq j \leq m} \max_{\mathbf{x} \in B} |\ell_j(\mathbf{x})| \tag{3.38}$$

where B is the region of interest. This region is often defined by a point (e.g., the best point found thus far in the overall optimization algorithm) and a radius (e.g., the trust-region radius). If an interpolation set is well poised in a ball B with radius Δ, then it will also be poised in any smaller ball centered at the same place[1]. This makes sense as the poisedness is defined as the maximum absolute value of all the Lagrange polynomials that can be found within the area of interest. If the area that is being searched for a maximum value is decrease, the output will be either the same or lower. Which again means that if the poisedness was lower than "Λ_{max}" before, then it will also be lower in the new smaller search area.

The poisedness of the set is completely independent of the true function we are trying to minimize. The formula for calculating the classical Lagrange polynomials in 1D is:

$$\ell_t(\mathbf{x}) = \prod_{\substack{j=1 \\ j \neq t}}^{m} \frac{\mathbf{x} - \mathbf{y}_j}{\mathbf{y}_t - \mathbf{y}_j}$$

which clearly doesn't care about the true function. For the under-determined case, this still remains true as we can see from (3.12) or (3.20). Remember that $\mathbf{W}$ and its inverse, $\mathbf{H}$, is solely determined by the interpolation points.

In the next section a procedure to improve the poisedness is given.

3.8 Model improvement algorithm

In this section an algorithm to improve the poisedness is presented and commented. The poisedness is improved by "moving the points around", while the search area remains unchanged, such that the value of (3.38) is lowered[8]. This algorithm is needed in the final algorithm to keep the surrogate model useful throughout the optimization procedure. The *model improvement algorithm*, which is a key component of the overall algorithm, is given in Algorithm 1. The algorithm is commented and then it is applied to a set of points to illustrate how it spreads the points.

[8]Remember that if one point is replaced with a new one, all the Lagrange polynomials must be updated.

k=1;
while *poisedness not good enough* **do**

$$\Lambda_k = \max_{1 \leq j \leq m} \max_{\mathbf{x} \in B} |\ell_j(\mathbf{x})| \tag{3.39}$$

if $\Lambda_k > \Lambda$ **then**
let $t \in \{1, 2, \ldots, m\}$ be an index of which

$$\max_{\mathbf{x} \in B} |\ell_t(\mathbf{x})| > \Lambda \tag{3.40}$$

and let $\mathbf{x}^+ \in B$ be a point that maximizes $|\ell_t(\mathbf{x})|$ in B.
Replace the one interpolation point
$\mathbf{y}_t := \mathbf{x}^+$
else
The interpolation set is Λ-poised. Exit.
end
Update all the Lagrange polynomials.
k += 1.
end

Algorithm 1: Model improvement algorithm.

Finding Λ_k in (3.39) involves finding the minimum of m quadratic functions within area B. This can therefore be a very time consuming task. Fortunately, an approximation of the upper and lower limits of the $|l_j(\mathbf{x})|$ can be used instead. Finding an upper limit of each polynomial is rather straightforward. Use (3.20d), but replace all the coefficients with their corresponding absolute value and then use $\mathbf{x} = [\Delta, \Delta, \ldots, \Delta]^\intercal$ as an input. A lower limit is more tedious to find. It is shown that if at least one of the coefficients in the polynomial has an absolute value of at least b, the ball is centered at the origin, and the radius is 1, then the lower limit is $b/4$ [1]. The case for when the radius is something else and/or the ball is centered elsewhere, is not shown. An idea to solve this issue is to simply scale and shift the interpolation points such that they are all contained in a ball of radius 1 centered at the origin with at least one point on the boundary. The poisedness of the shifted and scaled set will be the same as for the original one[1]. Of course, this means that we must calculate the Lagrange polynomials from scratch. The fact that the poisedness remains the same makes sense as we are measuring how the points lies relative to each other within an area, thus scaling both the points and the area should not impact this measurement, nor should a shifting of the points and area.

If one chooses to use an approximate approach in (3.39), then one must be sure that either $\Lambda_k > \Lambda$ or $\Lambda_k \leq \Lambda$. In other words, the estimated lower limit must be guaranteed to be higher than Λ or the estimated upper limit must be guaranteed to be lower or equal Λ. Λ is a predetermined constant (chosen by the user) which represents the required poisedness. The point $\mathbf{x}^+$ can also be approximated instead of solved explicitly. After this is done, an updating of the Lagrange polynomials can be effectively done as previously described. If the limits ($\Lambda_k > \Lambda$ or $\Lambda_k \leq \Lambda$) can not be guaranteed by the upper and lower bounds, then the algorithm should switch to an optimization algorithm to make sure that the limits are satisfied.

Finding the maximum of $|\ell_j(\mathbf{x})|$ within the ball can also be done by solving two maximization problems, one for $\ell_j(\mathbf{x})$ and another for $-\ell_j(\mathbf{x})$. The highest value of those two will be taken. Considering that this type of optimization problem is the exact same one as the one in the subproblem, the same solver can be applied. The Λ in the algorithm is a predetermined constant. When the poisedness of the interpolation set is less or equal to Λ, then the set is called Λ-*poised*.

The algorithm will finish in a finite, uniformly bounded amount of iteration[1]. However, the algorithm is not guaranteed to always improve the poisedness from one iteration to the next. Sometimes, it will get worse, and next time it might keep on improving.

Before we go to the example, two remarks will be given. For the first remark, let's look at equation (3.40), while simultaneously remember the one criterion Powell had on the new interpolation point such that (P1) and (P2) would be satisfied. Not only is $|\ell_t(\mathbf{x}^+)|$ nonzero, but it is in addition also bigger than 1 (Λ must be chosen greater than one). Hence, Powell's criterion is always fulfilled when Algorithm 1 is applied to increase the poisedness.

The second remark is that, as previously mentioned, the poisedness (and thus also the algorithm to improve the poisedness) is completely indifferent to the true function. This is an important feature, because we can calculate all the new points, and then evaluate them. This enables the possibility of using a parallel simulation scheme and it can be a good time saver.

Example of the Model-improvement algorithm

To give the reader a better feeling of what poisedness is, an example is given. Here, the model improvement algorithm is applied to a set of points which is not Λ-poised to begin with. The initial set is the $\mathbf{Y}_1$ below. In the example $n = 2$ and $m = 6$. We have done 5 iterations of the algorithm. The change in the set from one iteration to the next is marked in red both in the matrices below and in the Figure 3.8. The sets of points are also plotted in Figure 3.8 for easier visualization. The poisedness of each set if written below each plot. As we can see from the figure, the poisedness value decreases a lot in the beginning, but once the points are no longer on (or almost on) top of each other, the poisedness remains almost unchanged. The region where the poisedness is measured is a ball of radius 1 centered at the origin.

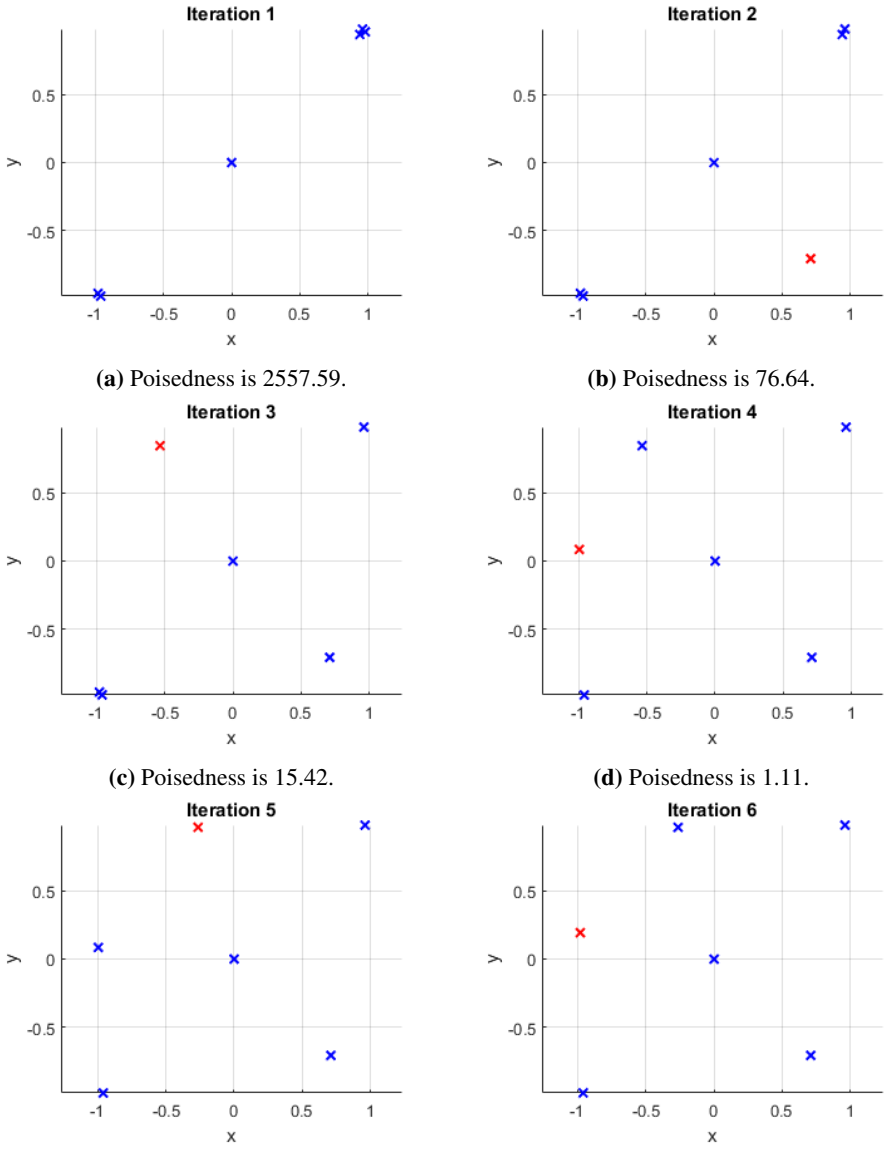
$$\mathbf{Y}_1^\intercal = \begin{bmatrix} -0.98000000 & -0.96000000 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.98000000 & 0.96000000 \\ 0.96000000 & 0.98000000 \\ 0.94000000 & 0.94000000 \end{bmatrix}, \quad \mathbf{Y}_2^\intercal = \begin{bmatrix} -0.98000000 & -0.96000000 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.70891722 & -0.70529170 \\ 0.96000000 & 0.98000000 \\ 0.94000000 & 0.94000000 \end{bmatrix}$$

$$\mathbf{Y}_3^\intercal = \begin{bmatrix} -0.98000000 & -0.96000000 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.70891722 & -0.70529170 \\ 0.96000000 & 0.98000000 \\ -0.53544173 & 0.84457217 \end{bmatrix}, \quad \mathbf{Y}_4^\intercal = \begin{bmatrix} -0.99641143 & 0.08464202 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.70891722 & -0.70529170 \\ 0.96000000 & 0.98000000 \\ -0.53544173 & 0.84457217 \end{bmatrix}$$

$$\mathbf{Y}_5^\intercal = \begin{bmatrix} -0.99641143 & 0.08464202 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.70891722 & -0.70529170 \\ 0.96000000 & 0.98000000 \\ -0.26438316 & 0.96441772 \end{bmatrix}, \quad \mathbf{Y}_6^\intercal = \begin{bmatrix} -0.98115169 & 0.19323911 \\ -0.96000000 & -0.98000000 \\ 0.00000000 & 0.00000000 \\ 0.70891722 & -0.70529170 \\ 0.96000000 & 0.98000000 \\ -0.26438316 & 0.96441772 \end{bmatrix}$$

3.9 Solving the subproblem

In this section, methods to solve the subproblem will be given. We start by considering solvers for the unconstrained optimization problem. This is the most common situation in the derivative-free trust-region model-based setting. Afterwards, the case when constraints



(a) Poisedness is 2557.59.

(b) Poisedness is 76.64.

(c) Poisedness is 15.42.

(d) Poisedness is 1.11.

(e) Poisedness is 1.02.

(f) Poisedness is 1.00.

Figure 3.8: The figures show how the poisedness is improved by using Algorithm 1. The red crosses are the points that have changed from one iteration to the next. "Iteration 1" is the initial set of points, i.e., $\mathbf{Y}_1$. "Iteration 2" is the points in $\mathbf{Y}_2$, and so on.

are present will be discussed.

As mentioned in the literature study, there are several ways of finding the solution of the subproblem in the classical unconstrained case. The subproblem is the task to find the minimum of the quadratic model within the trust-region:

$$\begin{array}{ll} \min\limits_{\mathbf{s}} & c + \mathbf{g}^\intercal \mathbf{s} + \mathbf{s}^\intercal \mathbf{G} \mathbf{s} \\ \text{s.t.} & ||\mathbf{s}||_2 \leq r \end{array} \tag{3.41}$$

where the c could be removed because it does not affect what the optimal solution vector is, it only influences the value of the objective function at that point. $\mathbf{s}$ is the distance from the current iterate, and the constraint makes sure that the distance from the current iterate to the next is no more than the trust-region radius. The form of the exact solution is presented before some approximation alternatives are shown.

3.9.1 The exact solution

The exact solution is on the form [12]:

Corollary 1. *Any global minimizer of* (3.41) *satisfies the equation*

$$\begin{aligned} (\mathbf{G} + \mathbf{I}\lambda^*)\mathbf{s}^* &= -g \\ (\mathbf{G} + \mathbf{I}\lambda^*) &\succeq 0 \\ \lambda^* &\geq 0 \\ \lambda^*(||\mathbf{s}^*||_2 - r) &= 0 \end{aligned}$$

If $(\mathbf{G} + \mathbf{I}\lambda^*)$ *is positive definite, then* s^* *is unique.*

The characterization of the solution was first obtained in [26]. An algorithm to find the (almost) exact solution is presented in great details in [12]. However, the algorithm is potentially computationally heavy as it relies on the Cholesky factorization of $(\mathbf{G}+\mathbf{I}\lambda^*)$ in an iterative process. Considering that the surrogate model is an under-determined model, in addition to the fact that the true function is not necessarily a second order polynomial, there is probably no point in using lots of resources to solve the approximation as exactly as possible. In other words, if the subproblem is solved exactly it does not necessarily give a better[9] solution than an approximate solver would have given. This is because the minimum of the surrogate model and the minimum of the true function do not necessarily coincide. The key idea behind the under-determined model is that it should capture the main curvature of the true function to help the optimization algorithm to move towards the (local) optimum.

3.9.2 Approximate solutions

Different algorithms for finding an approximate solution are given below. These algorithms are all based upon the conjugate gradient method, which is also presented. First we

[9] by "better" we mean a solution that will make the overall algorithm converge faster.

present the Cauchy and Eigenstep concepts because these are essential to the analysis of global convergence, and they are the most basic methods that can be applied. The following presentation of these concepts is based on the discussion given in [1].

The steepest descent direction (i.e., negative gradient direction) can be thought of as the driving force behind all optimization techniques. Global convergence (i.e., the algorithm ending in a finite number of iterations) requires that the model is minimized at least as good as something related to the steepest descent. This is where the Cauchy point enters, which is the step to the minimum of the model along the steepest descent direction within the trust region. In the case of negative curvature (i.e., an indefinite or negative definite second order derivative matrix) and the requirement to have global convergence to second-order critical points, then another step must also be considered, namely the Eigenstep. The Eigenstep is a step related to the most negative curvature. They will yield a global convergent algorithm, but that doesn't say anything about the rate of convergence. In [12] they compare an algorithm to the Cauchy point, and this is how they phrase it:

"*The resulting step is then barely, if at all, better than the Cauchy direction, and this leads to a slow but globally convergent algorithm in theory and a barely convergent method in practice.*"

In other words, looking for a better approach is a good idea.

The Conjugate Gradient method

The reason why the steepest descent method doesn't provide an optimal trajectory is that a later step will often undo some of the progress towards the optimum that was done in a previously taken step. This happens because only two consecutive steps are required to be orthogonal to each other, thus each step can undo some of the progress made by the steps before the last one. A solution to this is to make all steps conjugate to each other. This is what is defined as the conjugate gradient method(see Algorithm 2). A new basis for the search space, $\mathcal{R}^n$, is defined, and then steps along these vectors are taken. Actually, it is done in an iterative process, i.e., find a direction, take a step, and repeat. Two vectors, $\mathbf{u}$ and $\mathbf{v}$ are conjugate with respect to $\mathbf{G}$ if $\mathbf{u}^\intercal\mathbf{G}\mathbf{v} = 0$. This is also denoted $\mathbf{G}$-orthogonal or $\mathbf{G}$-conjugate[18]. Considering that the algorithm goes along vectors which span $\mathcal{R}^n$, then the optimum must be found within at most n steps. Unfortunately, this method requires the second derivative matrix to be positive definite. There are five reasons why this method is of interest[12]:

- The best general-purpose algorithm to solve (3.41) in an iterative way.
- Easy to understand and easy to implement. See Algorithm 2.
- Progress is made at each iteration, and global convergence can be concluded.
- If terminating early (i.e., before all vectors in the basis are used), a decent point can still be found.
- Can be modified to also work for the cases where the Hessian is not positive definite.

Given $\mathbf{x}_0$;
$\mathbf{g}_0 = \mathbf{G}\mathbf{x}_0 + \mathbf{g}$;
$\mathbf{p}_0 = -\mathbf{g}_0$;
for $k = 0, 1, 2, \ldots$ until convergence **do**
  $\alpha_k = ||\mathbf{g}_k||_2^2 / (\mathbf{p}_k^\intercal \mathbf{G}\mathbf{p}_k)$
  $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
  $\mathbf{g}_{k+1} = \mathbf{g}_k + \alpha_k \mathbf{G}\mathbf{p}_k$
  $\beta_k = ||\mathbf{g}_{k+1}||_2^2 / ||\mathbf{g}_k||_2^2$
  $\mathbf{p}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{p}_k$
end

Algorithm 2: The conjugate gradient method [12]

Truncated Conjugate Gradient method

This method is the same as the conjugate gradient method except that it looks for the solution within a ball (e.g., the trust-region), and the step is thus *truncated* if the method tries to go to a point which is outside of the ball. There are two additional exits. The first is the case when the model is convex, but the solution lies outside the trust-region. In this case, the selected point is where the line from the current point to the optimal point crosses the trust-region. The second case is when the model is nonconvex, also here the point lies on the boundary. The original method [16] is a preconditioned version, a version without the preconditioning is presented in [17], and is the one that is given below in Algorithm 3.

3.9.3 The constrained case

If there are constraints the subproblem becomes more difficult and other methods must be applied to solve it. Unless there is only bounds on the decision variables, the previously discussed methods can not be used. If there are only bounds, we can treat the bounds as the same way as the methods treat the trust-region radius. However, we are interested in more general constraints, such as linear and nonlinear inequality constraints. There are different options to choose between, such as interior point methods, genetic algorithms and sequential quadratic programming methods. The latter is chosen, mainly because we had access to a solid implementation of it which has been around for years and are used by professionals, namely, SNOPT[27].

Step 1
Given $\mathbf{g}$, and $\mathbf{G}$;
$\mathbf{x}_1 = 0$;
$\mathbf{g}_1 = \mathbf{g}$;
$\mathbf{d}_1 = -\mathbf{g}$;
$k = 1$;

Step 2
if $||\mathbf{g}_k|| = 0$ **then**
$\mathbf{x}^* = \mathbf{x}_k$;
Stop;
end
Compute $\mathbf{d}_k^\intercal \mathbf{G} \mathbf{d}_k$;
if $\mathbf{d}_k^\intercal \mathbf{G} \mathbf{d}_k \leq 0$ **then**
Go to step 4;
end
$\alpha_k = -\mathbf{g}_k^\intercal \mathbf{d}_k / (\mathbf{d}_k^\intercal \mathbf{G} \mathbf{d}_k)$;

Step 3
if $||\mathbf{x}_k + \alpha_k \mathbf{d}_k|| \geq r$ **then**
Go to step 4;
end
$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$;
$\mathbf{g}_{k+1} = \mathbf{g}_k + \alpha_k \mathbf{G} \mathbf{d}_k$;
$\beta_k = ||\mathbf{g}_{k+1}||^2 / ||\mathbf{g}_k||^2$;
$\mathbf{d}_{k+1} = -\mathbf{g}_k + \beta_k \mathbf{d}_k$;
$k := k + 1$;
Go to step 2;

Step 4
Compute $\alpha_k^* \geq 0$ such that $||\mathbf{x}_k + \alpha_k^* \mathbf{d}_k|| = r$,
i.e.. by taking the positive root:
$$\alpha_k^* = \frac{-2\mathbf{x}_k^\intercal \mathbf{d}_k + \sqrt{4(\mathbf{x}_k^\intercal \mathbf{d}_k)^2 - 4\mathbf{d}_k^\intercal \mathbf{d}_k (\mathbf{x}_k^\intercal \mathbf{x}_k - r^2)}}{2\mathbf{d}_k^\intercal \mathbf{d}_k};$$
$\mathbf{x}^* = \mathbf{x}_k + \alpha_k^* \mathbf{d}_k$;
Stop;

Algorithm 3: The truncated conjugate gradient method [17]

Sequential Quadratic Programming (SQP)

SNOPT, which is an implementation of a SQP algorithm, can solve problems on the form in (3.42).

$$\begin{aligned} &\min_{\mathbf{x}\in\mathcal{R}^n} \quad f(\mathbf{x}) \\ &\text{s.t.} \\ &\qquad \mathbf{l} \leq \begin{bmatrix} \mathbf{c}(\mathbf{x}) \\ \mathbf{A}\mathbf{x} \\ \mathbf{x} \end{bmatrix} \leq \mathbf{u} \end{aligned} \tag{3.42}$$

where f is the objective function, $\mathbf{A}\mathbf{x}$ is a set of linear equations and $\mathbf{c}(\mathbf{x})$ is a set of nonlinear constraints. $\mathbf{l}$ and $\mathbf{u}$ are the lower and upper bounds, respectively, for the constraints as well as the bounds for the decision variables. Only a brief outline of the method will be given here. The interested reader are referred to [28] and [15]. The main idea is in the name of the method, namely, to solve the nonlinear constrained problem (3.42) by solving a sequence of quadratic programs. The constraints of the subproblems are linearization of the constraints in (3.42). The objective function of the subproblems are a quadratic approximation to the Lagrangian function. The second derivatives are assumed to be to expensive to calculate, thus the Hessian is approximated in an iterative way. [28]

The observant reader might have seen a connection between the constraint handling in the SQP algorithm and how they could be included into the subproblem of the derivative-free trust-region model-based algorithm. The SQP algorithm makes a linearization, or one could say a linear model, of the nonlinear constraints. This is the exact same idea that was suggested to handle nonlinear constraints in the derivative-free method if we choose to model them as linear models and not under-determined second order polynomials.

This concludes the discussion on how to solve the subproblem. The next section presents a technique to make the derivative-free model-based trust-region algorithm more practical.

3.10 The scaling factor, r

In many of the situations where derivative-free model-based trust-region methods are applied, the function evaluations are expensive, and thus it might be a good idea to keep old interpolation points even though they are outside the trust-region Δ_k. It turns out that if we only work with one trust-region radius, the algorithm will be be rather impractical. This is because too many points must be replaced to keep them inside of the trust-region. This problem can be circumvented by introducing a second trust-region defined as an up-scaled version of the other one[1]. It is suggested that the value of the scaling factor is $r \geq 2$. The exact details of where the scaled region should be used and where the normal region should be used are not given. Thus, a suggestion is given here.

The normal trust-region should be used whenever a new point is selected. This goes for both the case when we are solving the subproblem or when we trying to improve the

geometry of the set of interpolation points. The scaled trust-region should be used when we are measuring poisedness and when we are checking if a point is outside the region or not.

In the next section a couple of terms that are frequently used in [1] will be presented.

3.11 Certifiably fully linear models

When talking about surrogate models, [1] introduces the terms fully linear (FL) and certifiably fully linear (CFL). These terms are used in the process of proving global convergence of the algorithms developed. Only a short description of the terms will be given here. For a more in-depth and mathematically description, the interested reader is referred to the book.

If there exists two fixed bounds, one on the difference between the true function and the model, and the other on the difference between the gradients of the same functions, within a region (e.g., the trust-region), then the model is said to be fully linear. For FL models, there exists a "Model-improvement" algorithm, that in a finite, uniformly bounded number of iterations can either conclude that the model is FL (in which case we refer to the model as CFL), or the algorithm will produce a new model which is FL within the trust-region. There is, fortunately, a convenient relationship between (C)FL and Λ-poisedness. If a model is based on a Λ-poised set in the region of interest (trust-region), then the model is fully linear. Thus, if we inspect the poisedness of the interpolation set, and the set is Λ-poised, then the model is CFL.

Algorithm 1 is such a Model-improvement algorithm. It satisfies all the criteria above. First, we note that Algorithm 1 will terminate in a finite, uniformly bounded number of iterations, as is mentioned in Section 3.8. When the poisedness is measured the first time (i.e., when $k = 1$ and Λ_1 is found) the algorithm either conclude that the set of interpolation points is Λ-poised, which means it concludes that the model based upon that interpolation set is CFL. In the other case when the algorithm concludes that the set is not Λ-poised, then it will start iterate and produce a Λ-poised set. The new model based upon the new set will be CFL.

The reason why we are talking about fully *linear* and not fully *quadratic* is because the interpolation model that are used in this project is an under-determined quadratic model and, thus, it cannot be fully quadratic.

3.12 The algorithm

As mentioned in the literature review, Algorithm 11.2 in "Introduction to Derivative-Free optimization"[1] is the algorithm that is chosen for this specialization project. Powell's method to build and update an under-determined quadratic model is embedded into the framework of the book.

All the needed building blocks to use the framework have been described: building, updating and maintaining the model and the interpolation set and how to solve the subproblem. Here we present the full derivative-free model-based trust-region optimization algorithm.

3.12.1 The derivative-free model-based trust-region algorithm

The first step of the derivative-free model-based trust-region algorithm is the initialization step, which is self-explanatory. Step 1 makes sure the surrogate model is a (C)FL model before we proceed by finding a solution of the subproblem in step 2. In step 3 we decide if the newly found point should be included into the interpolation set and if the surrogate model does mimic the true function satisfactory. Step 4 is only entered if the newly found point didn't become a new "best found" point. It tries to improve the model by replacing one of the interpolation points by a point that will improve the geometry of the set. In the last step, the trust region radius is updated and the model is updated based upon the previously done changes. Below the algorithm, further explanations are given.

Step 0 - Initialization

Choose $m \in [n+2, (n+1)(n+2)/2]$. $m = 2n+1$ is recommended.
Choose an initial point $\mathbf{y}_0$.
Choose a maximum radius Δ_{max}.
Choose an initial trust-region radius, $\Delta_0 \in (0, \Delta_{max}]$.
Choose the trust-region radius factor $r \geq 1$.
Compute the first set of interpolation points. Store the set of interpolation points in $\mathbf{Y}_0$.
Compute the initial minimum Frobenius norm Lagrange polynomials (i.e., find the initial $\mathbf{H}$ matrix and the initial quadratic model.).
Select a $\Lambda > 1$ to be used in the model improvement algorithm.
The constants $\eta_1, \gamma, \gamma_{icb}, \epsilon_c, \tau, \beta, \omega$, and μ must also be provided by the user,
and they must satisfy: $\eta_1 \in (0,1), 0 < \gamma < 1 < \gamma_{icb}, \epsilon_c > 0, 0 < \tau < 1, \omega \in (0,1)$ and $\mu > \beta > 0$.
Set $k = 0$.

Step 1 - Criticality step

If $||\nabla \mathcal{L}_k^{\text{icb}}||_2 > \epsilon_c$, then
    $Q_k = Q_k^{\text{icb}}$ and $\Delta_k = \Delta_k^{\text{icb}}$
    Done, go to next step.
Else,
    Check the poisedness of the interpolation set to attempt to certify if the model Q_k^{icb} is FL on $B(\mathbf{x}_k, r\Delta_k^{\text{icb}})$.
    If (the interpolation set is not Λ-poised) or ($\Delta_k^{\text{icb}} > \mu||\nabla \mathcal{L}_k^{\text{inc}}||_2$), then
        apply Algorithm 4 to construct a model $\tilde{m}_k(\mathbf{x}_k + \mathbf{s})$ with the gradient $\tilde{\mathbf{g}}_k$
        and the Hessian $\tilde{\mathbf{G}}_k$ at $\mathbf{s} = 0$, which is fully linear on the ball $B(\mathbf{x}_k, r\tilde{\Delta}_k)$),
        for some $\tilde{\Delta}_k \in (0, \mu||\nabla \tilde{\mathcal{L}}_k||]$ chosen by Algorithm 4.
        $Q_k = \tilde{Q}_k$ and $\Delta_k = \min\{\max\{\tilde{\Delta}_k, \beta||\nabla \tilde{\mathcal{L}}_k||_2\},\ \Delta_k^{\text{icb}}\}$

Else,
$Q_k = Q_k^{\text{icb}}$ and $\Delta_k = \Delta_k^{\text{icb}}$

Step 2 - Step calculation

Find the step, $\mathbf{s}_k$, towards the minimum of the model, e.g., using the truncated conjugate gradient method as in Algorithm 3 or a SQP algorithm.

Step 3 - Acceptance of the trial point

If $||\mathbf{s}_k||_2 \geq \tau \max\{||\mathbf{y}_j - \mathbf{x}_k|| : \mathbf{y}_j \in \mathbf{Y}_k\}$, then
compute $\mathbf{y}_t = \arg\max_j \{||\mathbf{x}_k - \mathbf{y}_j||_2 |\ell_j(\mathbf{x}_k + \mathbf{s}_k)| : \mathbf{y}_j \in \mathbf{Y}_k\}$.

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{s}_k)}{Q_k(\mathbf{x}_k) - Q_k(\mathbf{x}_k + \mathbf{s}_k)}$$

If ($\rho \geq \eta_1$) or ($\rho_k > 0$ and it is known that Q_k is CFL in $B(\mathbf{x}_k, r\Delta_k)$), then
$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$, and include $\mathbf{x}_k + \mathbf{s}_k$ into the set $\mathbf{Y}_{k+1}$ by replacing $\mathbf{y}_t$.
Else,
$\mathbf{x}_{k+1} = \mathbf{x}_k$.
If ($||\mathbf{y}_t - \mathbf{x}_k|| > r\Delta_k$) or ($|\ell_t(\mathbf{x}_k + \mathbf{s}_k)| > 1$), then
accept $\mathbf{x}_k + \mathbf{s}_k$ into the set $\mathbf{Y}_{k+1}$ by replacing $\mathbf{y}_t$.

Step 4 - Model improvement

If ($\mathbf{x}_{k+1} \neq \mathbf{x_k}$), then
go to Step 5.
Else,
Choose $\mathbf{y}_t = \arg\max_j \{||\mathbf{y}_j - \mathbf{x}_k|| : \mathbf{y}_j \in \mathbf{Y}_k\}$, and
find a new point $\mathbf{y}_t^* \in \arg\max\{|\ell_t(\mathbf{x})| : \mathbf{x} \in B(\mathbf{x}_k, \Delta_k)\}$.
If ($||\mathbf{y}_t - \mathbf{x}_k|| > r\Delta_k$) or ($|\ell_t(\mathbf{y}_t^*)| > \Lambda$), then
replace $\mathbf{y}_t$ by $\mathbf{y}_t^*$ in $\mathbf{Y}_{k+1}$.
Else, consider the next furthest point from $\mathbf{x}_k$ and repeat. If eventually a point $\mathbf{y}_t$ is found in $\mathbf{Y}_{k+1}$ such that $\max\{|\ell_t(\mathbf{x})| : \mathbf{x} \in B(\mathbf{x}_k, \Delta_k)\} > \Lambda$, then this point is replaced. If no such point is found, then there is no need to improve because $\mathbf{Y}_k$ is Λ-poised in $B(\mathbf{x}_k, r\Delta_k)$, which implies that Q_k is CFL in $B(\mathbf{x}_k, r\Delta_k)$.

Step 5 - Trust-region radius update

$$\Delta_{k+1}^{\text{icb}} = \begin{cases} [\Delta_k, \min\{\gamma_{\text{icb}}\Delta_k, \Delta_{\max}\}], & \text{if } \rho_k \geq \eta_1, \\ \gamma\Delta_k, & \text{if } \rho_k < \eta_1 \text{ and } Q_k \text{ is fully linear}, \\ \Delta_k, & \text{if } \rho_k < \eta_1 \text{ and } Q_k \text{ is not CFL} \end{cases}$$

Update the model Q_k to obtain Q_{k+1}^{icb}, and recompute the minimum Frobenius norm Lagrange polynomials.

$k := k + 1$.
Go to step 1.

3.12.2 Explanation of the algorithm

Some explanations of the different steps and logic are necessary. The following explanations are based, once again, upon the book [1].

Starting with step 0. The name of the step reveals what its main purpose is. How to find the first set of interpolation points and the initialization of the $\mathbf{H}$ matrix and the first quadratic model can be found in [10] and [11], if we have no constraints. However, if we have constraints, the selection of the interpolation points can be a lot more complex. However, for this project, we assume that the given initial point is feasible and that this point can be perturbed by the initial trust-region radius in each directions.

The $\mathbf{Y}_k$ is the interpolation set at iteration k. $\mathbf{x}_k$ is the best point found so far. Notice that the ball where we measure the poisedness about is centered at $\mathbf{x}_k$, while the model may be centered around another point! The superscript "icb" is short for incumbent, and is used to tell the difference between when something is partly updated and when it is actually fully updated. For example, Δ_k^{icb} may or may not be the final value for Δ_k. Further, Q_k denotes the quadratic model at iteration k.

The ρ_k (from Step 3), tells how good the model mimics the true function. Ideally ρ_k should be equal to 1, meaning that our model predicts the behavior of the true function perfectly, at the evaluated point $(\mathbf{x}_k + \mathbf{s}_k)$. However, demanding that ρ_k is equal to one is an unwise choice, and thus the η_1 is introduced. If $\rho_k \geq \eta_1$ then the model is "good enough". Further, the γ's are used in the updating of the trust region (in Step 5). In the case when $\rho_k \geq \eta_1$, the new trust-region radius can either be retained, or increased. In the other case, our model predicts too badly and thus the trust-region radius is reduced in order to try to make a better model, one can think of it as one zooms in and tries to get a better view of the function.

The Criticality step is perhaps the most non-intuitive one, particularly because of the naming of step 4 (Model improvement). The step named "Model improvement" does not use the model-improvement algorithm, whereas the Criticality step does. There might seem to be some redundant steps here. However, the Criticality step is necessary and it keeps the radius of the trust-region comparable to some measurement of stationarity so that when the measure of stationarity is close to zero, the model becomes more accurate. The measure of stationarity will be close to zero if the current iterate is close to a stationary point. The step also updates the trust-region radius, and it is this updating of the radius that forces it to converge to zero. The trust-region radius may, thus, be a natural stopping criterion for the algorithm. In addition, it is this step that makes sure the model is CFL or FL.

The $\nabla \mathcal{L}_k$ is the Lagrangian function of the subproblem at iteration k. The test $||\nabla \mathcal{L}_k^{\text{icb}}||_2 > \epsilon_c$ is clearly important in the Criticality step. The idea is that when $||\nabla \mathcal{L}_k^{\text{icb}}||_2 \leq \epsilon_c$, then

the step calculated will be a lot smaller than the trust-region radius (i.e., we are moving just a tiny bit inside the trust region). What happens next is that the model is being improved until some measurement of acceptance is reached. If the interpolation set is Λ-poised, but the criterion is still not met, then the trust-region is reduced and the model-improvement algorithm is applied again. This means that not only is a fully linear model required, but also some extra criterion on the relationship between the trust-region radius and the gradient of the Lagrangian of the subproblem must be satisfied. As we remember from the sections about poisedness and the model improvement algorithm, both of those are not concerned about the true function at all. This step will create a relationship and it is highly important regarding assuring global convergence. In the original algorithm in [1], the gradient of the model is used instead of the gradient of the Lagrangian. However, we have constraints. If we use the gradient of the model, we might end up with a high value of $||\mathbf{g}_k||_2$, but because of the constraints, we will not be able to achieve any decrease of the model while trying to solve the subproblem. Thus, we use the gradient of the Lagrangian to try to assure that there will be a decrease while solving the subproblem.

In Step 3 (Acceptance of the trial point) three things may happen. (i) The new point is accepted as the new iterate (i.e., it is better than the previously best found point), then $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$, and the point is also put into the set of interpolation points. If not, two alternatives are possible, (ii) the point is included into $\mathbf{Y}$ or (iii) the point is disregarded all together.

An interesting thing to notice is how the new interpolation points are chosen (if not produced by step 2), and also how the point that should be removed is selected. We see that in either case $|\ell_t(\mathbf{x}^+)|$ is nonzero, where t is the index of the old interpolation point and $(\mathbf{x}^+)$ is the new point. This means that Powell's criterion is satisfied, and thus the properties ((P1) and (P2)) in Section 3.3 will be satisfied.

$i = 0$;
$Q_k^i = Q_k^{\text{icb}}$;
for $i = 1, 2, \ldots$ until $\tilde{\Delta}_k \leq \mu||\nabla \mathcal{L}_k^i||$ **do**
  Use Algorithm 1 to improve the previous model, Q_k^{i-1},
  until it is FL on $B(\mathbf{x}_k, r\omega^{i-1}\Delta_k^{\text{icb}})$. Denote the new model Q_k^i.
  $\tilde{\Delta}_k = \omega^{i-1}\Delta_k^{\text{icb}}$;
  $\tilde{Q}_k = Q_k^i$;
end

Algorithm 4: Criticality step: first order[1].

The Model improvement step (step 4) is only executed if $\mathbf{x}_{k+1} = \mathbf{x}_k$ after step 3. That means that the step, $\mathbf{s}_k$, produced in step 2 wasn't accepted. This, in turn, implies that the model mimics the true function insufficiently. The point $\mathbf{x}_k + \mathbf{s}_k$ may or may not have been included into the interpolation set. The point that is furthest away from $\mathbf{x}_k$ is attempted

replaced with a point that improves the poisedness. Note that the point $\mathbf{x}_k + \mathbf{s}_k$ might have been included into the interpolation set. However, as stated in [1], this replacement will only either improve the poisedness or replace a point that is far away. Hence, if the model, m_k was FL at the beginning of step 3, then it will also be FL at the end of that step [1].

In the last step, the Trust-region radius step, the model is updated to include the new point(s) provided by the previous steps. If another point than $\mathbf{x}_k + \mathbf{s}_k$ has been included into into the interpolation set, one additional function evaluation is required.

In this algorithm the number of sample points, m, is fixed. Considering that we are dealing with expensive objective functions, maybe it could be an idea to include the already evaluated points if they are within the trust-region. However, this is not done in this algorithm. There exists different versions of the algorithm in the book [1], where at least one of them use a dynamic number of sample points. Even though old points are not included as interpolation points of the model, some information from the previous points are still part of the model, because of how we update the model. Remember that we use the old Hessian in the process of determining the new one. We minimize the change, thus, in some sense, information from the old points are still there. Of course, we could have included more information as is done in other algorithms. The idea of the interpolation model is not to very accurately replicate the true function, all that we want is to capture some information such that it will guide us towards the optimum in an iterative process.

3.12.3 Comparison with Powell's algorithms

As mentioned in the literature review, there are several differences between the algorithm just presented and Powell's algorithms (e.g., NEWUOA[10]). The comparison is done with the constraints taken out of the equation. The following three differences are given in the book [1]. Powell uses two different trust-region radii and they are not related by a constant factor like they are in our algorithm (Δ_k and $r\Delta_k$). The radius of the interpolation set will, in theory, eventually converge to zero, the trust-region radius is allowed to be bounded away from zero. The reason for allowing the trust-region radius to remain bounded away from zero is to allow large steps even when we are close to the optimum. The smallest of the two radii are also used to keep the interpolation points sufficiently spaced to avoid the influence of noise.

Another difference is how to select the point that should be replaced when a new, better point is included. Instead of working with $\mathbf{y}_i = \arg\max_j\{||\mathbf{x}_k - \mathbf{y}_j||_2|\ell_j(\mathbf{x}_k + \mathbf{s}_k)| : \mathbf{y}_j \in \mathbf{Y}_k\}$ as is done in the algorithm above, Powell suggests to optimize the coefficient of the rank-two update update of the system defining the Lagrange polynomials, which will explicitly improve the conditioning of the system. The rank-two update is a name of the quadratic model that is added to the old model, to get the new one. And the reason for it being called "rank-two" is that the $\mathbf{W}_{\text{new}} - \mathbf{W}_{\text{old}}$ is of rank two. The $\mathbf{W}$'s only differs in one row and one column, which can be seen from (3.10), (3.9) and the definition of $\hat{\mathbf{Y}}$ (remember that $\mathbf{W}$ is defined by the interpolation points alone).

Powell does not explicitly work with the concept of poisedness, and thus there is no testing of $|\ell_i(\mathbf{y}_i^*)| > \Lambda > 1$. The model-improvement step is far more complex, but the basic idea for selecting new points is the same. Namely, to choose a new point such that it gives a large value of the Lagrange polynomial belonging to the point that is going to be replaced. A perk of doing it this way, is that there is no need of doing the optimization of the Lagrange polynomials as is done in the model-improvement algorithm. Avoiding the optimization in Algorithm 1 is a tempting idea. Luckily, Algorithm 1 can be modified to include these kind of "cheap" steps, i.e., replace a point by another for which the absolute value of the corresponding Lagrange polynomials exceeds Λ.

The details of this idea are not suggested in [1]. A suggestion by the author of this report for how to do this is as follows. Instead of choosing the point that should be removed based upon the maximization of $|\ell_i(x)|$ within the ball, one could use the strategy as is given in the model-improvement step, i.e., choose the point that is farthest away from the current best point, $\mathbf{y}_t$ say. Use the approximation techniques given in Section 3.3 to find an upper bound. If the upper bound is greater than Λ, find a new point, $\mathbf{y}_t^*$ say, by using (a version of) the truncated gradient method on $\ell_t(\mathbf{x})$ and $-\ell_t(\mathbf{x})$. If the upper bound is lower than Λ, go to the next farthest point, and repeat. The only criterion for the point, $\mathbf{y}_t^*$, is that it should give a large value of $|\ell_t(\mathbf{x})|$ which is greater than Λ (and be within the trust-region).

However, using this kind of improvement strategy has one disadvantage. As mentioned, the number of iterations of Algorithm 1 is uniformly bounded. If we use the method just proposed (i.e., allowing "cheap" steps), then this property is weakened. The algorithm will then complete in a finite number of iterations. A strategic trick is to modify Algorithm 1 to allow for a fixed number of consecutive cheap steps, and if the limit is reached, switch to global optimization[1].

Another difference, which is sort of a consequence of the differences already given, is that Powell's algorithm is quite complex and the program flow is not easy to follow. The algorithm given in this section follows a nice loop, Step 1 - Step 5, whereas in Powell's algorithm there are a lot of jumps based upon what is happening. To see the flow, look at Figure 1 in [10]. The approach taken seems to be more of a "keep going as long as possible, and only use resources to fix the conditioning of the system if needed"-strategy, whereas in the given algorithm, maintaining the poisedness of the interpolation set is an essential part of the loop. This difference is observed by the author and is not given in [1]. Considering practical performance (measured in total run time), the author would not be surprised if Powell's algorithm performs better as less function evaluations are probably used to maintain the poisedness of the interpolation set.

Another difference is their capability of finding good local optimums, or potentially global optimum. There is done a comparison between different derivative-free algorithms in [29]. In Figure 2 in that paper, we can see which points have been evaluated by the algorithms. The function "six-hump camel back function',' which has six local optimums where two of them are global ones, where used. They were initiated with the same starting

point, which is closer to a local optimum than a global one. In the test, amongst others, NEWOUA and a method named DFO, which is more or less the same as the one presented in this thesis, were compared[10]. The result of the test on the six-hump camel back function was that the NEWOUA algorithm found the closest local optimum very fast, whereas the DFO method found the global optimum. The reason for why the DFO algorithm was able to find it, is due to how it handles the trust-region radius and how it slowly decreases. If the initial radius is "big enough", and the initial point is close to a local optimum, we can still be able to find a better local optimum, this is because we slowly zoom in on the function and while we do this, new points will be evaluated and we might stumble upon points with lower objective function values and the attention is moved towards that point.

In this chapter, some of the relevant theory to create a globally convergent derivative-free model-based trust-region algorithm has been presented. The main focus has been on the building, updating and maintaining an under-determined quadratic model. Information regarding how to include gradients into the model-making process has been given. The derivative-free model-based trust-region algorithm has been extended to handle constraints. Lastly, these components were put into a globally convergent framework which we presented and explained.

[10]The DFO algorithm is also given in the book [1]. The difference is in how it handles geometry of the set of interpolation points and that the number of interpolation points can change during the algorithm.

Chapter 4

Testing of the algorithm

In this chapter we will see if the selected method of incorporating constraints into the optimization procedure works. In addition, we will see which effect it has to include gradient information into the model-making process. These are the main goals of this chapter. At the end of the chapter, the algorithm is applied to a well placement task to see if the algorithm is able to find a minimum when a simulator is used as the true function.

While testing the chosen constraint handling technique, we will also explore the effect of varying the number of interpolation points, m. In addition, some comments on different aspects of the convergence of the algorithm will be given.

4.1 Incorporating constraints

The selected method to deal with constraints is to simply add the constraints into the subproblem as previously discussed. The feasible area defined by the constraints is added as a mask on top of the trust-region. The constraints are assumed to be hard or unrelaxable. This means that the constraints must be included while checking and improving the poisedness of the set.

Before the test problems are presented, some comments on the implementation is given and the chosen parameters are listed.

4.1.1 Implementation details

Until now the trust-region has been defined by the the Euclidean norm. However, in the implementation, the infinity norm is used. This is because this constraint can be defined by bounds on the variables instead of adding a nonlinear constraint which has a undefined gradient at the origin. It is mentioned in [1] that it is an option to use this norm.

The following parameters are found by trial and error and are by no means optimized and further tuning and testing should be performed.

$$
\begin{aligned}
r &= 2 \\
\omega &= 0.66 \\
\beta &= 0.09 \\
\tau &= 0.1 \\
\eta_1 &= 0.1 \\
\gamma &= 0.7 \\
\gamma_{icb} &= 1.2 \\
\epsilon_c &= 0.1 \\
\mu &= 3 \\
\Lambda &= 2
\end{aligned}
$$

The sophisticated updating scheme was implemented during the specialization project, but because there was some hidden bug(s), I decided to switch to the simple scheme instead.

The termination criterion is the trust-region radius. When it becomes less than, Δ_{min}, the algorithm terminates.

4.1.2 Test problems for the constraint handling

The tests in this part are performed in 2D such that it is easy to visualize the problems and solutions. These two functions will be used:

$$f_m(x_1, x_2) = 0.26(x_1^2 + x_2^2) - 0.48 x_1 x_2 = \frac{1}{2} \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 0.52 & -0.48 \\ -0.48 & 0.52 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.1}$$

$$f_r(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{4.2}$$

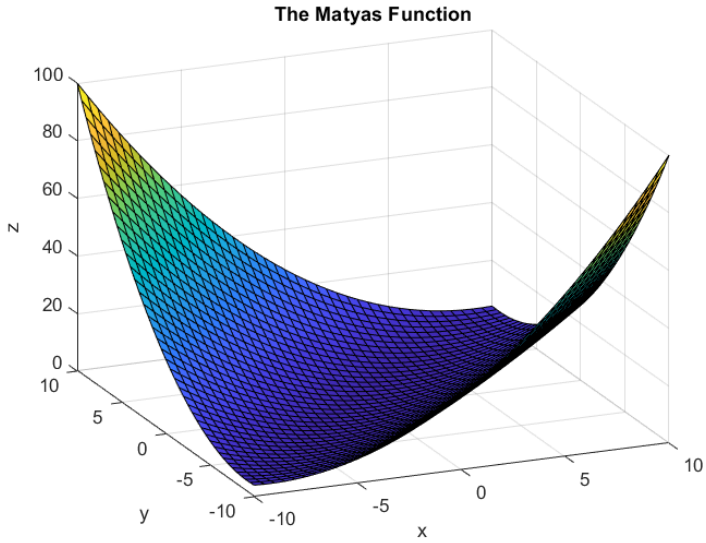
These are two classical test functions for optimization. The Matyas function, (4.1), is a convex second order polynomial with global optimum at $(0, 0)$ and $f_m(0, 0) = 0$. It is plotted in Figure 4.1. The Rosenbrock function, (4.2), is a lot more complex. It is also a polynomial, but it has many stationary points along the valley, as can be seen in Figure 4.2. Its global optimum is at $(1, 1)$ and $f_r(1, 1) = 0$.

The chosen bounds for the Matyas function is

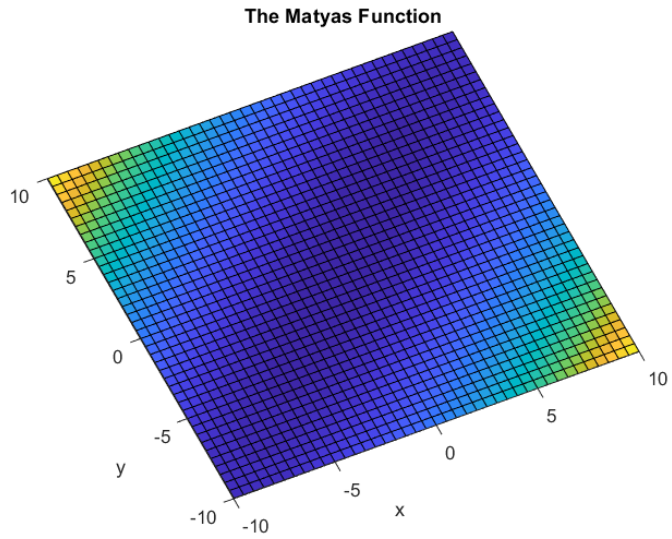
$$\mathbf{lb}_m = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \leq \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 5 \\ 5 \end{bmatrix} = \mathbf{ub}_m \tag{4.3}$$

and the chosen bounds for the Rosenbrock function is

$$\mathbf{lb}_r = \begin{bmatrix} 1.5 \\ 1.2 \end{bmatrix} \leq \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 5 \\ 5 \end{bmatrix} = \mathbf{ub}_r \tag{4.4}$$

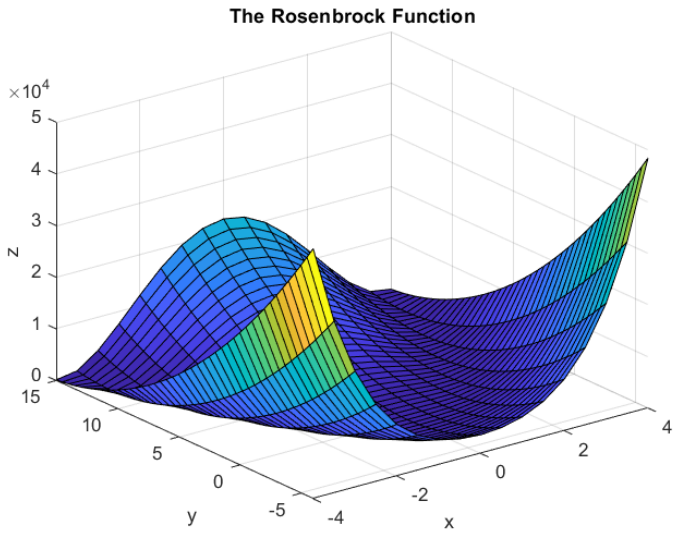


(a) The Matyas function. Global optimum at $(0, 0)$.

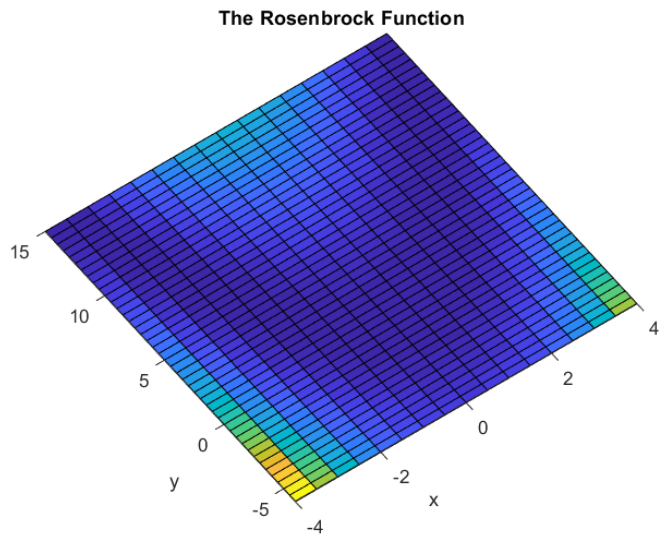


(b) The contours of the Matyas function. Global optimum at $(0, 0)$.

Figure 4.1: In the figures above, the Matyas function is plotted. The darker the blue color is, the lower the function value is.



(a) The Rosenbrock function. Global optimum at $(1, 1)$. There are many local optimums along the valley



(b) The contours of the Rosenbrock function. Global optimum at $(1, 1)$.

Figure 4.2: In the figures above, the Rosenbrock function is plotted. The darker the blue color is, the lower the function value is.

There will be five different test problems. The first one, Case 1, is the unconstrained case with an initial point of $(4, 4)$. Then the four following cases will be tried.

- **Case 2**. Bounds as given in (4.3) and (4.4).
  Initial point: $[4, 4]$.
- **Case 3**. Bounds and linear constraint.

  $$y \geq 6 - x$$

  Initial point: $[4, 4]$.
- **Case 4**. Nonlinear constraint.

  $$y \geq x^2 + 1.2$$

  Initial point: $[0, 4]$.
- **Case 5**. Bounds and nonlinear nonconvex constraint.

  $$y \geq 3\cos(0.5x)$$

  Initial point: $[4, 4]$.

The initial point is only changed in the nonlinear constraint to make it feasible. The answer for each of the cases are given in Table 4.1. The subscripts "m" and "r" indicate that they belong to the Matyas and the Rosenbrock function, respectively. The * will indicate optimal values, either global or local.

Table 4.1: The table shows the answers of the different cases for both the Matyas function and the Rosenbrock function. The optimums are all global.

	x_m^*	y_m^*	$f_m^*(x_m^*, y_m^*)$	x_r^*	y_r^*	$f_r^*(x_r^*, y_r^*)$
Case 1	0	0	0	1	1	0
Case 2	0.5	0.5	0.01	1.5	2.25	0.25
Case 3	3	3	0.36	1.9996	4.0004	0.9996
Case 4	0.4294	1.3844	0.2609	1	2.2	144
Case 5	1.8355	1.8228	0.1339	1.5	2.25	0.25

4.1.3 Results - constraint handling

The main purpose of this section is to show that the chosen constraint handling technique works. All the cases 1-5 for both test functions have been performed. The results are plotted in Figure 4.4 and Figure 4.5. These figures show that the constraint handling is working as expected. All the evaluated points are plotted in the figures. We can see that none of the points are violating the constraints. The red marks are points that have been found while solving the subproblem. The green mark is the final solution, and the black

mark is the initial point. The constraints are painted in black. For all tests in this subsection, the initial-trust region radius was set to $\Delta_0 = 1$, the final was set to $\Delta_{\text{min}} = 0.001$ and the max was set to $\Delta_{\text{max}} = 2$.

If there were any soft(relaxable) constraints. Then there would have been blue marks violating the constraints, whereas the red ones should still obey all the constraints.

A lot of test have been performed on Case 1-5. For each case and for each function there have been done three tests with different m's. This allows for all the different options to be tested. The minimum is $m = n + 2 = 4$, the recommended value by Powell is $m = 2n + 1 = 5$, whereas the maximum number is $m = (n + 1)(n + 2)/2 = 6$. Performing these tests is not a main goal of this thesis, but it is an interesting field to explore. For each test there are one figure and one table. All of them are available in Appendix A. Figure 4.3 and Table 4.2 are an example. They are repeated here such that an explanation can be given. The top left subfigure is the feasible area plotted together with the contours of the function. The other three plots show the different points that have been evaluated for the three different m's. The colors are as defined above.

The Table 4.2 shows the answers that were found for the different scenarios in Figure 4.3, and it also tells how many function evaluations were needed to converge, n_e. The n_p value tells how many times we would need to evaluate functions if parallalization was supported.

Table 4.2: The Matyas function. Nonlinear constraint.

	m=4	**m=5**	**m=6**
n_e	25	35	50
n_p	16	19	25
(x,y)	(0.42946, 1.38443)	(0.42939, 1.38438)	(0.42939, 1.38438)
f(x,y)	0.26090	0.26090	0.26090

Comments on the algorithm - Convergence

Figure A10 in Appendix A shows some interesting aspects of the algorithm. This is the scenario where the Rosenbrock function is minimized subject to bounds and a nonconvex nonlinear constraint. For the cases when $m = 5$ and $m = 6$ we can see that the algorithm converges to the global optimum, despite being attracted into the valley during some of the first iterates. However, a high amount of function evaluations are required. This is because the gradient of the Lagrangian is small and a lot of small steps are performed, we can also see (by noticing how close the blue marks are to the red ones) that the trust-region is getting small too. The reason why it is able to keep on going despite being around the stationary points, are because when we keep on improving the geometry, we keep on evaluation points around the current iterate, and these new points will contribute with new

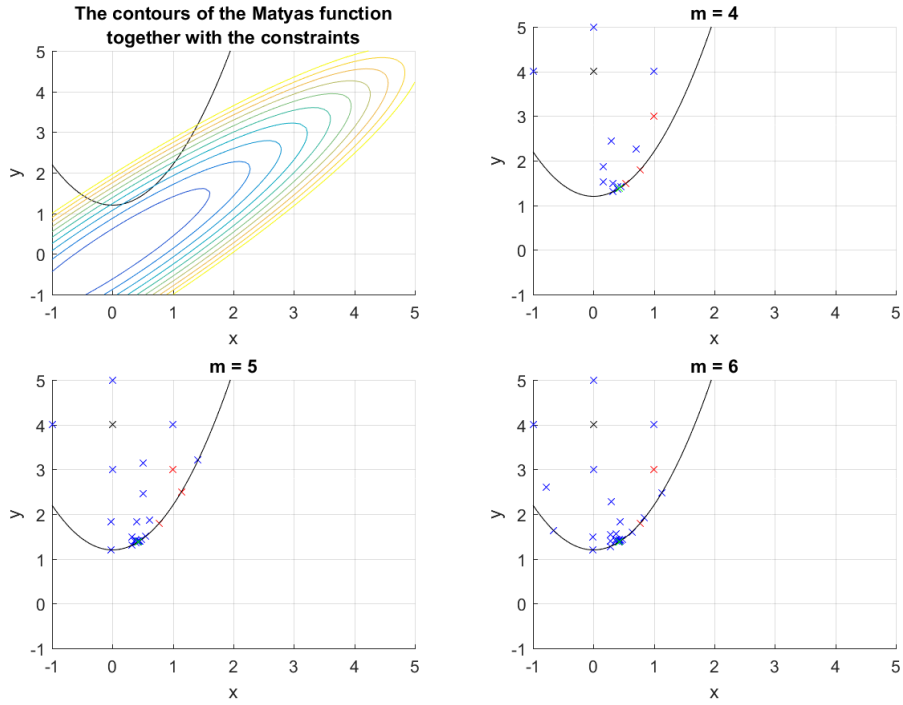


Figure 4.3: The top left plot shows the contour of the function together with the constraint. The three other plots shows which points have been evaluated. The difference is the amount of sample points used to create the model, i.e., m. The value is given in the title of each figure. Green mark means final solution. Red mark means point found by solving the subproblem. Black mark is the initial point. The black line is the constraint.

information.

If they are chosen in the direction of the global minimum they will contribute with a lower function value and when this is included into the model, we will go in that direction the next time we solve the subproblem. Another possibility is that the points may contribute with worse function values if they are placed a bit "higher up the valley", and this can lead to that the curvature of the model will start to coincide with the curvature of the true function, and this can also give us a lower value when the subproblem is solved. In other words, the points that gives a high function value will make the algorithm go away from that area.

When $m = 4$, we are not that lucky with which points are found during the geometry improving, and we get stuck in a local optimum. This is highly likely because too few points are being sampled, and thus the algorithm is not able to capture enough information of the true function.

If the behaviour in the cases when $m = 5$ and $m = 6$ is undesirable, i.e., it is preferable to just converge to a local optimum, two different approaches can be taken. The first is to simply increase the final trust-region radius, Δ_{min}, such that the algorithm is not allowed to take these small steps. The other approach is to add a restriction on how many consecutive steps with a small decrease is allowed. For the second approach the user will have to specify how many small decreases he/she would allow and what would be defined as a small decrease. If the user has knowledge of the application area this should be doable. Both of these approaches could be combined.

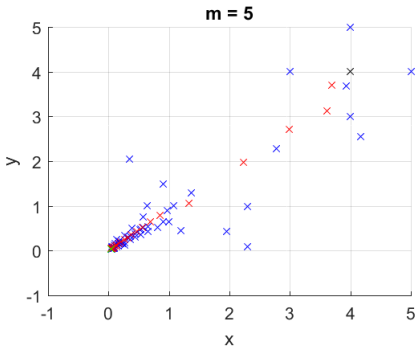
Comments on the algorithm - Parallelization

By comparing the values n_e and n_p in all the tables in Appendix A, we can see, not surprisingly, that parallelization has the potential to increase the effectiveness of the algorithm by a good amount!

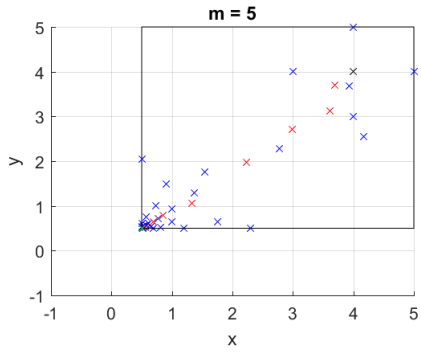
Comments on the algorithm - Selecting m

The choice of m is not simple. The recommended value by Powell does an overall decent job. However, sometimes $m = 4$ is best and sometimes $m = 6$ is best. Further testing must be done to conclude anything. But following the recommended value by the creator of the method will be the choice for now.

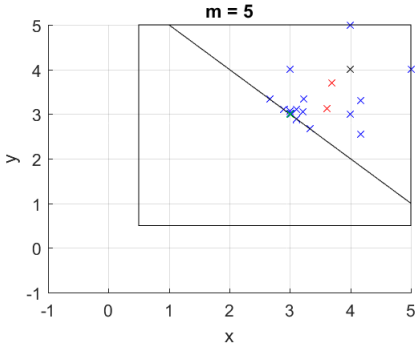
Summary Now that it is shown that the chosen constraint handling technique works, we will change focus towards including gradients into the model-making process. The author would like to point out that he is aware that more testing should be done before we can conclude that the implementation is robust when it comes to handling constraints. However, the chosen method is suggested in [1] and we have produced results that are satisfying and show that the implementation handles the constraints as it should. A natural question is how many constraint there can be. This will depend upon at least two parts.



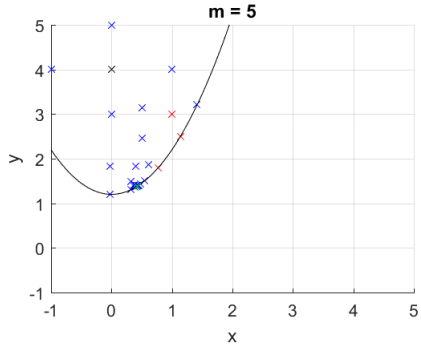
(a) Unconstrained.



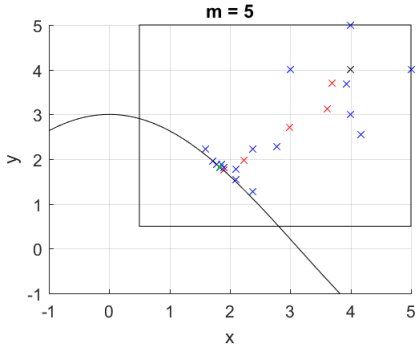
(b) Bounds.



(c) Bounds and linear constraint.

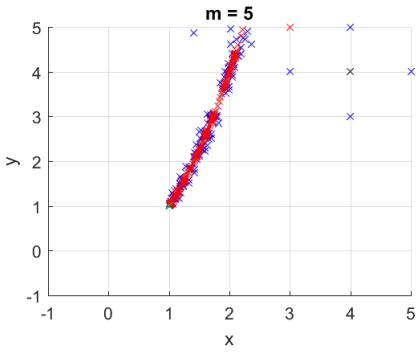


(d) Nonlinear constraint.

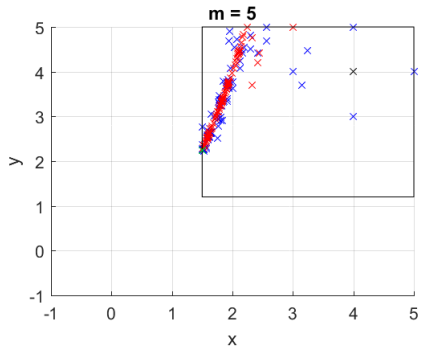


(e) Bounds and nonlinear nonconvex constraint.

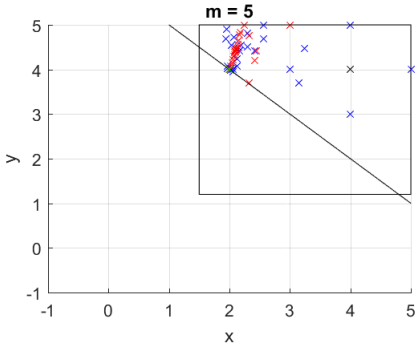
Figure 4.4: The algorithm converges with all the different constraints. The Matyas function is used and $m = 5$ for all tests.



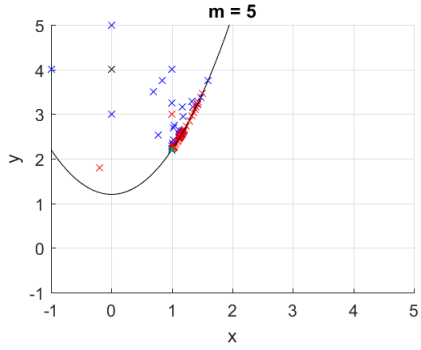
(a) Unconstrained.



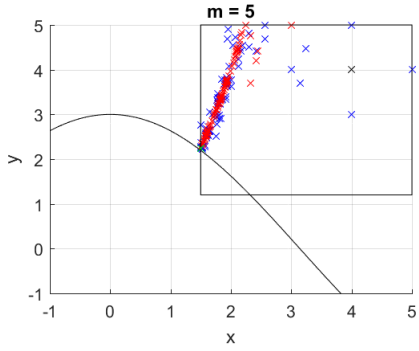
(b) Bounds.



(c) Bounds and linear constraint.



(d) Nonlinear constraint.



(e) Bounds and nonlinear nonconvex constraint.

Figure 4.5: The algorithm converges with all the different constraints. The Rosenbrock function is used and $m = 5$ for all tests.

One limitation is how good the third-party solver is to solve the subproblems. However, the chosen solver (SNOPT) is a very solid and robust solver, thus this is not the first limitation. Another part that must be considered is how much one can restrict the feasible area and still be able to find feasible points such that the required poisedness can be achieved.

4.2 Gradient enhanced models

4.2.1 Convex functions

In this section we will see how gradient information included into the model-making process effects the total number of function evaluations and the quality of the solution. First, the Matyas function will be used. Then the 10 dimensional sphere will be tested. The references cases will be $m = n + 2$, $m = 2n + 1$ and $m = (n + 1)(n + 2)/2$.

In the theory chapter, two ideas for applying gradient information were given. One was to only include gradients at the center point of the model. The other one was to include all the available gradient information by solving a minimization problem while the gradient of the center point was still set exactly. The first case corresponds to setting $\alpha = 1$ and setting the n_g to the number of available derivatives of the objective function. The second case corresponds to setting α to a value in the open interval 0 to 1. If $\alpha = 0$, then n_g must be set to n. This means that all coefficients will be determined by the interpolation conditions and the gradients, and the minimum change of the Hessian is ignored.

Table 4.3: The table shows the difference in the amount of function evaluations needed to converge when different parameters are set. The Matyas function, (4.1), was used as test function. In all scenarios, the global optimum was found.

	m	n_g	α	n_e
No gradients	6	0	1	117
	5	0	1	99
	4	0	1	78
Gradients	3	2	1	47
	2	1	0.5	33
	2	2	0.5	34
	2	2	0	34

The results are given in Table 4.3. From the table, we can see that including gradient information can reduce the amount of evaluations needed by up to a factor of around 3. Thus, we can conclude that including gradient information into the model-making process can be beneficial when we have a convex function such as the Matyas function. To further test this idea, we will, as told, use the multidimensional sphere.

The results can be seen in Table 4.4. The conclusion are the same as the one in the previous paragraph. Including gradient information into the model is preferable when we

Table 4.4: The table shows the amount of function evaluations needed to converge when different parameters are set. The 10 dimensional sphere was used as test function. In all scenarios, the global optimum was found.

	m	n_g	α	n_e
No gradients	66	0	1	588
	21	0	1	267
	12	0	1	254
Gradients	10	10	1	122
	10	10	0.5	81
	10	5	0.5	81

have a convex function such as the sphere.

Until now, it seems like including the gradient information is an option that should be followed when such information is available. However, this view will change during the next tests.

4.2.2 Nonconvex function

Here we will inspect how well the gradient enhanced model performs when the function is no longer convex. The Ackley function (3.37) will be used for this purpose.

Table 4.5: The table shows the difference in the amount of function evaluations needed to converge when different parameters are set. The Ackley function was used as test function. In all scenarios, the global optimum was found.

	m	n_g	α	n_e
No gradients	6	0	1	65
	5	0	1	71
	4	0	1	74
Gradients	2	2	1	82
	2	2	0.5	48
	2	1	0.5	79

The results are given in Table 4.5. To the author's great surprise, the algorithm found the global optimum in all of the scenarios given in the table. In addition, we can see that the second last row ($n_g = n = 2, m = 2, \alpha = 0.5$) is the one that used the least amount of function evaluations! These results were unexpected. The initial point was $[4, 4]$ and the initial trust-region radius was set to 8, the Δ_{max} to 12 and the final trust-region radius to 0.001.

The prediction in part 3.6.1 was that the gradients would make the algorithm less "immune" against bad local optimums in nonconvex and/or noisy functions, but in the just

performed test the algorithm performed **better** with the gradient enhanced model.

A change in the initial trust-region radius was made, and it was set to 9.6 (instead of 8.0). The algorithm converges to a local optimum after 112 function calls, the parameters were $n_g = n = 2, m = 2$ and $\alpha = 0.5$. The algorithm was ran once more, but with no gradient information and $m = 5$. The global optimum was found after 78 function evaluations. This means that the gradient enhanced model performed better only because it was lucky with which points were selected. To demonstrate how fragile the gradient enhanced model are to the point selection, we will perturb the starting y-coordinate by small amounts. For each initial starting point, the algorithm is ran with both the gradient enhanced model ($n = n_g = 2, m = 2, \alpha = 0.5$) and the "normal" model ($n = 2, m = 5, n_g = 0$). We will take a note of if it did find the global optimum and if not, what was the function value at that point. The number of function evaluations are also stored.

Table 4.6: The table shows how fragile the gradient enhanced model is to the selection of points. The gradient enhanced model is compared with the regular model. The only difference for for each test is the initial point. The second coordinate is changed from 4.0 to 4.5 as is shown in the y-column. The first coordinate remains unchanged.

	y	n_e	Global?
No gradients	4.0	67	Yes
	4.1	57	Yes
	4.2	60	Yes
	4.3	75	Yes
	4.4	72	Yes
	4.5	53	Yes
Gradients	4.0	48	Yes
	4.1	52	Yes
	4.2	38	13.6016
	4.3	52	Yes
	4.4	67	3.5745
	4.5	42	6.6745

Table 4.6 shows the results. The regular model is able to find the global minimum every time, whereas the gradient enhanced model is not. This emphasizes that the gradient enhanced model is very exposed to local minimums, which are either due to the function or due to noise. The gradient enhanced model has one advantage, which is that in all cases it converges faster (i.e., needs less function evaluations) to the local optimum. If we know that we don't have a noisy function and we don't mind finding local optimums, the gradient enhanced model may be used. Remember that the algorithm (with or without the gradients) is not a global optimizer. However, as we have seen, it possess the property of finding a good local optimum (or even the global optimum) despite the function being noisy and/or containing a lot of minimums.

4.2.3 10 dimensional nonconvex function

Until now mainly 2 dimensional functions have been used. To test if the algorithm still works when the dimension is increased, we will try a 10 dimensional nonconvex function. Three different aspects will be tested.

1. Is the value that Powell recommended, $m = 2n + 1$, a good choice?
2. How will the gradient enhanced model perform now that there are more degrees of freedom? I.e., more degrees of freedom are not taken up by the interpolation conditions.
3. Will we be able to find the global optimum?

The function we will use is the 10 dimensional Rastrigin function:

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n}(\mathbf{x}_i^2 - 10\cos(2\pi\mathbf{x}_i)) \tag{4.5}$$

where $n = 10$ and the search domain is $-5.12 \leq \mathbf{x}_i \leq 5.15$. The global optimum has the value 0 and is located at $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$. The 2D version of the function can be seen in Figure 4.6. The starting point was $[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]$, and the initial starting radius was set to 3, the max to 8 and the final trust-region radius was not changed.

The results are given in Table 4.7. First, note that taking advantage of parallelization is highly advantageous in all cases. Remember that n_p is the number of simultaneously evaluations of the function, whereas n_e is the total number. For the cases where no gradient information were included, Powell's recommended number of interpolation points was indeed the best. If $m = n + 2$, then we are not able to explore the search area well enough and only a local solution is found. However, a lot less function evaluations was needed. When the gradient information was included, only local optimums were found in all scenarios. The amount of available gradients was set to $n_g = 5$ and $n_g = 10$. In all of these scenarios the amount of function evaluations were lower than for the scenarios with no gradient information. The table demonstrates how the gradient information actually makes the model worse. We went from finding the global optimum with $m = 21$ to only finding local optimums when m remains unchanged and n_g was set to 5 and 10. Another interesting fact is that using $m = n + 2 = 12$ with no gradient information finds a better local optimum than any of the runs where gradient information were included. It can be seen from the table that, once again, the algorithm converges using less function evaluations when gradient information is included.

The algorithm has been tested on different ordinary mathematical functions. The results have been promising. Of course, more testing should be performed, but the preliminary results are promising. As mentioned throughout the this thesis, the true function could also have been a simulator (i.e., a black-box). In the next section, this will be the scenario.

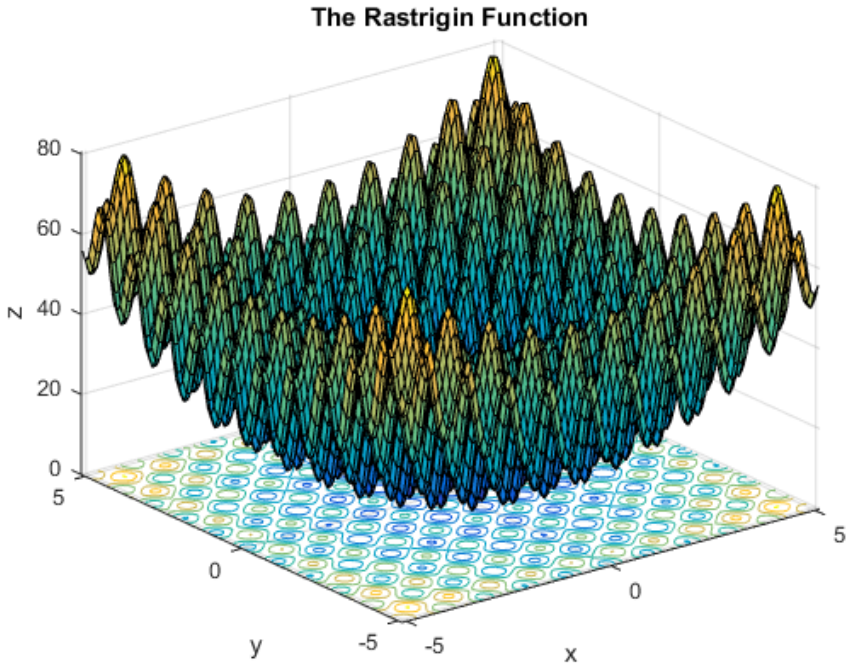


Figure 4.6: The 2 dimensional function of (4.5). I.e., $n = 2$. The figure illustrates that the this function is highly nonlinear and nonconvex.

Table 4.7: The table shows how the algorithm performs on a nonconvex 10 dimensional function. All parameters not mentioned in the table remains unchanged.

	m	n_g	$alpha$	n_e	n_p	Global?
No gradients	12	0	0	147	66	20.8952
	21	0	0	223	61	Yes
	66	0	0	486	32	Yes
Gradients	10	10	0.5	78	37	120.0418
	21	10	0.5	109	34	100.7540
	21	5	0.5	140	49	49.5252

4.3 Testing on an oil reservoir simulator

The algorithm was tested in the well placement challenge. This test is just to show that the algorithm also work when the true function is not an analytic function. We have already showed that it converges to local optimums when the true function is an ordinary mathematical function.

Only a restriction on the well length of the producer well was added in this case. Implementing the constraints in FieldOpt is not straightforward. The current constraint handling is based upon adding penalty terms to the objective function. That means that the optimization algorithm finds a point, and first then the constraints are dealt with. If the point is infeasible, the point will be projected back into the feasible area.

The road taken in the algorithm presented in this thesis is quite different. The constraints are always included such that only feasible points are produced. The difficulty arises because of how FieldOpt is constructed. The idea in FieldOpt is that the optimization algorithms should be agnostic to the variable type. I.e., they should not care if the variable represents a z-coordinate or the bottom hole pressure (BHP). The order of the elements of the vector of decision variables is random. However, this also complicates the implementation of our type of algorithm. In contrast to the agnostic philosophy of FieldOpt, we need to know the variable type and we need to know the exact meaning of the variable. E.g., if we want to impose a restriction of the well, we must know which variables corresponds to the heel and the toe of the well. In addition, we would like to scale the variables differently.

The functionality to extract the gradients are not ready in FieldOpt. However, based upon the result in the previous section, this is not a big drawback. Most likely the gradient enhanced model would, at its best, find the same good local optimum as the algorithm would have found without gradients.

4.3.1 Scaling

Before the results are shown, a small note on scaling will be given. In general, scaling of the decision variables and the objective function value and the possible constraints are very important for the speed of the convergence of an optimization algorithm. The objective function returned by FieldOpt is already scaled. The scaling procedure can be used to achieve two different goals. The first goal is the most obvious one which is to get the magnitude of the variables to about the same level. The second one is less intuitive and to motivate it an example from the application area is given.

Let's say that we are perturbing the toe of the well. The perturbations along the xy-plane should be of many magnitudes larger than the perturbations along the z-axis. This is simply how oil reservoirs are made by the nature. The thickness (i.e., along the z-axis) of the layer of oil is very small compared to how wide (i.e., along the xy-plane) the layer is. To take this consideration of the variables into account, scaling can be applied. The idea is that those variables that are not allowed to change a lot will be multiplied by a factor greater than 1. Let's call this scaling factor β. The value of β will depend upon the relationship between the different variables. The goal is to make it such that, approximately, an increase of 1 in each variable will represent a desired step of the "unscaled" variable. E.g., if the scaled z-coordinate change by 1, maybe the real change is about 1 meter, whereas for the x-coordinate the real change is about 30 meters or more.

The result of this scaling will be that when you search for a solution of the subproblem within the trust-region, the boundary of the trust-region will be hit faster by those variables that are not allowed to change that much. This is because a step along that axis will be multiplied by β.

Another method to achieve the same kind of limitations on how much the different variables are allowed to change, is to simply scale the trust-region directly[15]. If a variable is not allowed to change a lot (such as the z-coordinate of the toe of the well), then the trust-region would be smaller along that axis. The scaling of the trust-region might be more logical, however, if we follow the other approach, then we only have to scale once. Considering that we are already scaling the variables to obtain approximately the same magnitudes, doing another scaling at the same time is not much of an extra work.

The x and y coordinates were scaled by $1/10000$ and the z coordinates were scaled by $30/10000$ in the testing of the algorithm.

4.3.2 Results of the well placement challenge

The chosen case has three wells where two of them are injectors. Injectors are wells that inject fluid to maintain the pressure of the reservoir such that the oil will keep on going up of the reservoir. These two wells have fixed positions. The last well is a producer. Producers are the wells where fluid goes up, i.e., the wells that produce the oil and gas. This well is the well that we will optimize on. Both the toe and the heel of the well are variables. That means we have 6 decision variables in total. A restriction of the maximum length

was added and set to 500. It was tried to do the optimization without this restriction, but it led to a solution which had a very long well.

The reservoir can be seen in Figure 4.7. The meaning of the colors are given in the description of the figure. It is a little hard to see, but the injector wells are marked with "I01" and "I03". The well that is going to be optimized is marked with "P01". The total simulation time is about 6 years.

Let T_o, T_g and T_w be the total accumulated oil, gas and water, respectively. They are given in standard cubic meter (SM3). The objective function is defined as follows:

$$f = 299.9611e^{-7}T_o + 0.15032e^{-7}T_g - 4.9690e^{-7}T_w$$

As we can see, the water contributes with a negative factor in the objective function. The initial trust-region radius was set to 0.001 and the final one was set to 0.00001. If the initial trust-region was set too big, the initial set of perturbed vectors would be infeasible. This is a big challenge in constrained optimization. However, the goal of this test is to show that the algorithm works when the true function is a simulator, and the goal is not to find the best possible well location. Of course, the goal is to find an improved position, i.e., a local optimum, but we will not use resources on finding the optimal parameters to find a best possible local optimum.

We used the suggested value by Powell with $m = 2n + 1$. The objective function had an improvement of about 226%. The initial well position was

$$[7864.2123, 14229.1378, 1533.4939, 8000.8869, 14236.4459, 1541.9891],$$

where the three first values are the $[x, y, z]$ coordinates of the heel, whereas the three last are the same coordinates of the toe. The final position was

$$[7690.4719, 14097.4035, 1532.4248, 8153.8992, 14284.7164, 1544.6308].$$

The change of the location of the well can be seen in Figure 4.8. If we compare the initial and the final positioning of the well, we can see that the z-coordinates have changed by only a couple of meters, whereas the others has been changed by up to around 150 meters. Some graphs of the most important key information is given in Figure 4.9 and 4.10. The total amount of function evaluations was 314 and if parallalization was utilized it would have been 96. This might seem like a high number, but the author would like to point out that the resolution of the solution is quite high. By that it is meant that the final trust-region radius is very small and the final improvements will be very small. The step sizes at the end of the optimization procedure is around 0.1 meters[1].

The author does not have knowledge about petroleum, but he contacted a petroleum engineer, Brage K. Strand, for help. The engineer did not do a thorough analysis, but his preliminary conclusion was that the optimized well location offers a solid improvement.

[1] It was tested with a bigger starting radius and a higher final radius. This will be presented soon.

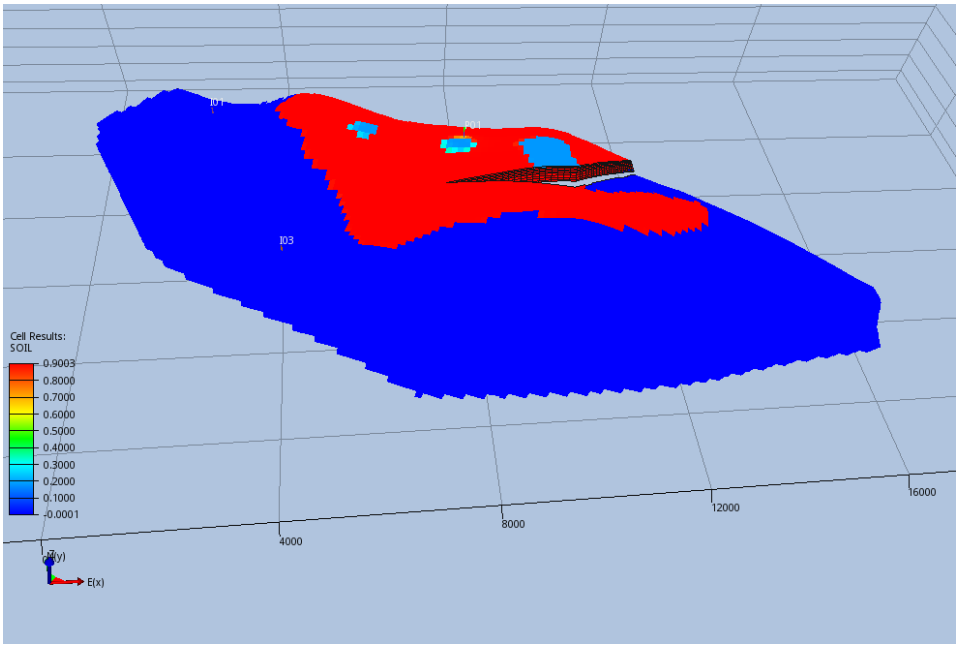
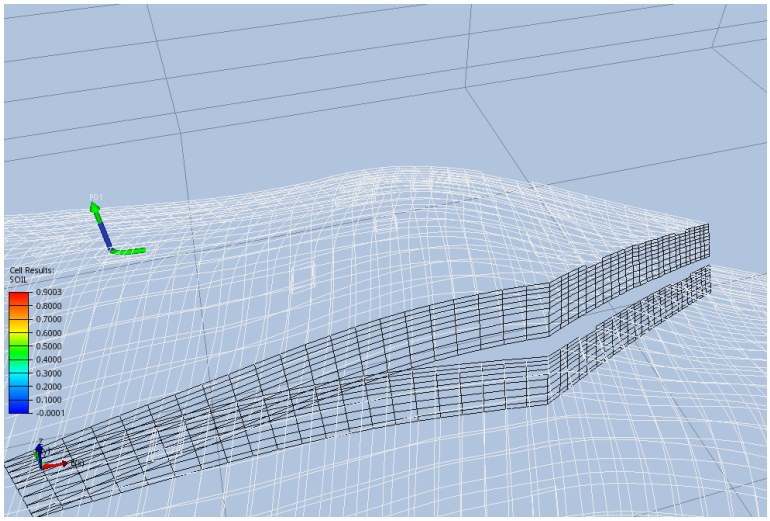


Figure 4.7: The reservoir which has been used for testing. Red indicates oil, brown indicates gas and blue indicates water

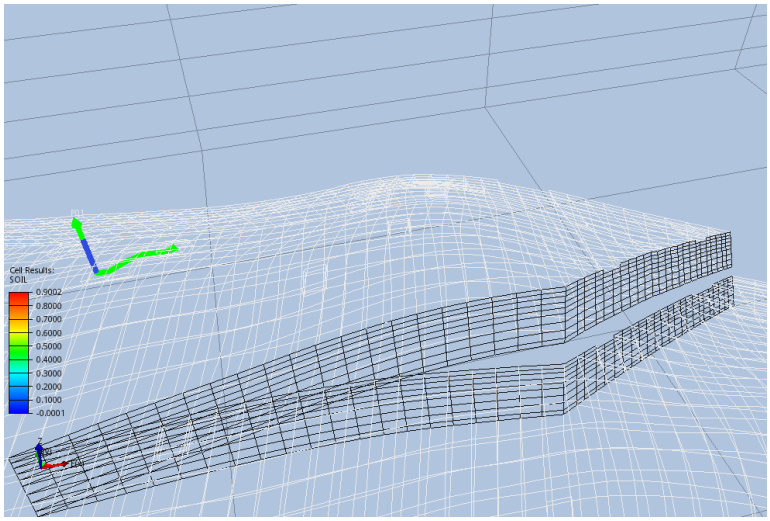
We can see from Figure 4.9a that the oil production is more than doubled, and the same goes for the gas production (Figure 4.9b). The two lower graphs in Figure 4.9 might seem a bit alerting. However, please look at the scales of 4.9c and compare it with 4.9a. The algorithm has found a solution such that the *water-breakthrough* happens at the end of the horizon of the simulation. This is achieved by having a negative coefficient associated with the total production of water in the objective function value. A water-breakthrough is when water starts to enter the well. Once this has happen, the water will keep on flowing into the well. This has to do with the properties of the different fluids. Having the water-breakthrough happen late in the simulation is (often) advantageous because it means that we are producing more or less only hydrocarbons (i.e., oil and gas) the entire considered lifespan. The Figure 4.9d shoes the *water cut*. The water cut is the rate of produced water divided by the rate of total produced fluids. Even though it is "exploding" at the end, which is reasonable as we are having a water-breakthrough, the value is no more than 0.0035 at the end. This is a very low value. A rather normal water cut level to shut down wells is around 0.9.

Figure 4.10 is for the people who know more about petroleum, but all that we are concluding from this figure is that the pressure stays more or less the same. A stable pressure is good.

The number of function evaluations was quite high, and thus another test was per-

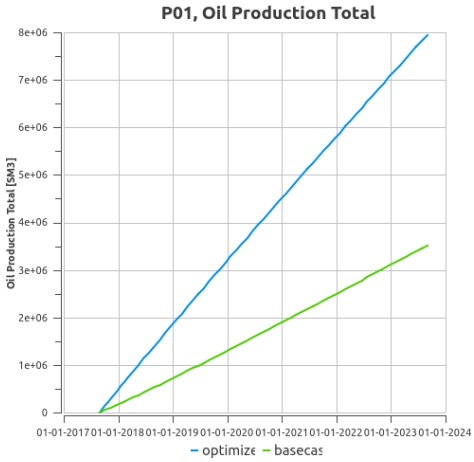


(a) The initial position of the well.

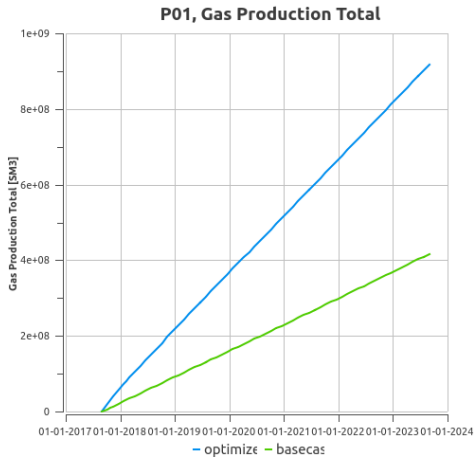


(b) The optimized well.

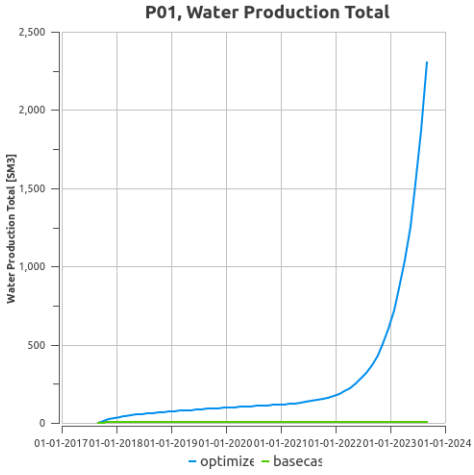
Figure 4.8: The initial and optimized placement of the well. The fault (crack) of the reservoir can be used as reference point for comparison.



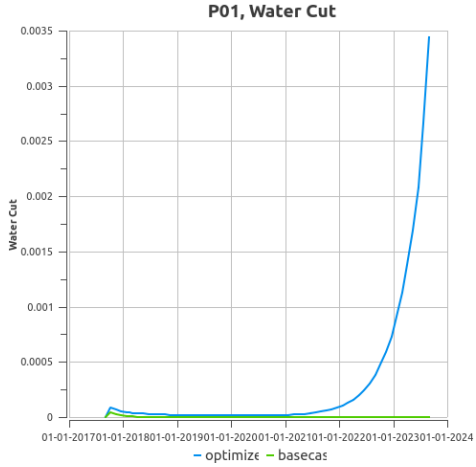
(a) The total oil production in the well.



(b) The total gas production in the well.



(c) The total water production in the well.



(d) The water cut of the well.

Figure 4.9: Key information for the base case and the optimized case.

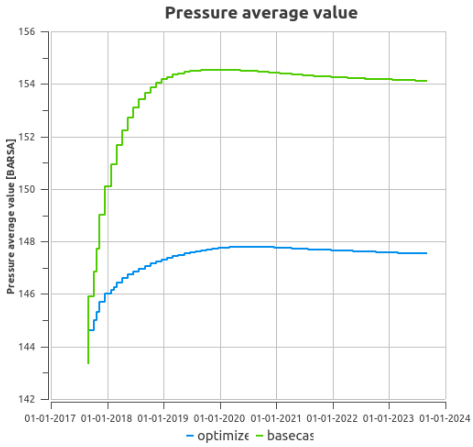


Figure 4.10: The average pressure value. The important thing to note is that it remains stable.

formed. It was assumed that the high number of evaluations was due to a very low final trust-region radius. Only the objective function was inspected. An analysis of the solution was not performed. The initial radius was set to 0.01 and the final to 0.001. Looking at it only from an optimization perspective without evaluating the quality of the solution, the result was a great improvement. Compared to the initial placement of the well, the objective function value increased by about 332 %. The number of function evaluations was only 49 and 20 if parallalization was utilized. This shows that setting the parameters correctly have a crucial impact on the performance of the algorithm. About 1/16 of the function evaluations was used, and the objective function value was about 47 % better! In this last scenario the final trust-region radius was set such that the last steps was around 10 meters for the x-coordinate. This is more reasonable than using one tenth of a meter.

The found position of the well was:

$$[7753.1103, 14369.2628, 1526.7029, 8080.5994, 13991.5888, 1537.34286].$$

Once again, we can see that the z-coordinate only changes slightly.

This concludes the testing of the algorithm. The algorithm seems to work both with and without constraints. Including gradient information into the model-making process might not be advantageous as we lose some of the robustness of the derivative-free model-based trust-region algorithm. However, a faster convergence is often obtained, but to a less optimal local solution. It was also tested on the well placement challenge, and the preliminary results are very promising. In the next chapter this thesis will be summarized.

Chapter 5

Conclusion

This thesis was concerned with an optimization problem where constraints were both present and absent, and where gradients were both available and not. Relevant theory for a derivative-free model-based trust-region algorithm has been present. Extensions to include constraints was also given. Methods to include the gradients into the model-making process was designed by the author. A small modification to the algorithm given in [1] was made. The change was that the gradient of the Lagrangian should be used instead of the gradient of the surrogate model in the scenario when constraints are present.

The suggested algorithm with the possibility of using a gradient enhanced model was implemented. In addition, the support for adding hard (unrelaxable) constraints is implemented. Even though the support for derivative-free constraints (i.e., constraints that must be modeled) are not implemented, it is straight forward to add it considering that the under-determined Lagrange polynomials are available and the subproblem already uses a SQP solver to find a solution.

The different steps of the selected algorithm has been explained and commented. Further, the algorithm has been compared to one that uses the same method to deal with the under-determined quadratic model.

The implementation has been used to produce results for the different scenarios mentioned in the first paragraph of this chapter. The results are that the chosen constraint handling technique is working, but that there will be some limitations on how "difficult" the constraint can be. This is because we need to be able to find points such that the poisedness (the geometry) is still good considering that we are dealing with hard (unrelaxable) constraints. These types of constraints must be considered when the poisedness is being measured and improved. This means that equality constraints can be problematic. If such constraints are desirable, the most reasonable idea would be to add them as soft (relaxable) constraints. Doing so will allow the algorithm that improves the poisedness to choose points that violates the constraints, but the constraints could still be included when

the subproblem is being solved.

Further, tests on how the gradient enhanced model compares to the regular model were performed. The conclusion was that as long as the true function is not convex, the regular model will be most likely to find the better local solution. However, the gradient enhanced model has a tendency to converge faster. The gradient enhanced model was also more fragile to the selection of points compared to the regular one. In addition, the robustness against noise was decreased when gradients were included.

To sum up, the desired functionality has been implemented and tested. The results of including gradients into the model-making process was as forecasted in the theory chapter, and the selected constraint handling technique is working.

5.1 Further work

There are several possible extensions that can be made. The most relevant theory has already been presented and references have been given. If the author should keep on working with this project, he would like to explore the following ideas.

5.1.1 Implementation

There are a lot that can be done with the implementation.

- **The updating scheme**. As mentioned, due to some hidden bugs, the current updating scheme is the simplified and slow one.
- **The constraint handling**. The setup to use a SQP to solve the subproblem is already there. Natural extensions are to add functionality to model derivative-free constraints. In addition, one could add soft (non)linear constraints. Which means that they should be included while solving the subproblem, but ignored while dealing with the poisedness.
- **Cheap improvements**. In the theory chapter there is mentioned the possibility of doing cheap improvements of the geometry. However, if hard (unrelaxable) constraints are included, care must be taken such that those are not violated.

5.1.2 Initialization

The initialization procedure should be explored. As mentioned, the basic idea of perturbing the initial point along the different axes to create an initial set of interpolation points may not be feasible when we have constraints. A simple example of why this initialization procedure must be improved is as follows. Let's say that we have a well and an initial positioning of it. The initial length of the well is same as the maximum allowed length. This is not unreasonable, because this algorithm is supposed to be used by experts who have knowledge in the application area. If the maximum length is D_{max}, and we know that in general a longer well will produce more oil, then the experts will make the well

length around D_{max}. Thus, perturbing the initial point along the different axes will not be allowed by the maximum length constraint.

A possible idea is that instead of looking at one decision variable at the time, one could look at the heel and the toe in a more complex manner. Another idea is to demand more initial guesses from the user such that it will be easier to find more perturb solutions that are feasible.

5.1.3 Optimize the parameters for the application area

There are a lot of parameters in the algorithm. These should be optimized for the application area.

5.1.4 Scaling

A dynamic way of doing the scaling of the decision variables should be added. Scaling can be the difference between a successful and an unsuccessful optimization.

5.1.5 Global optimization of subproblem

An interesting idea is to add a global optimization procedure to solve the subproblem. This might be preferable if the constraints are very complicated and it is hard to find anything but bad local optimums.

5.1.6 Make it a global algorithm

The algorithm is a local solver, but we have seen that it is able to find good local optimums and even the global one. If the global solution is desired, then this algorithm could be extended into a multi-start algorithm. That means that it will be initiated from several different starting points. Thus, the chance of finding the global optimum increases. If this approach is taken, one should try to reuse the already simulated cases whenever possible. Maybe a point that is going to be evaluated is more or less (within some possibly dynamic tolerance) the same as a previous one. Then this point should highly likely be used instead of evaluating a new one.

5.1.7 Gradient enhanced models

I would not recommend to pursue the idea of including the gradients into the model-making process. Unless you know that the true function is convex, or you would not mind getting a possibly worse local solution, the gradient enhanced model will not be favorable. However, if all that you want is an improvement of the initial solution, then this approach is still of interest. But the good property of derivative-free algorithms to be robust against noise is weakened.

Bibliography

[1] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to Derivative-Free Optimization*, volume 8. Siam, 2009.

[2] IEA. *Key World Energy Statistics 2017*. 2017.

[3] Jean-Thomas Camino, Christian Artigues, Laurent Houssin, and Stéphane Mourgues. Linearization of euclidean norm dependent inequalities applied to multibeam satellites design. 2016.

[4] IEA. *Energy Access Outlook 2017*. 2017.

[5] Ministry of Petroleum and Energy Norway (2011). *An industry for the future — Norway's petroleum activities.* `https://www.regjeringen.no/en/dokumenter/meld.-st.-28-20102011/id649699/` (visited on 12/04/2018).

[6] Mathias C Bellout. Joint optimization of well placement and controls for petroleum field development. 2014.

[7] M. J. D. Powell. On the use of quadratic models in unconstrained minimization without derivatives. *Optimization Methods and Software*, 19, 2004.

[8] M. J. D. Powell. Least frobenius norm updating of quadratic models that satisfy interpolation conditions. *Math. Programming*, 100, May 2004.

[9] M. J. D. Powell. On updating the inverse of a KKT matrix. *Numerical Linear Algebra and Optimization*, January 2004.

[10] M. J. D. Powell. The NEWOUA software for unconstrained optimization without derivatives. *Large-Scale Nonlinear Optimization*, 83, November 2006.

[11] M. J. D. Powell. The BOBYQA algorithm for bounded constrained optimization without derivatives. August 2009.

[12] A. Conn, N. I. M. Gould, and P. L. Toint. *Trust Region Methods*. Society for Industrial and Applied Mathematics, 2000.

[13] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. A family of trust-region-based algorithms for unconstrained minimization with strong global convergence properties. *SIAM Journal on Numerical Analysis*, 22, 1985.

[14] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. Approximate solution of the trust region problem by minimization over two-dimensional subspaces. *Math. Programming*, 40, 1988.

[15] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

[16] T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, 20, 1983.

[17] Y. Yuan. On the truncated conjugate gradient method. *Math. Programming*, 87, 2000.

[18] N. I. M Gould, S. Lucidi, M. Roma, and P. L Toint. Solving the trust-region subproblem using the lanczos method. *SIAM Journal on Optimization*, 9, 1999.

[19] L. H. Zhang, C. Shen, and R.C. Li. On the generalized lanczos trust-region method. *SIAM Journal on Optimization*, 27, 2017.

[20] Stefan Martin Wild. *Derivative-free optimization algorithms for computationally expensive functions*. Cornell University, 2009.

[21] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[22] Mark A Abramson, Charles Audet, and John E Dennis. Generalized pattern searches with derivative information. *Mathematical Programming*, 100(1):3–25, 2004.

[23] Michael JD Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*, pages 51–67. Springer, 1994.

[24] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[25] David H Ackley. The model. In *A Connectionist Machine for Genetic Hillclimbing*. Springer, 1987.

[26] S. M. Goldfeld, R. E. Quandt, and H.F. Trotter. Maximization by quadratic hillclimbing. *Econometrica*, 34, 1966.

[27] E Philip, Walter Murray, and Michael A Saunders. User's guide for snopt version 7: Software for large-scale nonlinear programming. 2008.

[28] Philip E Gill, Walter Murray, and Michael A Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM review*, 47(1):99–131, 2005.

[29] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

Appendix A - Results of testing the algorithm

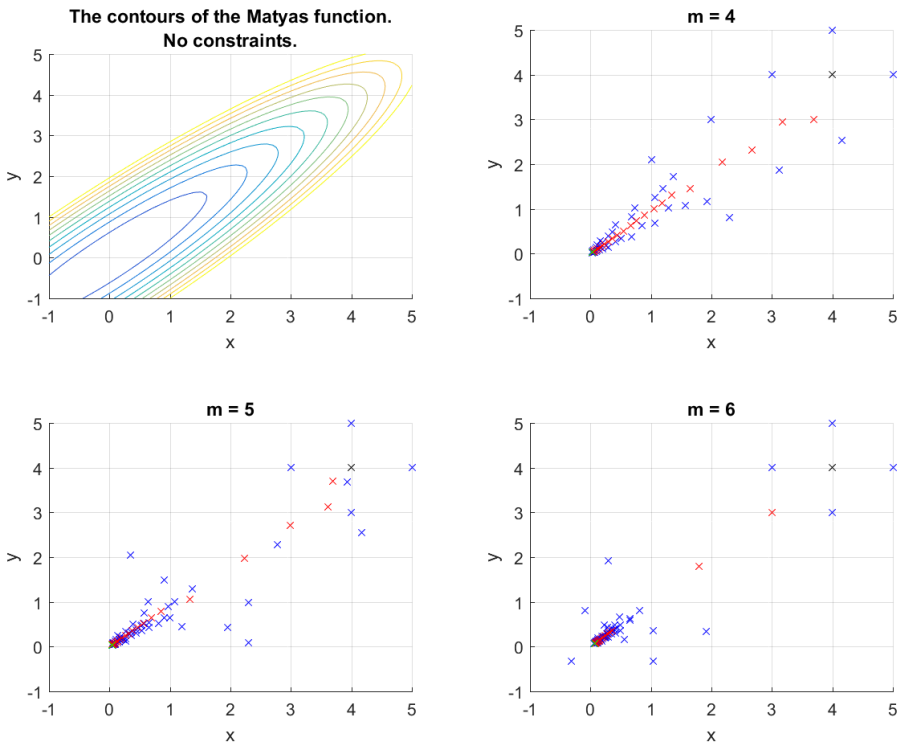


Figure A1: The points evaluated during optimizations runs of the Matyas function without constraints. See Table A1 for more information.

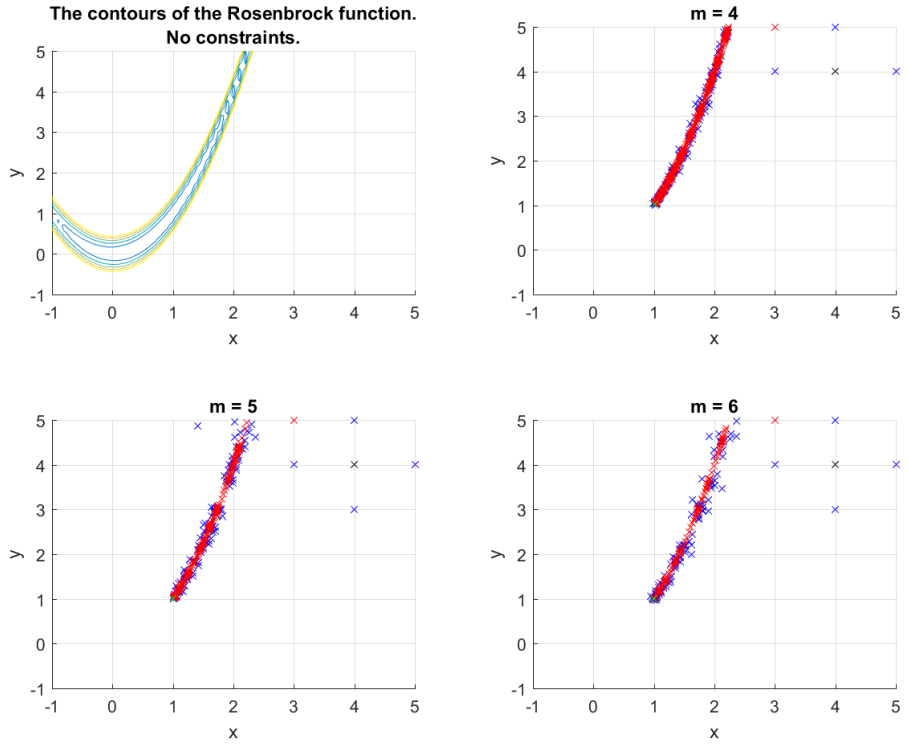


Figure A2: The points evaluated during optimizations runs of the Rosenbrock function without constraints. See Table A6 for more information.

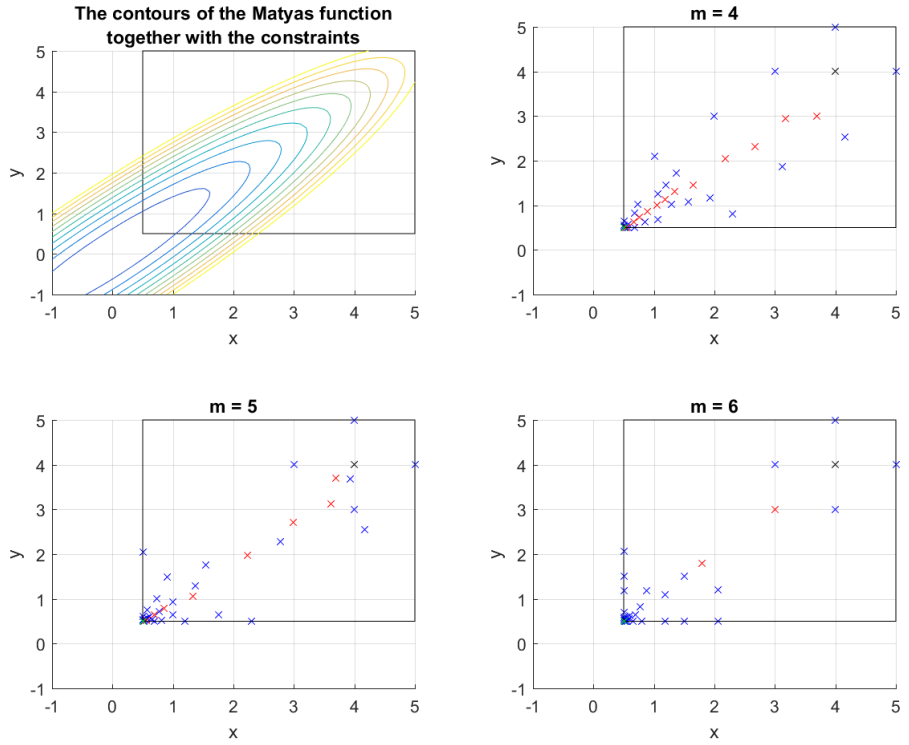


Figure A3: The points evaluated during optimizations runs of the Matyas function with bounds. See Table A2 for more information.

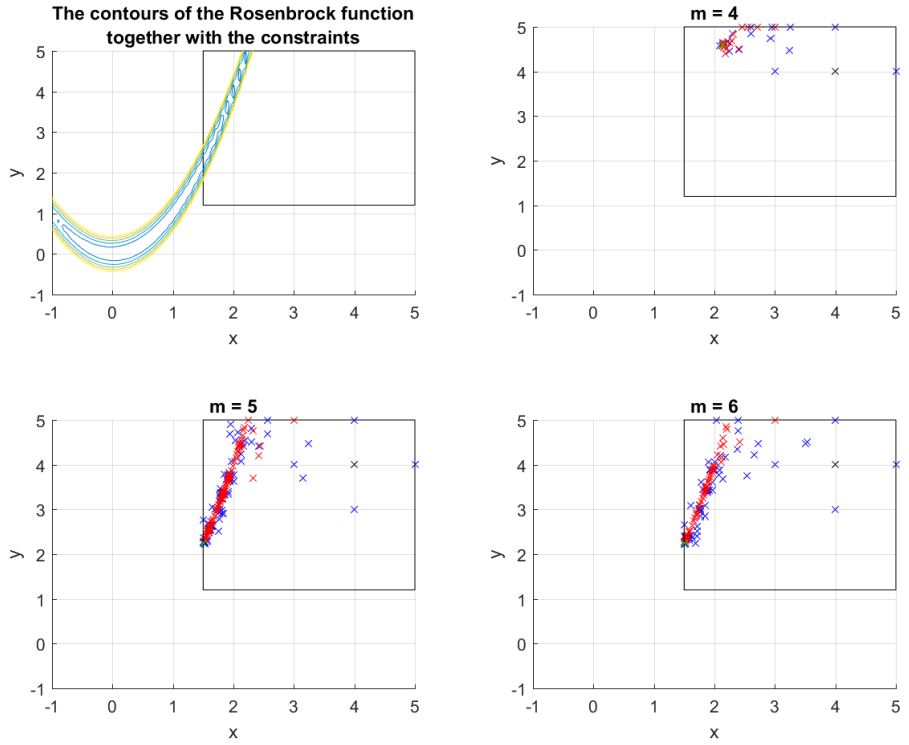


Figure A4: The points evaluated during optimizations runs of the Rosenbrock function with bounds. See Table A7 for more information.

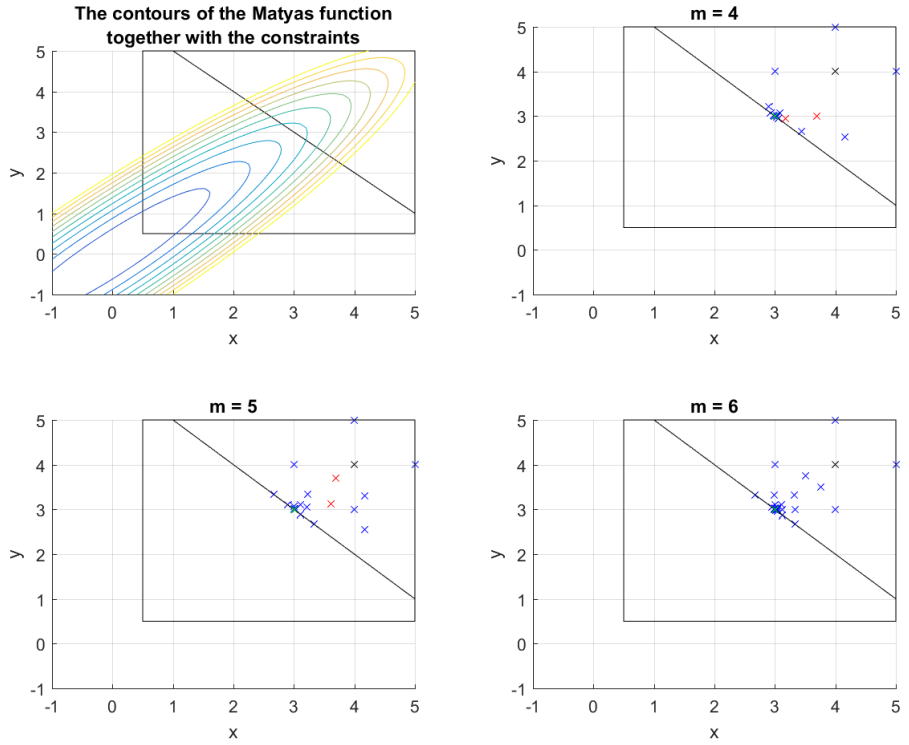


Figure A5: The points evaluated during optimizations runs of the Matyas function with bounds and linear constraint. See Table A3 for more information.

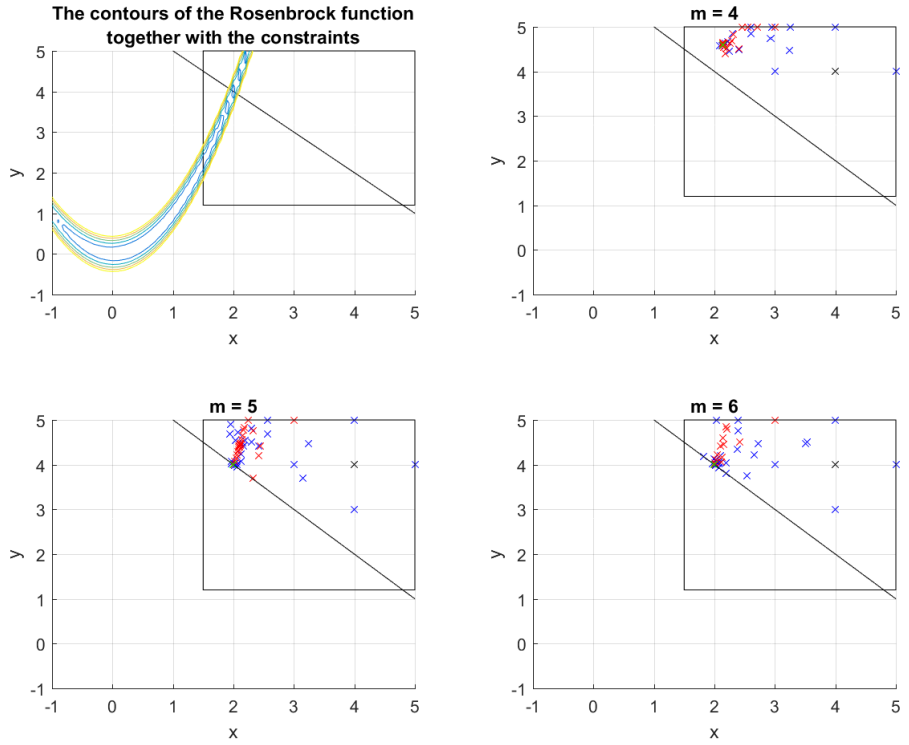


Figure A6: The points evaluated during optimizations runs of the Rosenbrock function with bounds and linear constraint. See Table A8 for more information.

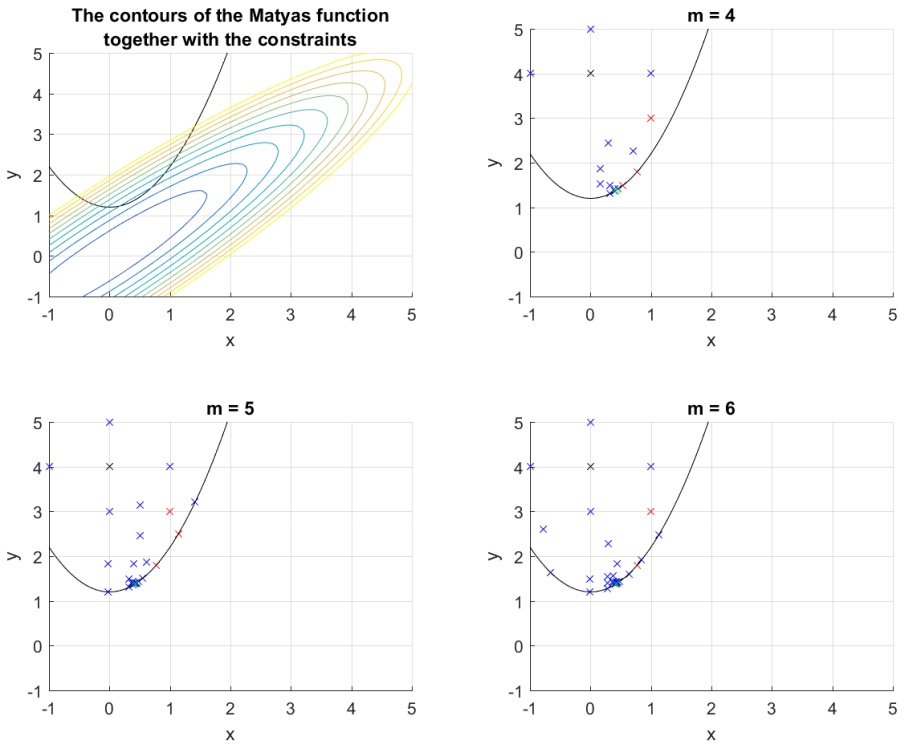


Figure A7: The points evaluated during optimizations runs of the Matyas function with a nonlinear constraint. See Table A4 for more information.

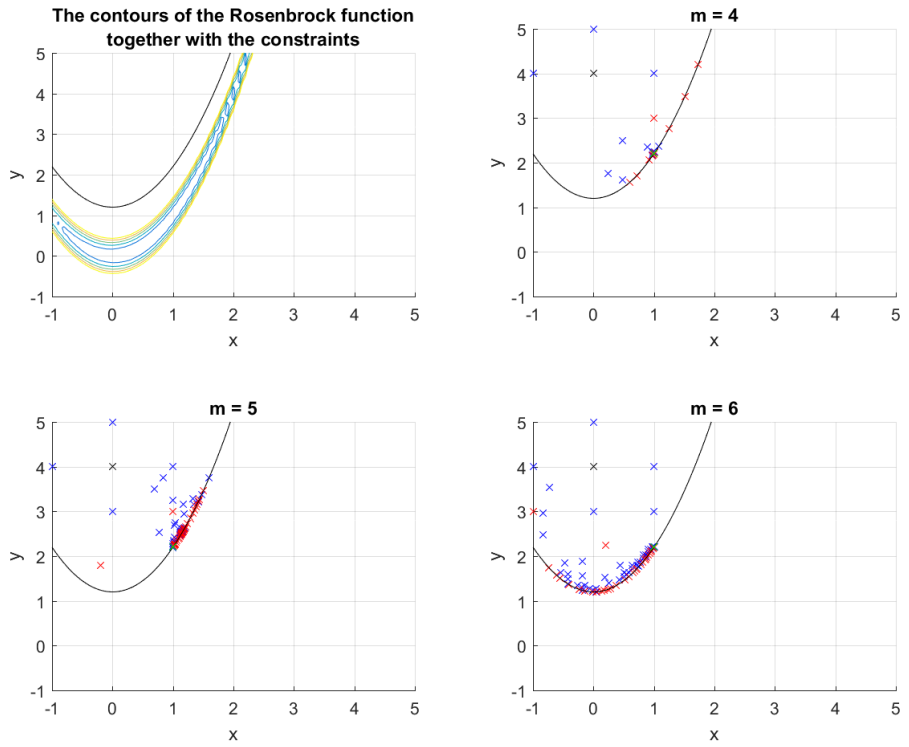


Figure A8: The points evaluated during optimizations runs of the Rosenbrock function with a non-linear constraint. See Table A9 for more information.

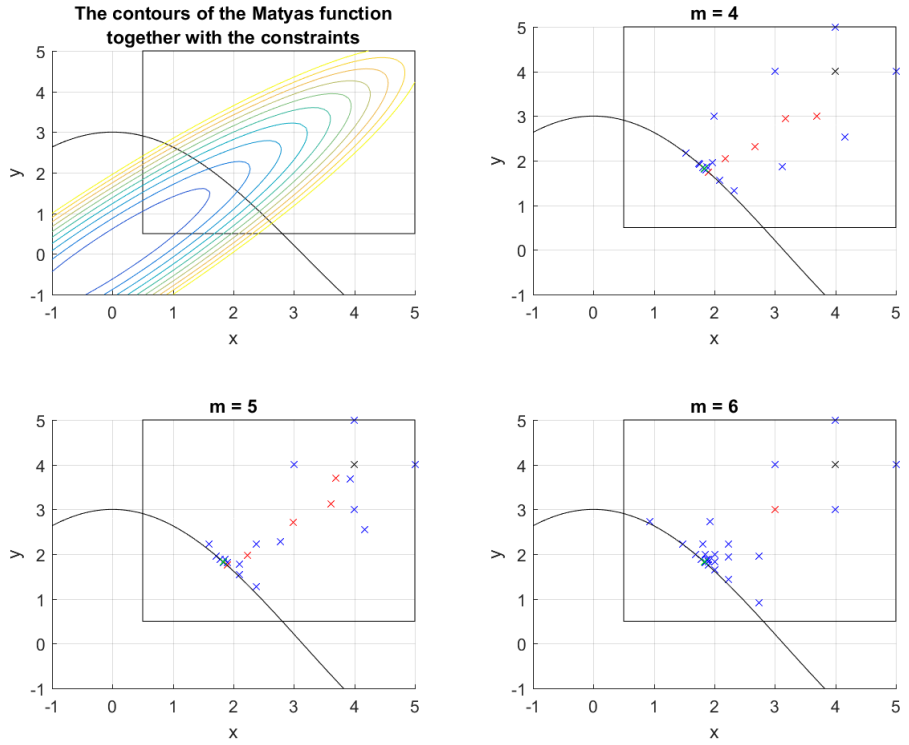


Figure A9: The points evaluated during optimizations runs of the Matyas function with a nonlinear nonconvex constraint and bounds. See Table A5 for more information.

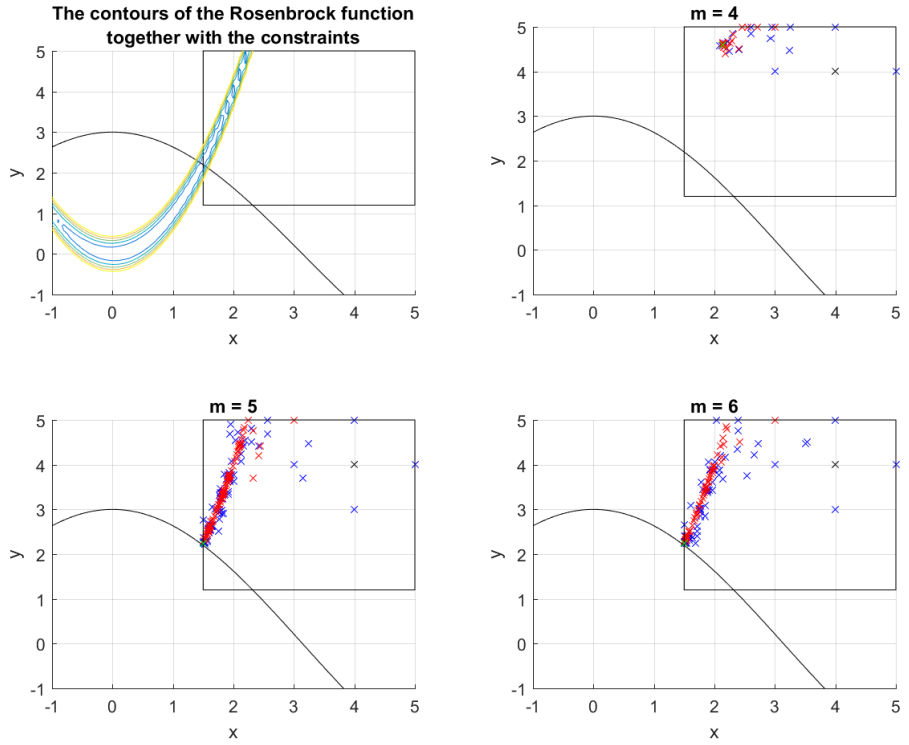


Figure A10: The points evaluated during optimizations runs of the Rosenbrock function with a nonlinear nonconvex constraint and bounds. See Table A10 for more information.

Table A1: The Matyas function. No constraints.

	m=4	**m=5**	**m=6**
n_e	78	99	117
n_p	58	58	64
(x^*, y^*)	(0.02735, 0.02742)	(0.03893, 0.03893)	(0.06292, 0.06292)
$f^*(x^*, y^*)$	0.00003	0.00006	0.00016

Table A2: The Matyas function. Bounds.

	m=4	**m=5**	**m=6**
n_e	49	53	52
n_p	36	34	25
(x^*, y^*)	(0.50000, 0.50000)	(0.50000, 0.50000)	(0.50000, 0.50000)
$f^*(x^*, y^*)$	0.01000	0.01000	0.01000

Table A3: The Matyas function. Bounds and linear constraint.

	m=4	**m=5**	**m=6**
n_e	23	29	38
n_p	14	14	10
(x^*, y^*)	(3.00000, 3.00000)	(3.00047, 2.99953)	(3.00000, 3.00000)
$f^*(x^*, y^*)$	0.36000	0.36000	0.36000

Table A4: The Matyas function. Nonlinear constraint.

	m=4	**m=5**	**m=6**
n_e	27	36	44
n_p	13	17	18
(x^*, y^*)	(0.42939, 1.38437)	(0.42939, 1.38438)	(0.42939, 1.38438)
$f^*(x^*, y^*)$	0.26090	0.26090	0.26090

Table A5: The Matyas function. Bounds and nonlinear nonconvex constraint

	m=4	**m=5**	**m=6**
n_e	28	34	38
n_p	20	18	12
(x^*, y^*)	(1.83541, 1.82294)	(1.83513, 1.82326)	(1.83555, 1.82277)
$f^*(x^*, y^*)$	0.13387	0.13387	0.13387

Table A6: The Rosenbrock function. No constraints.

	m=4	**m=5**	**m=6**
n_e	842	469	393
n_p	725	363	274
(x^*, y^*)	(1.00966, 1.01946)	(1.00296, 1.00611)	(0.99990, 0.99981)
$f^*(x^*, y^*)$	0.00009	0.00001	0.00000

Table A7: The Rosenbrock function. Bounds.

	m=4	**m=5**	**m=6**
n_e	47	201	149
n_p	38	148	100
(x^*, y^*)	(2.14443, 4.60090)	(1.49961, 2.24891)	(1.50000, 2.25004)
$f^*(x^*, y^*)$	1.31026	0.24961	0.25000

Table A8: The Rosenbrock function. Bounds and linear constraint.

	m=4	**m=5**	**m=6**
n_e	47	65	55
n_p	38	49	32
(x^*, y^*)	(2.14443, 4.60090)	(1.99960, 4.00040)	(1.99960, 4.00040)
$f^*(x^*, y^*)$	1.31026	0.99960	0.99960

Table A9: The Rosenbrock function. Nonlinear constraint.

	m=4	**m=5**	**m=6**
n_e	41	106	105
n_p	33	75	65
(x^*, y^*)	(0.99974, 2.19948)	(1.00000, 2.19999)	(1.00094, 2.20188)
$f^*(x^*, y^*)$	144.00000	144.00000	143.99999

Table A10: The Rosenbrock function. Bounds and nonlinear nonconvex function.

	m=4	**m=5**	**m=6**
n_e	47	200	137
n_p	38	146	93
(x^*, y^*)	(2.14443, 4.60090)	(1.50000, 2.25000)	(1.50000, 2.25000)
$f^*(x^*, y^*)$	0.61289	0.25000	0.25000

Appendix B - The implementation

EigenUtil.h

```
//
// Created by joakim on 16.04.18.
//
#ifndef FIELDOPT_EIGEN_UTIL_H
#define FIELDOPT_EIGEN_UTIL_H

#include <Eigen/Core>
#include <Eigen/Dense>


inline void eigen_tail(Eigen::VectorXd &lhs, const Eigen::VectorXd &rhs,
 ↪  int a) {
  int d = lhs.rows() - a;
  for (int i = 0; i < a; ++i) {
    lhs[i + d] = rhs[i];
  }
}

inline void eigen_head(Eigen::VectorXd &lhs, const Eigen::VectorXd &rhs,
 ↪  int a) {
  for (int i = 0; i < a; ++i) {
    lhs[i] = rhs[i];
  }
}

inline void eigen_col(Eigen::MatrixXd &lhs, const Eigen::VectorXd &rhs,
 ↪  int a) {
  for (int i = 0; i < lhs.rows(); ++i) {
    lhs(i, a) = rhs[i];
  }
}

inline void eigen_row(Eigen::MatrixXd &lhs, const  Eigen::VectorXd &rhs,
 ↪  int a) {
  for (int i = 0; i < lhs.cols(); ++i) {
    lhs(a, i) = rhs[i];
  }
}

inline void eigen_block(Eigen::MatrixXd &lhs, const Eigen::MatrixXd &rhs,
 ↪  int startRow, int startCol) {
```

```
  for (int i = 0; i < rhs.rows(); ++i) {
    for (int j = 0; j < rhs.cols(); ++j) {
      lhs(startRow + i, startCol + j) = rhs(i, j);
    }
  }
}


#endif //FIELDOPT_EIGEN_UTIL_H
```

DFO_Model.h

```
#ifndef FIELDOPT_DFO_MODEL_H
#define FIELDOPT_DFO_MODEL_H

#include <iostream>
#include <Eigen/Dense>
#include <random>
#include <math.h>
#include <Settings/optimizer.h>
#include <Subproblem.h>
#include "Subproblem.h"
#include "EigenUtil.h"
#include "GradientEnhancedModel.h"
/* References
This implementation is based upon the following papers and book, and I would recommend anyone who
is trying to understand the code to actively use those references.
[1] The BOBYQA algorithm for bound constrained optimization without derivatives by M.J.D. Powell.
[2] The NEWUOA software for unconstrained optimization without derivatives by M.J.D. Powell.
[3] Introduction to Derivative-Free Optimization by Andrew R. Conn, Katya Scheinberg and Luis N. Vicente.
[4] Least Frobenius norm updating of quadratic models that satisfy interpolation conditions by M.J.D. Powell.

Only the parts about model improvement (i.e., finding upper/lower/high value of the Lagrange polynomials,
are based upon [3]. Almost everything else is based upon [2] (practical approach) and [4] (more theoretical view).
*/
namespace Optimization {
namespace Optimizers {

class DFO_Model {

 private:

  Eigen::VectorXd copyOfStartingPoint;
  Eigen::VectorXi mapNormalToFieldopt;

  Eigen::VectorXd lagrangeMultipliers;
  const int normType;
  double lagabsvalMin = 0.5; // works ok: 0.001

```

```
38  Settings::Optimizer *settings_;
39  GradientEnhancedModel enhancedModel;
40  Subproblem subproblem;
41  unsigned int m; // Number of interpolation points used to create the
     ↪  model. Does not change.
42  unsigned int n; // Number of decision variables in your model.
43  unsigned int ng; // Number of decision variables WITH gradients.
44  double rho; // Trust-region radius.
45  double lambda; // The required poisedness of the set of interpolation
     ↪  points.
46  double r;
47
48  Eigen::MatrixXd Winv;
49  Eigen::MatrixXd W;
50
51  Eigen::VectorXd y0; // The point which the model is ceneterd around.
52
53  Eigen::MatrixXd
54      Y;  // Container for the interpolation points. The interpolation
         ↪  point i is given in the following way:
55  // yi = y0 + Y.col(i); i.e. Y contains the displacements away from y0.
56
57  Eigen::MatrixXd derivatives;
58  Eigen::VectorXd derivativeAtCenterpoint;
59
60  Eigen::VectorXd fvals; // Holds the function evaluations for the
     ↪  interpolation points.
61
62  Eigen::VectorXd bestPoint; // Displacement of the optimal point from
     ↪  y0. Absolute point: y0 + bestPoint;
63  int
64      bestPointIndex; // This assumes that the point is in the
         ↪  interpolation set (not sure if this will be discarded in
         ↪  future work)
65
66  Eigen::VectorXd bestPointAllTime;
67  double bestPointAllTimeFunctionValue;
68
69  //These 4 variables are only used in the case m>2*n+1. How to calculate
     ↪  the p's and q's are well explained in [1], for sigma see [2].
70  //The i's are used to find the corresponding interpolation points and
     ↪  function values in Y and Fvals, respectively.
71  //The qs, qs and is come in a set of 3, such that the same index of
     ↪  each vector belong together.
72  Eigen::VectorXi sigmas;
73  Eigen::VectorXi ps;
74  Eigen::VectorXi qs;
75  Eigen::VectorXi is;
76
77  // The inverse KKT matrix, H. See [2]
78  Eigen::MatrixXd Xi;
79  Eigen::MatrixXd Upsilon;
80  Eigen::MatrixXd Z;
81  Eigen::DiagonalMatrix<double, Eigen::Dynamic>
82      S; //Should have been a integer matrix, but Eigen doesn't support
         ↪  diagonal int matrices.
83
```

```
84
85      // Containers for the 2nd order model.
86      Eigen::VectorXd gradient;
87      Eigen::VectorXd centerPoint;
88      Eigen::MatrixXd hessian;
89      double constant = 0;
90
91      Eigen::MatrixXd Gamma;
92      Eigen::VectorXd gammas;
93
94      bool modelInitialized;
95      bool initialInterpolationPointsFound;
96
97      int isModelCFL = -1; // -1: Don't know.     0: No.    1: Yes.
98
99      /**
100     Checks if a value is almost zero.
101
102     @value the value to be checked.
103     @zeroLimit the highest value that still is counted as zero.
104     @return true / false
105     */
106     bool isApproxZero(double value, double zeroLimit);
107
108     /**
109     The classic kronecker-delta function.
110
111     Compares if two integers are equal or not.
112
113     @param[in] i integer 1
114     @param[in] j integer 2
115     @return 1 if i==j and zero otherwise.
116     */
117     int kroneckerDelta(int i, int j);
118
119     /**
120     Returns the sign of the value.
121
122     0 is counted as positive (+1).
123
124     @param[in] the value to be checked.
125     @return the sign of the value.
126     */
127     int sign(double value);
128
129
130
131     bool cmp(Eigen::VectorXd a, Eigen::VectorXd b);
132
133   public:
134     //EIGEN_MAKE_ALIGNED_OPERATOR_NEW
135
136     enum UpdateReason {
137       IMPROVE_POISEDNESS = -1,
138       INCLUDE_NEW_OPTIMUM = -2,
139       INCLUDE_NEW_POINT = -3,
140       FORCED_IMPROVE_MODEL = -4
```

```
141   };
142
143   bool isInitialInterpolationPointsFound() {
144     return initialInterpolationPointsFound;
145   }
146
147   bool isModelInitialized() {
148     return modelInitialized;
149   }
150
151   /**
152   Constructor for the class.
153
154   A naming convenvention:
155   model = this entire class.
156   quadratic model = just the quadratic model.
157   Inverse KKT matrix = H
158
159   @param[in] m number of interpolation points.
160   @param[in] n number of decision variables.
161   @param[in] y0 the center point of the model.
162   @param[in] rhoBeg the initial trust-region radius.
163   @param[in] lambda the required poisedness of the interpolation set
      ↪  (lambda > 1)
164   */
165   DFO_Model(unsigned int m,
166             unsigned int n,
167             unsigned int ng,
168             Eigen::VectorXd y0,
169             double rhoBeg,
170             double lambda,
171             double weight_objective_minimum_change,
172             QList<double> weights_derivatives,
173             Settings::Optimizer *settings);
174
175   DFO_Model() : normType(0) {};
176
177   Eigen::VectorXd
      ↪  ScaleVariablesFromApplicationToAlgorithm(Eigen::VectorXd point);
178
179   /**
180   Finds the first set of interpolation points.
181
182   Finds the first set of interpolation points based
183   upon the initial center point (y0) and the trust-region radius.
184
185   */
186   Eigen::MatrixXd findFirstSetOfInterpolationPoints();
187
188   /**
189   Finds the last set of interpolation points.
190
191   If m > 2n+1, then this function must also be ran. Before it is ran,
192   make sure that the local variable fvals is filled up with the function
193   evaluations for the first set of interpolation points.
194
195   */
```

```
196   Eigen::MatrixXd findLastSetOfInterpolationPoints();
197
198   /**
199   Initializes the model.
200
201   Both findFirstSetOfInterpolationPoints() and
      ↪  findLastSetOfInterpolationPoints() (if m > 2n+1)
202   must be called before this function.
203   */
204   void initializeModel();
205
206   /**
207   Updates the model with a new point.
208
209   Updates the model with a new point, the reason for updating must be
      ↪  provided.
210   Some changes must be done, and this function is NOT done!
211   Cannot be used when INCLUDE_NEW_OPTIMUM is the reason.
212
213   @param[in] yNew is the  is the displacement of the new point from the
      ↪  current center point(y0).
214   @param[in] fvalNew is the function evaluation corresponding to yNew.
215   @param[in] t-1 is the index of the point that is going to be replaced
      ↪  by yNew in Y.
216   @param[in] updateReason is either IMPROVE_POISEDNESS or
      ↪  INCLUDE_NEW_OPTIMUM.
217   */
218   void update(Eigen::VectorXd yNew,
219               double fvalNew,
220               Eigen::VectorXd gradient,
221               unsigned int t,
222               UpdateReason updateReason);
223
224   void update(Eigen::MatrixXd yNews,
225               Eigen::VectorXd fvalNews,
226               Eigen::MatrixXd gradients,
227               Eigen::VectorXi indicies,
228               int numberOfPoints,
229               UpdateReason updateReason);
230
231   /**
232   Evaluates the current quadratic model at point.
233
234   @param[in] point is the displacement from current center point (y0).
235   @return the value of the model at point.
236   */
237   double evaluateQuadraticModel(Eigen::VectorXd point);
238
239   /**
240   This is a bad way of accessing and updating the function evaluations.
241
242   It will be replaced by other functions later, when I have decided how
243   the interface should be.
244   @return a reference to the fvals
245   */
246   Eigen::VectorXd *getFvalsReference() {
247     return &(this->fvals);
```

```
248   }
249
250   /**
251   This is a bad way of accessing Y outside of the class.
252
253   It will be replaced by other functions later, when I have decided how
254   the interface should be.
255   @return a reference to the Y
256   */
257   Eigen::MatrixXd *getYReference() {
258     return &(this->Y);
259   }
260
261   Eigen::MatrixXd *getDerivativeReference() {
262     return &(this->derivatives);
263   }
264
265   /**
266   Returns the center point of the quadratic model
267
268   @return the center point of the quadratic model
269   */
270   Eigen::VectorXd getCenterPoint() {
271     return y0;
272   }
273
274
275   void findWorstPointInInterpolationSet(Eigen::VectorXd &dNew, int
      ↪  &indexOfWorstPoint);
276   void calculateAMatrix(Eigen::MatrixXd &A, Eigen::MatrixXd &Ycopy);
277
278
279   Eigen::VectorXd FindLocalOptimumOfAbsoluteLagrangePolynomial(int t);
280
281   /**
282   Finds the point that is best to replace with the new one.
283
284   @yNew is the point that we want to add to the model, where yNew is
    ↪  given as the displacement of the current center point.
285   @return index of the point that is best to replace.
286   */
287   int findPointToReplaceWithNewOptimum(Eigen::VectorXd yNew);
288
289   /**
290   Returns the index of the best point.
291
292   Note: this will only be valid as long as the best yet found
293   point is never removed from the interpolation set.
294
295   @return the index of the best point.
296   */
297   int getBestPointIndex();
298
299   void SetFunctionValue(int t, double value);
300   void SetFunctionValueAndDerivatives(int t, double value,
      ↪  Eigen::VectorXd grad);
301   void SetTrustRegionRadiusForSubproblem(double radius);
```

```
302   bool FindReplacementForPointsOutsideRadius(double radius,
      ↪  Eigen::MatrixXd &newPoints, Eigen::VectorXi &newIndices);
303   double GetFunctionValue(int t) {
304     return fvals[t - 1];
305   }
306
307   double GetTrustRegionRadius() {
308     return rho;
309   }
310
311   void SetTrustRegionRadius(double radius) {
312     rho = radius;
313   }
314
315   void SetRequiredPoisedness(double lambda) {
316     this->lambda = lambda;
317   }
318   void SetInitialStartPoint(Eigen::VectorXd startPoint) {
319     y0 = startPoint;
320   }
321   void SetNumberOfVariables(int n) {
322     this->n = n;
323   }
324   void SetNumberOfInterpolationPoints(int m) {
325     this->m = m;
326   }
327
328   Eigen::VectorXd GetGradient() {
329     return gradient;
330   }
331   Eigen::VectorXd GetGradientAtPoint(Eigen::VectorXd point) {
332     enhancedModel.ComputeModel(Y, derivatives, derivativeAtCenterpoint,
      ↪  fvals, y0, bestPoint, rho, r, ng);
333     double e_c = 0;
334     Eigen::VectorXd e_g(n);
335     Eigen::MatrixXd e_h(n, n);
336     enhancedModel.GetModel(e_c, e_g, e_h);
337     return e_g + e_h * point;
338
339   }
340   Eigen::VectorXd GetPoint(int t) {
341     return Y.col(t - 1);
342   }
343   Eigen::VectorXd GetBestPoint() {
344     return Y.col(bestPointIndex - 1);
345   }
346   double GetBestFunctionValueAllTime() {
347     return bestPointAllTimeFunctionValue;
348   }
349   Eigen::VectorXd GetBestPointAllTime() {
350     return bestPointAllTime;
351   }
352   Eigen::VectorXd FindLocalOptimum();
353   double findLargestDistanceBetweenPointsAndOptimum();
354   double ComputeLagrangePolynomial(int t, Eigen::VectorXd point);
355   double PrintLagrangePolynomial(int t);
356   Eigen::VectorXi GetInterpolationPointsSortedByDistanceFromBestPoint();
```

```
357   bool FindPointToReplaceWithPointOutsideScaledTrustRegion(int t,
       ↪  Eigen::VectorXd &dNew);
358   void wtf(Eigen::VectorXd &da) {
359     return;
360   }
361   int isPointAcceptable(Eigen::VectorXd point);
362   int GetNumberOfPointsOutsideRadius(double radius);
363   double norm(Eigen::VectorXd a);
364   Eigen::MatrixXd calculateWExplicitly();
365   void UpdateOptimum();
366   bool isPoised(VectorXd &dNew, int &indexOfPointToBeReplaced, double
       ↪  radius);
367   void modelImprovementStep(VectorXd &dNew, int
       ↪  &indexOfPointToBeReplaced);
368   void Converged(int iterations,
369                  int number_of_tiny_improvements,
370                  int number_of_function_calls,
371                  int number_of_parallell_function_calls);
372   void isLagrangePoly(); // prints the (hopefully) kronecker-delta
       ↪  property. used for debugging.
373   void createLagrangePolynomial(int t, double &c, VectorXd &grad,
       ↪  MatrixXd &hess);
374   void updateQuadraticModelNew(Eigen::VectorXd yNew, double fvalNew,
       ↪  unsigned int t);
375   void shiftCenterPointOfQuadraticModelNew(Eigen::VectorXd s);
376   void createW();
377   int IsModelCFL() {
378     return isModelCFL;
379   }
380   Eigen::VectorXd GetLagrangianGradient(Eigen::VectorXd point);
381   void initLagrangeMultipliers(int a) {
382     lagrangeMultipliers = Eigen::VectorXd::Zero(a);
383   }
384   bool ModelImprovementAlgorithm(double radius, Eigen::MatrixXd
       ↪  &newPoints, Eigen::VectorXi &newIndices);
385   void calculateLagrangeMultipliers();
386   //These two functions are only used to create an example to show how
       ↪  the model improvement algorithm works
387   void setInitialy0(Eigen::VectorXd);
388   void findVariableMeaning(Eigen::VectorXd realvars, Eigen::VectorXd
       ↪  scaling);
389   void PrintSortedBestPoint(Eigen::VectorXd scaling);
390
391 };
392 }
393 }
394
395 #endif //FIELDOPT_DFO_MODEL_H
```

DFO_Model.cpp

```
1  #include "DFO_Model.h"
2  namespace Optimization {
3  namespace Optimizers {
4
5  bool DFO_Model::cmp(Eigen::VectorXd a, Eigen::VectorXd b) {
```

```
  return (norm(a.topRows(a.rows() - 1))) > (norm(b.topRows(b.rows() -
  ↪ 1)));
}

static int NormType = 0;

double norm2(Eigen::VectorXd a) {
  if (NormType == 0) {
    return (a).lpNorm<Infinity>();
  } else if (NormType == 2) {
    return (a).norm();
  }
}

bool cmp2(Eigen::VectorXd a, Eigen::VectorXd b) {
  return (norm2(a.topRows(a.rows() - 1))) > (norm2(b.topRows(b.rows() -
  ↪ 1)));
}

bool DFO_Model::isApproxZero(double value, double zeroLimit) {
  if (std::abs(value) <= zeroLimit)
    return true;
  return false;
}

int DFO_Model::kroneckerDelta(int i, int j) {
  if (i == j)
    return 1;
  return 0;
}

int DFO_Model::sign(double value) {
  if (value >= 0) {
    return 1;
  }
  return -1;
}



void DFO_Model::updateQuadraticModelNew(Eigen::VectorXd yNew, double
↪ fvalNew, unsigned int t) {
  double valModelOld = evaluateQuadraticModel(yNew);
  double diff = fvalNew - valModelOld;
  double c;
  Eigen::VectorXd grad(n);
  Eigen::MatrixXd hess(n, n);
  createLagrangePolynomial(t, c, grad, hess);

  constant += diff * c;
  gradient += diff * grad;
  hessian += diff * hess;

}


DFO_Model::DFO_Model(unsigned int m,
```

```
                    unsigned int n,
                    unsigned int ng,
                    Eigen::VectorXd y0,
                    double rhoBeg,
                    double lambda,
                    double weight_objective_minimum_change,
                    QList<double> weights_derivatives,
                    Settings::Optimizer *settings)
    : subproblem(settings),
      enhancedModel(n, m, ng, weights_derivatives,
       ↪  weight_objective_minimum_change),
      normType(settings->parameters().norm_type) {
  NormType = settings->parameters().norm_type;
  this->m = m;
  this->n = n;
  this->ng = ng;
  Eigen::MatrixXd wtf12(10, 10);
  Eigen::MatrixXd wtf11(20, 20);
  Eigen::MatrixXd wtf(m + n + 1, m + n + 1);
  Eigen::MatrixXd wtf2;
  wtf2.resize(m + n + 1, m + n + 1);
  this->Winv.resize(m + n + 1, m + n + 1);
  this->Winv = Eigen::MatrixXd::Zero(m + n + 1, m + n + 1);
  this->W = Eigen::MatrixXd::Zero(m + n + 1, m + n + 1);
  this->y0 = Eigen::VectorXd::Zero(n);

  this->r = settings->parameters().r;
  int j = 0;

  copyOfStartingPoint = Eigen::VectorXd::Zero(n);
  for (auto i = settings->parameters().starting_point.begin(); i !=
   ↪  settings->parameters().starting_point.end(); ++i) {
    this->y0[j] = *i;
    this->copyOfStartingPoint[j] = *i;
    j++;
    if (j >= n) {
      break;
    }
  }

  lagabsvalMin = settings->parameters().min_lagrange_abs_val;

  this->rho = rhoBeg;
  this->lambda = lambda;

  this->bestPoint = Eigen::VectorXd::Zero(n);
  this->bestPointAllTime = Eigen::VectorXd::Zero(n);
  this->bestPointAllTimeFunctionValue =
   ↪  std::numeric_limits<double>::max();
  this->Y = Eigen::MatrixXd::Zero(n, m);
  this->derivatives = Eigen::MatrixXd(ng, m);
  this->derivativeAtCenterpoint = Eigen::VectorXd(ng);
  this->fvals = Eigen::VectorXd(m);
  if (m >= n + 2) {
    this->Xi = Eigen::MatrixXd::Zero(n + 1, m);
    this->Upsilon = Eigen::MatrixXd::Zero(n + 1, n + 1);
    this->Z = Eigen::MatrixXd::Zero(m, m - n - 1);
```

```
114       this->S = Eigen::DiagonalMatrix<double, Eigen::Dynamic>(m - n - 1);
115       this->S.diagonal().setOnes();
116     }
117     if (m > 2 * n + 1) {
118       this->sigmas = Eigen::VectorXi(n);
119       this->ps = Eigen::VectorXi(m - 2 * n - 1);
120       this->qs = Eigen::VectorXi(m - 2 * n - 1);
121       this->is = Eigen::VectorXi(m - 2 * n - 1);
122     }
123
124     this->gradient = Eigen::VectorXd::Zero(n);
125     this->centerPoint = Eigen::VectorXd::Zero(n);
126     this->hessian = Eigen::MatrixXd::Zero(n, n);
127     this->Gamma = Eigen::MatrixXd::Zero(n, n);
128     this->gammas = Eigen::VectorXd::Zero(m);
129     this->initialInterpolationPointsFound = false;
130     this->modelInitialized = false;
131     this->settings_ = settings;
132     this->isModelCFL = -1;
133
134   }
135
136   Eigen::MatrixXd DFO_Model::findFirstSetOfInterpolationPoints() {
137     int numberOfPointsFound = 0;
138     if (m < n + 2) {
139       for (int i = 1; i <= n; ++i) {
140         if (i <= (m - 1)) {
141           Y(i - 1, i) += rho;
142         }
143       }
144       if (m >= n) {
145         for (int i = 1; i < m - n; ++i) {
146           Y(i - 1, i + n) -= rho;
147         }
148       }
149       numberOfPointsFound = m;
150       initialInterpolationPointsFound = true;
151     } else if (m >= 2 * n + 1 && m <= (n + 1) * (n + 2) * 0.5) {
152       for (int i = 1; i <= n; ++i) {
153         Y(i - 1, i) += rho;
154         Y(i - 1, i + n) -= rho;
155       }
156       numberOfPointsFound = 2 * n + 1;
157     } else if (m >= n + 2 && m <= 2 * n) {
158       for (int i = 1; i <= n; ++i) {
159         Y(i - 1, i) += rho;
160       }
161       for (int i = 1; i < m - n; ++i) {
162         Y(i - 1, i + n) -= rho;
163       }
164       numberOfPointsFound = m;
165       initialInterpolationPointsFound = true;
166     } else {
167       std::cout << "Invalid value of m = " << m << ". Choose  n+2 <= m <=
          ↪  (n+1)(n+2)/2. Recommended: m = 2n+1."
168                 << std::endl;
169       std::cin.get();
```

```
170       std::exit(1);
171     }
172     return Y.block(0, 0, n, numberOfPointsFound);
173   }
174
175   Eigen::MatrixXd DFO_Model::findLastSetOfInterpolationPoints() {
176     if (m > 2 * n + 1) {
177       // Calculate the sigmas
178       for (int i = 1; i <= n; ++i) {
179         if (fvals[i + n] < fvals[i])
180           sigmas[i - 1] = -1;
181         else
182           sigmas[i - 1] = 1;
183       }
184
185       // Find the last set of interpolation points, while also storing the
          ↪  ps, qs and is for later usage.
186       int j = 2 * n + 2;
187       int l = 1; //Number of cycles  (i.e. number of times i has become  i
          ↪  == 3 * n + 2)
188       int p;
189       int q;
190       int index = 0;
191
192       for (int i = 2 * n + 2; i <= m; ++i) {
193
194         if (j >= 3 * n + 2) {
195           j = j - n;
196           l++;
197         }
198         p = (j - 2 * n - 1);
199
200         if (p + l >= 1 && p + l <= n)
201           q = p + l;
202         else
203           q = p + l - n;
204
205         ps[index] = p;
206         qs[index] = q;
207         is[index] = i;
208
209         Y(p - 1, i - 1) += rho * sigmas[p - 1];
210         Y(q - 1, i - 1) += rho * sigmas[q - 1];
211
212         index++;
213         j++;
214
215       }
216     } else {
217       std::cout << "findLastSetOfInterpilationPoints() was called when m <=
          ↪  2*n + 1" << std::endl;
218     }
219     initialInterpolationPointsFound = true;
220     return Y.block(0, 2 * n + 1, n, m - (2 * n + 1));
221   }
222
223   void DFO_Model::initializeModel() {
```

```
224   derivativeAtCenterpoint = derivatives.col(0);
225   Winv = calculateWExplicitly();
226   createW();
227   modelInitialized = true;
228   bestPointIndex = 1;
229   bestPoint = Y.col(0);
230
231 }
232
233 void DFO_Model::update(Eigen::MatrixXd yNews,
234                        Eigen::VectorXd fvalNews,
235                        Eigen::MatrixXd gradients,
236                        Eigen::VectorXi indicies,
237                        int numberOfPoints,
238                        UpdateReason updateReason) {
239   for (int i = 0; i < numberOfPoints; ++i) {
240     update(yNews.col(i), fvalNews(i), gradients.col(i), indicies(i),
      ↪  updateReason);
241   }
242 }
243
244 void DFO_Model::update(Eigen::VectorXd yNew,
245                        double fvalNew,
246                        Eigen::VectorXd grad,
247                        unsigned int t,
248                        UpdateReason updateReason) {
249   int oldBestPointIndex = bestPointIndex;
250   int oldBestFval = fvals[bestPointIndex - 1];
251
252   if (updateReason == INCLUDE_NEW_OPTIMUM) {
253     lagrangeMultipliers = subproblem.getLagrangeMultipliers();
254
255     if (fvalNew < bestPointAllTimeFunctionValue) {
256       bestPointAllTimeFunctionValue = fvalNew;
257       bestPointAllTime = yNew;
258     }
259   }
260
261   if (updateReason == IMPROVE_POISEDNESS) {
262     if (fvalNew < bestPointAllTimeFunctionValue) {
263       bestPointAllTimeFunctionValue = fvalNew;
264       bestPointAllTime = yNew;
265     }
266   }
267   if (updateReason == INCLUDE_NEW_POINT) {
268     if (fvalNew < bestPointAllTimeFunctionValue) {
269       bestPointAllTimeFunctionValue = fvalNew;
270       bestPointAllTime = yNew;
271     }
272   }
273
274   if (updateReason == FORCED_IMPROVE_MODEL) {
275
276   } else {
277     if (fvalNew < fvals[bestPointIndex - 1] && updateReason ==
      ↪  INCLUDE_NEW_OPTIMUM) {
278       bestPoint = yNew;
```

```
279       bestPointIndex = t;
280     }
281   }
282
283   eigen_col(Y, yNew, t - 1);
284   fvals(t - 1) = fvalNew;
285   if (ng > 0) {
286     eigen_col(derivatives, grad, t - 1);
287   }
288   updateQuadraticModelNew(yNew, fvalNew, t);
289
290   if ((updateReason == IMPROVE_POISEDNESS || updateReason ==
      ↪  INCLUDE_NEW_POINT)) {
291     if (t == oldBestPointIndex && fvalNew > oldBestFval) { // removing
        ↪  optimum :(
292       bestPointIndex = 1;
293       for (int j = 2; j <= m; ++j) {
294         if (fvals[j - 1] < fvals[bestPointIndex - 1]) {
295           bestPointIndex = j;
296         }
297       }
298       bestPoint = Y.col(bestPointIndex - 1);
299     }
300   }
301   Winv = calculateWExplicitly();
302   createW();
303
304   isModelCFL = -1;
305
306 }
307
308 double DFO_Model::evaluateQuadraticModel(Eigen::VectorXd point) {
309   enhancedModel.ComputeModel(Y, derivatives, derivativeAtCenterpoint,
      ↪  fvals, y0, bestPoint, rho, r, ng);
310
311   double e_c = 0;
312   Eigen::VectorXd e_g(n);
313   Eigen::MatrixXd e_h(n, n);
314   enhancedModel.GetModel(e_c, e_g, e_h);
315   double val = e_c + point.transpose() * e_g + 0.5 * point.transpose() *
      ↪  e_h * point;
316   return val;
317 }
318
319 void DFO_Model::shiftCenterPointOfQuadraticModelNew(Eigen::VectorXd s) {
320   double tmp1 = gradient.transpose() * s;
321   double tmp2 = (0.5 * (s.transpose() * hessian) * s);
322   constant += tmp1 + tmp2;
323   gradient += hessian * s;
324   for (int i = 1; i <= m; ++i) {
325     eigen_col(Y, Y.col(i - 1) - s, i - 1);
326   }
327   bestPoint -= s;
328   bestPointAllTime -= s;
329   y0 += s;
330 }
331
```

```
332
333 void DFO_Model::findWorstPointInInterpolationSet(Eigen::VectorXd &dNew,
    ↪  int &indexOfWorstPoint) {
334   double worstPoisedness = 0;
335   Eigen::VectorXd poisedness(m);
336   int index = -1;
337   double c;
338   Eigen::VectorXd grad(n);
339   Eigen::MatrixXd hess(n, n);
340
341   // Creating the lagrange polynomial.
342   for (int t = 1; t <= m; ++t) {
343     createLagrangePolynomial(t, c, grad, hess);
344
345     // Find min and max of l_t(x)
346     subproblem.setConstant(c);
347     subproblem.setGradient(grad);
348     subproblem.setHessian(hess);
349     vector<double> xsolMax;
350     vector<double> fsolMax;
351     vector<double> xsolMin;
352     vector<double> fsolMin;
353     //PrintLagrangePolynomial(t);
354     subproblem.Solve(xsolMax, fsolMax, (char *) "Maximize", y0,
      ↪  bestPoint, Y.col(t - 1));
355     subproblem.Solve(xsolMin, fsolMin, (char *) "Minimize", y0,
      ↪  bestPoint, Y.col(t - 1));
356     poisedness(t - 1) = std::max(abs(fsolMax[0]), abs(fsolMin[0]));
357
358     Eigen::VectorXd d1(n);
359     Eigen::VectorXd d2(n);
360     for (int i = 0; i < n; ++i) {
361       d1[i] = xsolMax[i];
362       d2[i] = xsolMin[i];
363     }
364
365     double temp = 0;
366     if ((abs(fsolMax[0]) >= abs(fsolMin[0])) && abs(fsolMax[0]) >=
      ↪  worstPoisedness) {
367       worstPoisedness = abs(fsolMax[0]);
368       for (int i = 0; i < xsolMax.size(); ++i) {
369         dNew[i] = xsolMax[i];
370       }
371       index = t;
372     } else if ((abs(fsolMin[0]) > abs(fsolMax[0])) && abs(fsolMin[0]) >=
      ↪  worstPoisedness) {
373       worstPoisedness = abs(fsolMin[0]);
374       for (int i = 0; i < xsolMin.size(); ++i) {
375         dNew[i] = xsolMin[i];
376       }
377       index = t;
378     }
379   }
380   if (worstPoisedness > lambda) {
381
382     if (index == bestPointIndex) {
383       int k = -1;
```

```
384       double tmp = -1;
385       for (int j = 1; j <= m; j++) {
386         if (poisedness[j - 1] > lambda && poisedness[j - 1] > tmp && j !=
            ↪  bestPointIndex) {
387           k = j;
388           tmp = poisedness[j - 1];
389         }
390       }
391       if (k != -1) {
392         indexOfWorstPoint = k;
393         indexOfWorstPoint = -1;
394         return;
395       } else {
396         indexOfWorstPoint = -1;
397         return;
398       }
399
400     } else {
401       indexOfWorstPoint = index;
402     }
403
404     createLagrangePolynomial(indexOfWorstPoint, c, grad, hess);
405     subproblem.setConstant(c);
406     subproblem.setGradient(grad);
407     subproblem.setHessian(hess);
408     vector<double> xsolMax;
409     vector<double> fsolMax;
410     vector<double> xsolMin;
411     vector<double> fsolMin;
412
413     subproblem.SetTrustRegionRadius(GetTrustRegionRadius() * 1);
414     subproblem.Solve(xsolMax, fsolMax, (char *) "Maximize", y0,
        ↪  bestPoint, bestPoint);
415     subproblem.Solve(xsolMin, fsolMin, (char *) "Minimize", y0,
        ↪  bestPoint, bestPoint);
416
417     if ((abs(fsolMax[0]) >= abs(fsolMin[0]))) {
418       for (int i = 0; i < xsolMax.size(); ++i) {
419         dNew[i] = xsolMax[i];
420       }
421     } else {
422       for (int i = 0; i < xsolMin.size(); ++i) {
423         dNew[i] = xsolMin[i];
424       }
425     }
426   } else {
427     indexOfWorstPoint = -1; // Indicates that the required poisedness is
        ↪  already achieved
428   }
429   std::cout << "Required poisedness: " << lambda << "\nPoisedness: " <<
      ↪  worstPoisedness << "\n";
430
431 }
432
433
434 void DFO_Model::calculateAMatrix(Eigen::MatrixXd &A, Eigen::MatrixXd
    ↪  &Ycopy) {
```

```
435     int elem = 0;
436     /// Constant
437     for (int i = 1; i <= m; ++i) {
438       A(i - 1, 0) = 1;
439     }
440     elem++;
441     /// Linear
442     for (int i = 1; i <= n; ++i) {
443       if (elem < m) {
444         for (int j = 1; j <= m; ++j) {
445           A(j - 1, elem) = Ycopy(i - 1, j - 1);
446         }
447         elem++;
448       } else {
449         break;
450       }
451     }
452     /// Squared
453     for (int i = 1; i <= n; ++i) {
454       if (elem < m) {
455         for (int j = 1; j <= m; ++j) {
456           A(j - 1, elem) = Ycopy(i - 1, j - 1) * Ycopy(i - 1, j - 1);
457         }
458         elem++;
459       } else {
460         break;
461       }
462     }
463     /// Cross terms
464     int rows = n;
465     int it = 1;
466     int col = 1;
467     int t = 2;
468     for (int k = 1; k <= n - 1; ++k) {
469       for (int i = t; i <= n - 1; ++i) {
470         for (int j = 1; j <= m; ++j) {
471           if (elem < m) {
472             double x1 = Ycopy(k - 1, j - 1);
473             double x2 = Ycopy(i - 1, j - 1);
474             A(j - 1, elem) = x1 * x2;
475           } else {
476             break;
477           }
478         }
479         elem++;
480       }
481       t++;
482     }
483 }
484
485
486
487 int DFO_Model::findPointToReplaceWithNewOptimum(Eigen::VectorXd yNew) {
488     // Create the w vector
489     Eigen::VectorXd w(n + m + 1);
490     for (int i = 1; i <= m; ++i) {
491       w(i - 1) = 0.5 * std::pow((Y.col(i - 1)).transpose() * (yNew), 2);
```

```
  }
  eigen_tail(w, yNew, n);
  w(m) = 1;

  Eigen::VectorXd Hw = Eigen::VectorXd::Zero(m + n + 1);
  Hw = Winv * w;

  int indexToBeReplaced = 1;
  double currentMax = -1;
  for (int i = 1; i <= m; ++i) {
    if (i == bestPointIndex) {
      continue;
    }
    double distance = norm((bestPoint - Y.col(i - 1)));
    double distanceWeight = distance;

    double lagval = std::abs((Hw)(i - 1));
    double value = distanceWeight * lagval;
    if (value >= currentMax) {
      indexToBeReplaced = i;
      currentMax = value;
    }
  }

  std::cout << "Point selected: " << indexToBeReplaced << "\n";

  return indexToBeReplaced;
}

int DFO_Model::getBestPointIndex() {
  return bestPointIndex;
}

void DFO_Model::SetFunctionValue(int t, double value) {
  fvals[t - 1] = value;
}

void DFO_Model::SetFunctionValueAndDerivatives(int t, double value,
 ↪  Eigen::VectorXd grad) {
  fvals(t - 1) = value;
  for (int i = 0; i < ng; ++i) {
    derivatives(i, t - 1) = grad(i);
  }
}

void DFO_Model::SetTrustRegionRadiusForSubproblem(double radius) {
  subproblem.SetTrustRegionRadius(radius);
}

Eigen::VectorXd DFO_Model::FindLocalOptimum() {
  Eigen::VectorXd localOptimum(n);
  vector<double> xsol;
  vector<double> fsol;
  /// The enhanced model;
  enhancedModel.ComputeModel(Y, derivatives, derivativeAtCenterpoint,
   ↪  fvals, y0, bestPoint, rho, r, ng);
  double e_c = 0;
```

```
547   Eigen::VectorXd e_g(n);
548   Eigen::MatrixXd e_h(n, n);
549   enhancedModel.GetModel(e_c, e_g, e_h);
550   subproblem.setHessian(e_h);
551   subproblem.setGradient(e_g);
552   subproblem.setConstant(e_c);
553   subproblem.SetTrustRegionRadius(rho);
554   subproblem.Solve(xsol, fsol, (char *) "Minimize", y0, bestPoint,
      ↪  bestPoint);
555   for (int i = 0; i < n; i++) {
556     localOptimum[i] = xsol[i];
557   }
558   return localOptimum;
559 }
560
561
562 double DFO_Model::findLargestDistanceBetweenPointsAndOptimum() {
563   int t = -1;
564   double maxDistance = -1;
565   for (int i = 0; i < m; ++i) {
566     double dist = norm(Y.col(i) - bestPoint);
567     if (dist > maxDistance) {
568       t = i + 1;
569       maxDistance = dist;
570     }
571   }
572   return maxDistance;
573 }
574 double DFO_Model::ComputeLagrangePolynomial(int t, Eigen::VectorXd point)
      ↪  {
575   double c;
576   Eigen::VectorXd grad(n);
577   Eigen::MatrixXd hess(n, n);
578   createLagrangePolynomial(t, c, grad, hess);
579   double val = c + grad.transpose() * point + 0.5 * point.transpose() *
      ↪  hess * point;
580   return val;
581 }
582
583 double DFO_Model::PrintLagrangePolynomial(int t) {
584
585   double c;
586   Eigen::VectorXd grad(n);
587   Eigen::MatrixXd hess(n, n);
588   createLagrangePolynomial(t, c, grad, hess);
589   std::cout << "Lagrange polynomial ---------------------- " << t <<
      ↪  "\n";
590   std::cout << "c = " << c << std::endl;
591   std::cout << "gradient = " << std::endl << grad << std::endl;
592   std::cout << "hessian = " << std::endl << hess << std::endl;
593 }
594
595 Eigen::VectorXd
      ↪  DFO_Model::FindLocalOptimumOfAbsoluteLagrangePolynomial(int t) {
596   double c;
597   Eigen::VectorXd grad(n);
598   Eigen::MatrixXd hess(n, n);
```

```
599   createLagrangePolynomial(t, c, grad, hess);
600
601   // Find min and max of l_t(x)
602   subproblem.setConstant(c);
603   subproblem.setGradient(grad);
604   subproblem.setHessian(hess);
605   vector<double> xsolMax;
606   vector<double> fsolMax;
607   vector<double> xsolMin;
608   vector<double> fsolMin;
609   subproblem.Solve(xsolMax, fsolMax, (char *) "Maximize", y0, bestPoint,
      ↪  Y.col(t - 1));
610   subproblem.Solve(xsolMin, fsolMin, (char *) "Minimize", y0, bestPoint,
      ↪  Y.col(t - 1));
611
612   Eigen::VectorXd optimum(n);
613   for (int i = 0; i < xsolMax.size(); ++i) {
614     if (abs(fsolMax[0]) >= abs(fsolMin[0])) {
615       optimum[i] = xsolMax[i];
616     } else {
617       optimum[i] = xsolMin[i];
618     }
619   }
620   return optimum;
621 }
622
623 Eigen::VectorXi
    ↪  DFO_Model::GetInterpolationPointsSortedByDistanceFromBestPoint() {
624   std::vector<Eigen::VectorXd> tmp;
625   for (int i = 0; i < m; ++i) {
626     Eigen::VectorXd t(n + 1);
627     for (int j = 0; j < n; ++j) {
628       t(j) = Y(j, i) - Y(j, bestPointIndex - 1);
629     }
630     t(n) = i + 1;
631     tmp.push_back(t);
632   }
633   std::sort(tmp.begin(), tmp.end(), cmp2);
634   Eigen::VectorXi indicesSortedByDescendingNorm(m);
635   for (int i = 0; i < m; ++i) {
636     indicesSortedByDescendingNorm[i] = tmp[i][n];
637   }
638   return indicesSortedByDescendingNorm;
639 }
640
641 bool DFO_Model::FindPointToReplaceWithPointOutsideScaledTrustRegion(int
    ↪  t, Eigen::VectorXd &dNew) {
642   subproblem.SetTrustRegionRadius(rho);
643   dNew = FindLocalOptimumOfAbsoluteLagrangePolynomial(t);
644   if (std::abs(ComputeLagrangePolynomial(t, dNew)) > lambda) {
645     return true;
646   }
647   return false;
648 }
649
650
651 int DFO_Model::isPointAcceptable(Eigen::VectorXd point) {
```

```
652   // Create the w vector
653   Eigen::VectorXd w(n + m + 1);
654   for (int i = 1; i <= m; ++i) {
655     w(i - 1) = 0.5 * std::pow((Y.col(i - 1)).transpose() * (point), 2);
656   }
657   eigen_tail(w, point, n);
658   w(m) = 1;
659
660   Eigen::VectorXd Hw = Eigen::VectorXd::Zero(m + n + 1);
661   Hw = Winv * w;
662
663   int indexToBeReplaced = -1;
664   double currentMax = -1;
665   for (int j = 1; j <= m; ++j) {
666     if (j == bestPointIndex) {
667       continue;
668     }
669     double lagval = std::abs((Hw)(j - 1));
670     if ((lagval > 1) || (norm(Y.col(j - 1) - bestPoint) > r * rho)) {
671       double distance = norm((bestPoint - Y.col(j - 1)));
672       double distanceWeight = std::pow(distance, 2);
673       if (distance > 2 * rho) {
674         distanceWeight += 100000000 * distanceWeight;
675       }
676       double value = distanceWeight * lagval;
677       if (value >= currentMax) {
678         indexToBeReplaced = j;
679         currentMax = value;
680       }
681     }
682   }
683   return indexToBeReplaced;
684 }
685
686 double DFO_Model::norm(Eigen::VectorXd a) {
687   if (normType == 0) {
688     return (a).lpNorm<Infinity>();
689   } else if (normType == 2) {
690     return (a).norm();
691   }
692 }
693
694 bool DFO_Model::FindReplacementForPointsOutsideRadius(double radius,
695                                                       Eigen::MatrixXd
                                                        ↪  &newPoints,
696                                                       Eigen::VectorXi
                                                        ↪  &newIndices) {
697   Eigen::DiagonalMatrix<double, Eigen::Dynamic> copyS = S;
698   Eigen::MatrixXd copyZ = Z;
699   Eigen::MatrixXd copyUpsilon = Upsilon;
700   Eigen::MatrixXd copyXi = Xi;
701   Eigen::MatrixXd copyY = Y;
702   Eigen::MatrixXd copyWinv = Winv;
703   bool retVal = true;
704
705
706   /// Find points outside r*radius
```

```
  Eigen::VectorXi sortedPoints =
   ↪  GetInterpolationPointsSortedByDistanceFromBestPoint();
  int number_of_points_outside = GetNumberOfPointsOutsideRadius(radius);
  sortedPoints.conservativeResize(number_of_points_outside);

  if (number_of_points_outside <= 0) {
    return false;
  }
  newIndices.resize(number_of_points_outside);

  for (int i = 0; i < number_of_points_outside; ++i) {
    newIndices(i) = -1;
  }


  subproblem.SetTrustRegionRadius((radius / r) * 0.9);
  newPoints.resize(n, number_of_points_outside);
  newPoints.setZero();
  Eigen::VectorXd dNew(n);
  int addedPoints = 0;
  int j = 0;
  for (int i = 0; i < number_of_points_outside; ++i) {
    dNew = FindLocalOptimumOfAbsoluteLagrangePolynomial(sortedPoints(i));
    double lagabsval =
     ↪  std::abs(ComputeLagrangePolynomial(sortedPoints(i), dNew));
    if (lagabsval > lagabsvalMin) {
      newIndices(addedPoints) = sortedPoints(i);
      newPoints.col(addedPoints) = dNew;
      eigen_col(newPoints, dNew, addedPoints);

      eigen_col(Y, dNew, sortedPoints(i) - 1);
      Winv = calculateWExplicitly();
      createW();
      addedPoints++;
      j++;
      //break;
    } else {
      PrintLagrangePolynomial(sortedPoints(i));
      if (newIndices.rows() == 0 || newIndices.rows() == addedPoints) {
        break;
      }
      //break;
    }
  }

  if (addedPoints != newIndices.rows()) {
    newIndices.conservativeResize(addedPoints);
    newPoints.conservativeResize(n, addedPoints);
  }
  if (addedPoints == 0) {
    retVal = false;
  }

  /// reset!!
  Y = copyY;
  Xi = copyXi;
  Upsilon = copyUpsilon;
```

```
762   S = copyS;
763   Z = copyZ;
764   Winv = copyWinv;
765   createW();
766   return retVal;
767 }
768 int DFO_Model::GetNumberOfPointsOutsideRadius(double radius) {
769   /// Number of points outside radius
770   int number = 0;
771   for (int j = 1; j <= m; ++j) {
772     if (norm(Y.col(j - 1) - bestPoint) > radius) {
773       number++;
774     }
775   }
776   return number;
777 }
778
779
780 void DFO_Model::createW() {
781   W = Eigen::MatrixXd::Zero(m + n + 1, m + n + 1);
782   Eigen::MatrixXd A = Eigen::MatrixXd::Zero(m, m);
783   Eigen::MatrixXd X = Eigen::MatrixXd::Zero(n + 1, m);
784   for (int i = 1; i <= m; ++i) {
785     for (int j = 1; j <= m; ++j) {
786       A(i - 1, j - 1) = 0.5 * std::pow(Y.col(i - 1).transpose() * Y.col(j
          ↪  - 1), 2);
787     }
788   }
789
790   for (int i = 1; i <= m; ++i) {
791     X(0, i - 1) = 1;
792   }
793
794   for (int i = 1; i <= m; ++i) {
795     Eigen::VectorXd tmp = X.col(i - 1);
796     eigen_tail(tmp, Y.col(i - 1), n);
797     eigen_col(X, tmp, i - 1);
798   }
799
800   W.topLeftCorner(m, m) = A;
801   W.bottomLeftCorner(n + 1, m) = X;
802   W.topRightCorner(m, n + 1) = X.transpose();
803 }
804
805 // This is slow and should only be used for testing and debugging.
806 Eigen::MatrixXd DFO_Model::calculateWExplicitly() {
807   double precision = 0.00001;
808   Eigen::MatrixXd W = Eigen::MatrixXd::Zero(m + n + 1, m + n + 1);
809   Eigen::MatrixXd A = Eigen::MatrixXd::Zero(m, m);
810   Eigen::MatrixXd X = Eigen::MatrixXd::Zero(n + 1, m);
811   for (int i = 1; i <= m; ++i) {
812     for (int j = 1; j <= m; ++j) {
813       A(i - 1, j - 1) = 0.5 * std::pow(Y.col(i - 1).transpose() * Y.col(j
          ↪  - 1), 2);
814     }
815   }
816   for (int i = 1; i <= m; ++i) {
```

```
817       X(0, i - 1) = 1;
818     }
819     for (int i = 1; i <= m; ++i) {
820       Eigen::VectorXd tmp = X.col(i - 1);
821       eigen_tail(tmp, Y.col(i - 1), n);
822       eigen_col(X, tmp, i - 1);
823     }
824
825     W.topLeftCorner(m, m) = A;
826     W.bottomLeftCorner(n + 1, m) = X;
827     W.topRightCorner(m, n + 1) = X.transpose();
828     Eigen::MatrixXd Winv(m + n + 1, m + n + 1);
829     Winv = W.inverse();
830     Eigen::FullPivLU<Eigen::MatrixXd> lu_decompW(W);;
831     Eigen::MatrixXd WLUinv(m + n + 1, m + n + 1);
832     WLUinv = lu_decompW.inverse();
833     return WLUinv;
834   }
835
836
837   void DFO_Model::UpdateOptimum() {
838     int i = 1;
839     for (int j = 2; j <= m; ++j) {
840       if (fvals[j - 1] < fvals[i - 1]) {
841         i = j;
842       }
843     }
844     if (i != bestPointIndex) {
845       bestPointIndex = i;
846       bestPoint = Y.col(i - 1);
847     }
848   }
849
850   bool DFO_Model::isPoised(Eigen::VectorXd &dNew, int
      ↪  &indexOfPointToBeReplaced, double radius) {
851     bool ispoised = false;
852     indexOfPointToBeReplaced = -1;
853     int numberOfPointsOutsideRadius =
       ↪  GetNumberOfPointsOutsideRadius(radius);
854     if (numberOfPointsOutsideRadius >= 1) {
855       // Find points outside r*radius
856       Eigen::VectorXi sortedPoints =
        ↪  GetInterpolationPointsSortedByDistanceFromBestPoint();
857       sortedPoints.conservativeResize(numberOfPointsOutsideRadius);
858       subproblem.SetTrustRegionRadius(radius / r);
859       for (int i = 0; i < numberOfPointsOutsideRadius; ++i) {
860         dNew =
          ↪  FindLocalOptimumOfAbsoluteLagrangePolynomial(sortedPoints[i]);
861         double lagabsval =
          ↪  std::abs(ComputeLagrangePolynomial(sortedPoints[i], dNew));
862         if (lagabsval > lagabsvalMin) {
863           indexOfPointToBeReplaced = sortedPoints[i];
864           break;
865         } else {
866
867           PrintLagrangePolynomial(sortedPoints[i]);
868         }
```

```
    }
  } else {
    subproblem.SetTrustRegionRadius(radius);
    findWorstPointInInterpolationSet(dNew, indexOfPointToBeReplaced);
  }

  if (indexOfPointToBeReplaced == -1) {
    isModelCFL = 1;
    ispoised = true;
  } else {
    isModelCFL = 0;
  }
}

void DFO_Model::modelImprovementStep(Eigen::VectorXd &dNew, int
 ↪  &indexOfPointToBeReplaced) {
  indexOfPointToBeReplaced = -1;
  Eigen::VectorXi sortedPoints =
   ↪  GetInterpolationPointsSortedByDistanceFromBestPoint();
  for (int i = 0; i < m; ++i) {
    int t = sortedPoints[i];
    subproblem.SetTrustRegionRadius(rho);
    dNew = FindLocalOptimumOfAbsoluteLagrangePolynomial(t);
    double lagabsval = std::abs(ComputeLagrangePolynomial(t, dNew));
    if (lagabsval > lambda || norm((Y.col(t - 1) - bestPoint))
        > r * rho) {

      if (lagabsval > lagabsvalMin) {
        indexOfPointToBeReplaced = t;
        break;
      }
    }
  }

  if (indexOfPointToBeReplaced == -1) {
    isModelCFL = 1;
  } else {
    isModelCFL = -1;
  }
}

void DFO_Model::isLagrangePoly() {
  for (int t = 1; t <= m; ++t) {

    double c;
    Eigen::VectorXd grad(n);
    Eigen::MatrixXd hess(n, n);
    createLagrangePolynomial(t, c, grad, hess);

    for (int j = 1; j <= m; j++) {
      double val = c + grad.transpose() * Y.col(j - 1) + 0.5 * (Y.col(j -
       ↪  1)).transpose() * hess * Y.col(j - 1);
      if (j == t) {
        std::cout << "value should be 1, but is: " << val << "\n";
      } else {
        std::cout << "value should be 0, but is: " << val << "\n";
      }
```

```
923       }
924     }
925   }
926
927   void DFO_Model::Converged(int iterations,
928                             int number_of_tiny_improvements,
929                             int number_of_function_calls,
930                             int number_of_parallell_function_calls) {
931     {
932       Eigen::VectorXd gradient = GetLagrangianGradient(GetBestPoint());
933       Eigen::MatrixXd Yabs(n, m);
934       for (int j = 0; j < m; ++j) {
935         Eigen::VectorXd sd = (Y).col(j) + getCenterPoint();
936         eigen_col(Yabs, sd, j);
937       }
938       std::cout.clear();
939       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
940       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
941       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
942       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
943       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
944       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
945       std::cout << "\033[1;36;mDFO terminated. Trust region radius too
          ↪  small.\033[0m" << std::endl;
946
947       std::cout << "\033[1;34;m " << "Norm of gradient at best point = " <<
          ↪  "\033[0m" << gradient.norm() << "\n";
948       std::cout << "\033[1;34;m " << "Best point index = " << "\033[0m" <<
          ↪  bestPointIndex << "\n";
949       std::cout << "\033[1;34;m " << "Fvals = \n" << "\033[0m" << fvals <<
          ↪  "\n";
950       std::cout << "\033[1;34;m " << "Y = \n" << "\033[0m" << Y << "\n";
951       std::cout << "\033[1;34;m " << "Y absolute = \n" << "\033[0m" << Yabs
          ↪  << "\n";
952       std::cout << "\033[1;34;m " << "Ybest (abs) = \n" << "\033[0m" <<
          ↪  bestPoint + y0 << "\n";
953       std::cout << "\033[1;34;m " << "Trust region radius is: " <<
          ↪  "\033[0m" << rho << std::endl;;
954       std::cout << "\033[1;34;m " << "Best found, all time, value: " <<
          ↪  "\033[0m" << bestPointAllTimeFunctionValue
955                 << "\n";
956       std::cout << "\033[1;34;m " << "Best found, all time, point: \n" <<
          ↪  "\033[0m" << bestPointAllTime + y0;
957
958       std::cout << "\nm = " << m << "\n";
959       std::cout << "n = " << n << "\n";
960       std::cout << "ng = " << ng << "\n";
961
962       std::cout << "Best point (absolute):\n" << getCenterPoint() +
          ↪  GetBestPoint()
```

```
963                 << "\nWith value: " <<
                     ↪  GetFunctionValue(getBestPointIndex()) << "\n";
964
965     std::cout << "\033[1;36;mFunction Calls \033[0m" <<
         ↪  number_of_function_calls << "\n";
966     std::cout << "\033[1;36;mParallell Function Calls \033[0m" <<
         ↪  number_of_parallell_function_calls << "\n";
967   }
968 }
969
970 void DFO_Model::createLagrangePolynomial(int t, double &c,
     ↪  Eigen::VectorXd &grad, Eigen::MatrixXd &hess) {
971   grad.setZero();
972   hess.setZero();
973   c = 0;
974   createW();
975   Eigen::VectorXd ans(n + m + 1);
976   Eigen::VectorXd rhs = Eigen::VectorXd::Zero(n + m + 1);
977   rhs(t - 1) = 1;
978   ans = W.colPivHouseholderQr().solve(rhs);
979
980   c = ans(m);
981   grad = ans.tail(n);
982   for (int k = 1; k <= m; ++k) {
983     hess += ans(k - 1) * (Y.col(k - 1)) * (Y.col(k - 1)).transpose();
984   }
985 }
986
987 // return true when no new points are found. is_poised = true
988 bool DFO_Model::ModelImprovementAlgorithm(double radius, Eigen::MatrixXd
     ↪  &newPoints, Eigen::VectorXi &newIndices) {
989   /// Get all points inside of the trust-region
990   Eigen::MatrixXd tmpNewPoints;
991   Eigen::VectorXi tmpNewIndices;
992   FindReplacementForPointsOutsideRadius(radius, tmpNewPoints,
       ↪  tmpNewIndices);
993   Eigen::MatrixXd copyY = Y;
994   Eigen::VectorXi changed(m);
995   changed.setZero();
996   for (int i = 0; i < tmpNewPoints.cols(); ++i) {
997     if (tmpNewIndices(i) != -1) {
998       eigen_col(Y, tmpNewPoints.col(i), tmpNewIndices(i) - 1);
999       changed(tmpNewIndices(i) - 1) = 1;
1000    } else {
1001      break;
1002    }
1003  }
1004  subproblem.SetTrustRegionRadius(radius);
1005  /// Improve poisedness until required poisedness is achieved.
1006  Eigen::VectorXd dNew(n);
1007  int index = -1;
1008  while (1) {
1009    findWorstPointInInterpolationSet(dNew, index);
1010    if (index == -1) {
1011      break;
1012    } else {
1013      eigen_col(Y, dNew, index - 1);
```

```
      changed(index - 1) = 1;
    }
  }

  int number_of_new_points = 0;
  for (int i = 0; i < m; i++) {
    if (changed(i) == 1) {
      number_of_new_points++;
    }
  }
  newPoints.resize(n, number_of_new_points);
  newIndices.resize(number_of_new_points);

  int i = 0;
  for (int j = 1; j <= m; ++j) {
    if (changed(j - 1) == 1) {
      eigen_col(newPoints, Y.col(j - 1), i);
      newIndices(i) = j;
      i++;
    }
  }
  Y = copyY; /// reset.
  createW();

  if (number_of_new_points == 0) {
    isModelCFL = 1;
  } else {
    isModelCFL = 0;
  }

  return (number_of_new_points == 0);
}

// Will only be valid for bestpoint, because the lagrange multipliers are
 ↪  found for that point.
Eigen::VectorXd DFO_Model::GetLagrangianGradient(Eigen::VectorXd point) {
  Eigen::VectorXd lagGrad(n);
  lagGrad = GetGradientAtPoint(point);
  subproblem.SetTrustRegionRadius(rho);
  if (ng > 0) {
    calculateLagrangeMultipliers();
    Eigen::MatrixXd conGrad = subproblem.getGradientConstraints(point +
     ↪  y0);
    for (int i = 0; i < subproblem.getNumberOfConstraints(); i++) {
      lagGrad -= lagrangeMultipliers[i] * ((conGrad.row(i)).transpose());
    }
  }
  return lagGrad;
}

void DFO_Model::calculateLagrangeMultipliers() {

  subproblem.calculateLagrangeMultipliers((char *) "Minimize", y0,
   ↪  bestPoint, bestPoint);
  lagrangeMultipliers = subproblem.getLagrangeMultipliers();

}
```

```
1068
1069 Eigen::VectorXd
     ↪ DFO_Model::ScaleVariablesFromApplicationToAlgorithm(Eigen::VectorXd
     ↪ point) {
1070   Eigen::VectorXd ret(point.rows());
1071   point = point / 10000.0;
1072   for (int i = 0; i < point.rows(); ++i) {
1073     if (i % 2 == 0 && i != 0) {
1074       //z coord
1075       ret[i] = point[i] * 30.0;
1076     } else {
1077       //xy coord
1078       ret[i] = point[i];
1079     }
1080   }
1081   return ret;
1082
1083 }
1084 void DFO_Model::setInitialy0(Eigen::VectorXd a) {
1085
1086   this->y0 = a;
1087 }
1088 void DFO_Model::findVariableMeaning(Eigen::VectorXd realvars,
     ↪ Eigen::VectorXd scaling) {
1089   mapNormalToFieldopt = Eigen::VectorXi::Zero(6); // [x0 y0 z0, x1 y1 z1]
1090   for (int i = 0; i < 6; i++) {
1091     for (int j = 0; j < 6; j++) {
1092       if (abs(copyOfStartingPoint[i] - realvars[j]) <= 0.000001) {
1093         mapNormalToFieldopt[i] = j;
1094       }
1095     }
1096   }
1097   subproblem.SetMappingVariables(mapNormalToFieldopt, scaling);
1098 }
1099
1100 void DFO_Model::PrintSortedBestPoint(Eigen::VectorXd scaling) {
1101   Eigen::VectorXd wellStart(3);
1102   Eigen::VectorXd wellEnd(3);
1103   Eigen::VectorXd splineIsh = y0 + bestPoint;
1104
1105   for (int i = 0; i < 6; ++i) {
1106     splineIsh[i] = splineIsh[i] * scaling[i];
1107   }
1108   for (int i = 0; i < 3; i++) {
1109     wellStart[i] = splineIsh(mapNormalToFieldopt[i]);
1110     wellEnd[i] = splineIsh(mapNormalToFieldopt[i + 3]);
1111   }
1112   std::cout << "Well information\nCoordinates:\n" << wellStart << "\n" <<
     ↪ wellEnd << "\n" << "spline length: "
1113             << (wellStart - wellEnd).norm() << "\n\n";
1114 }
1115
1116 }
1117 }
```

Subproblem.h

```
//
// Created by joakim on 08.03.18.
//

#ifndef FIELDOPT_SUBPROBLEM_H
#define FIELDOPT_SUBPROBLEM_H

#include <Eigen/Core>
#include <Eigen/Dense>
#include "FieldOpt-3rdPartySolvers/handlers/SNOPTHandler.h"
#include "FieldOpt-3rdPartySolvers/handlers/SNOPTLoader.h"
#include "Optimization/optimizer.h"
#include "VirtualSimulator.h"
namespace Optimization {
namespace Optimizers {
class Subproblem {

/*
 This class will find _one_ maximum of a quadratic function (specified by c_, g_ and H_) subject to some specified constraints.
 The constraints must be specified by whomever uses this function and they must be specified by editing the code explicitly
 (i.e, they cannot be set through function calls).
 The objective function and the constraints (except the simple/basic bounds) are put together into one set of equations:
 Let cl_i represent a linear constraint, and let cn_i represent a nonlinear one.
 let n_l and m be the numbers of linear constraints and nonlinear constraints, respectively.
 F = [f_obj, cl_0, cl_1, ..., cl_n_l, cn_0, cn_1, ..., cn_m]^T;

 The inequalities for the constraints and the objective function is given by:
 Flow <= F <= Fupp

 If you don't have a limit, use the infinity_ *(+-1) value.

 maximize    f_obj = c_ + g_^T x + x^T H_ x

 subject to     Flow <= F <= Fupp
        xlow <= x <= xupp


 Linear and nonlinear constraints are specified differently.

 Note!
 Because F contains both the objective functions and the constraints, the row-indices will start at row 1 and not row 0.

 Linear
  The linear constraints are specified through lenA, iAfun, jAvar and A.
  The first 2 are used because the matrix hatA ( "lower <= hatA*x <= upper") might be very sparse. An example will illustrate the usage:

  A = [0 1 0
```

```
    5 0 6
    0 0 8].

 lenA = 4; // There are four nonzero elements in A
 iAfun[0] = 1; //These two indices belongs to the element (0,1) in A
↪  (namely, the value 1). Now A[0] must be set to 1.
 jAvar[0] = 1;
 A[0] = 1;

 iAfun[1] = 2; //These two indices belongs to the element (1,0) in A
↪  (namely, the value 5). Now A[1] must be set to 5.
 jAvar[1] = 0;
 A[1] = 5;

 iAfun[2] = 2; //These two indices belongs to the element (1,2) in A
↪  (namely, the value 6). Now A[2] must be set to 6.
 jAvar[2] = 2;
 A[2] = 6;

 iAfun[3] = 3; //These two indices belongs to the element (2,2) in A
↪  (namely, the value 8). Now A[3] must be set to 8.
 jAvar[3] = 2;
 A[3] = 8;


Nonlinear
 The nonlinear constraints are specified through lenG, neG, iGfun and
↪  jGvar. The actual G matrix is specified in the userfunc.
 The G matrix contains both the derivative of the objective function and
↪  the derivative of the nonlinear constraints.
 The nonlinear constraints must also be specified by the userfunc, and
↪  put into appropriate place in F:
 F[4]=x_1^2 + x_2^2 + x_3^2;
 Let's say that we now, in addition to the linear constraints above,
↪  also has 1 nonlinear constraint. ("lower <= x_1^2 + x_2^2 + x_3^2
↪  <= upper").
 The partial derivatives with respect to the variables will then be:
 G[y] = 2*x_1;
 G[y+1] = 2*x_2;
 G[y+2] = 2*x_3;
 Where y is the number of partial derivatives of the objective
↪  functions.
 If f_obj = x_1 + x_2 + x_3, then
 y = 3; and
 F[0] = x_1 + x_2 + x_3;
 G[0] = 1;
 G[1] = 1;
 G[2] = 1;

 Now we must specify iGfun and jGvar.

 //From the objective function:
 iGfun[0] = 0;
 jGvar[0] = 0;
 iGfun[1] = 0;
 jGvar[1] = 1;
 iGfun[2] = 0;
```

```
  jGvar[2] = 2;


  //From the nonlinear constraints:
  iGfun[3] = 4; // NOTE NOTE! The reason why this value is 4 is because
↪  the first row is for the objective function, then  we have 3 linear
↪  constraints
  jGvar[3] = 0;

  iGfun[4] = 4;
  jGvar[4] = 1;

  iGfun[5] = 4;
  jGvar[5] = 2;


  neG = lenG = 6;




*/


 private:
  Eigen::VectorXd lastLagrangeMultipliers;
  int normType_;
  Eigen::VectorXd y0_;
  Eigen::VectorXd bestPointDisplacement_;

  int n_; // Number of variables
  int m_;    // Number of nonlinear constraints
  integer neF_; // Number of element in F
  integer neG_;
  integer lenG_;
  integer objRow_;
  double objAdd_;
  double trustRegionRadius_;

  integer *iAfun_ = NULL;
  integer *jAvar_ = NULL;
  double *A_ = NULL;
  integer lenA_;
  integer neA_;

  integer *iGfun_ = NULL;
  integer *jGvar_ = NULL;

  double *x_;

  // lower and upper bounds
  double *xlow_ = NULL;
  double *xupp_ = NULL;

  Eigen::VectorXd xlowCopy_;
  Eigen::VectorXd xuppCopy_;

```

```
150     // the initial guess for Lagrange multipliers
151     double *xmul_ = NULL;;
152
153     // the state of the variables (whether the optimal is likely to be on
154     // the boundary or not)
155     integer *xstate_ = NULL;
156
157     double *F_ = NULL;
158     double *Flow_ = NULL;
159     double *Fupp_ = NULL;
160     double *Fmul_ = NULL;
161     integer *Fstate_ = NULL;
162     char *xnames_ = NULL;
163     char *Fnames_ = NULL;
164
165     integer nxnames_;
166     integer nFnames_;
167     Settings::Optimizer *settings_;
168
169     // this is the value SNOPT considers as infinity
170     double infinity_ = 1e20;
171
172     void setConstraintsAndDimensions();
173     void setOptionsForSNOPT(SNOPTHandler &snoptHandler);
174     bool loadSNOPT(string libname = "libsnopt-7.2.12.2.so");
175     void setAndInitializeSNOPTParameters();
176     void passParametersToSNOPTHandler(SNOPTHandler &snoptHandler);
177     void setNormType(int type);
178     void setCenterPointOfModel(Eigen::VectorXd cp);
179     void setCurrentBestPointDisplacement(Eigen::VectorXd db);
180
181   public:
182
183     void SetMappingVariables(Eigen::VectorXi map1, Eigen::VectorXd
        ↪  scaling1);
184     Eigen::VectorXd GetInitialPoint();
185     //EIGEN_MAKE_ALIGNED_OPERATOR_NEW
186
187     enum NormType {
188       INFINITY_NORM = 0,
189       L2_NORM = 2,
190     };
191
192     void setQuadraticModel(double c, Eigen::VectorXd g, Eigen::MatrixXd H);
193     void setGradient(Eigen::VectorXd g);
194     void setHessian(Eigen::MatrixXd H);
195     void setConstant(double constant);
196     void printModel();
197
198     void SetNormType(int type) {
199       normType_ = type;
200     }
201
202     void SetCenterPoint(Eigen::VectorXd cp);
203     void SetBestPointRelativeToCenterPoint(Eigen::VectorXd bp);
204     ~Subproblem();
205     SNOPTHandler initSNOPTHandler();
```

```
  Subproblem(Settings::Optimizer *settings);
  Subproblem() {};
  Eigen::VectorXd getLagrangeMultipliers() {
    return lastLagrangeMultipliers;
  }
  void ResetSubproblem();

  void SetTrustRegionRadius(double radius) {
    trustRegionRadius_ = radius;
    //Flow_[1] = 0;
    //Fupp_[1] = trustRegionRadius_;
  }
  void Solve(vector<double> &xsol,
             vector<double> &fsol,
             char *optimizationType,
             VectorXd centerPoint,
             VectorXd bestPointDisplacement,
             VectorXd startingPoint);

  Eigen::MatrixXd getGradientConstraints(Eigen::VectorXd point);
  int getNumberOfConstraints();
  void SolveVirtualSimulator();
  Eigen::VectorXd FindFeasiblePoint();

  void evaluateConstraints(Eigen::VectorXd point);

  void calculateLagrangeMultipliers(char *optimizationType,
                                    VectorXd centerPoint,
                                    VectorXd bestPointDisplacement,
                                    VectorXd startingPoint);

  bool isPointFeasible(Eigen::VectorXd point);
};

}
}

#endif //FIELDOPT_SUBPROBLEM_H
```

Subproblem.cpp

```
#include <limits>
#include "Subproblem.h"
namespace Optimization {
namespace Optimizers {

#ifdef __cplusplus
extern "C" {
#endif
int SNOPTusrFG3_(integer *Status, integer *n, doublereal x[],
                 integer *needF, integer *neF, doublereal F[],
                 integer *needG, integer *neG, doublereal G[],
                 char *cu, integer *lencu,
                 integer iu[], integer *leniu,
                 doublereal ru[], integer *lenru);
#ifdef __cplusplus
```

```
}
#endif

void smallTightning(double &value, bool lower) {
  //Not used anymore.
  return;

  if (lower && value < 0) {
    value = value * 0.999;
  } else if (lower && value > 0) {
    value = value * 1.001;
  } else if ((!lower) && value > 0) {
    value = value * 0.999;
  } else if ((!lower) && value < 0) {
    value = value * 1.001;
  }

}

static Eigen::VectorXd yb_rel;
static Eigen::VectorXd y0;
static int normType;

static Eigen::MatrixXd hessian;
static Eigen::VectorXd gradient;
static double constant;
static double scale;
static VirtualSimulator virtualSimulator;

static Eigen::VectorXi mapNormalToFieldopt;
static Eigen::VectorXd scaling;

Subproblem::Subproblem(Settings::Optimizer *settings) {
  settings_ = settings;
  n_ = settings->parameters().number_of_variables;
  y0_ = Eigen::VectorXd::Zero(n_);
  y0 = Eigen::VectorXd::Zero(n_);
  bestPointDisplacement_ = Eigen::VectorXd::Zero(n_);
  xlowCopy_ = Eigen::VectorXd::Zero(n_); /// OBS should be set by the
   ↪  driver file....
  xuppCopy_ = Eigen::VectorXd::Zero(n_);
  loadSNOPT();
  normType = settings->parameters().norm_type;
  normType_ = settings->parameters().norm_type;

  virtualSimulator =
   ↪  VirtualSimulator(settings->parameters().test_problem_file);
  m_ = virtualSimulator.GetNumberOfConstraints() + 1;
  lastLagrangeMultipliers = Eigen::VectorXd::Zero(m_);

  setConstraintsAndDimensions(); // This one should set the iGfun/jGvar
   ↪  and so on.
  setAndInitializeSNOPTParameters();

  scale = 1;
  hessian = Eigen::MatrixXd::Zero(n_, n_);
  gradient = Eigen::VectorXd::Zero(n_);
```

```
  constant = 0;

  ResetSubproblem();
}

SNOPTHandler Subproblem::initSNOPTHandler() {
  string prnt_file, smry_file, optn_file;
  optn_file = settings_->parameters().thrdps_optn_file.toStdString() +
   ↪  ".opt.optn";
  smry_file = settings_->parameters().thrdps_smry_file.toStdString() +
   ↪  ".opt.summ";
  prnt_file = settings_->parameters().thrdps_prnt_file.toStdString() +
   ↪  ".opt.prnt";
  SNOPTHandler snoptHandler(prnt_file.c_str(),
                            smry_file.c_str(),
                            optn_file.c_str());
  return snoptHandler;
}

void Subproblem::setAndInitializeSNOPTParameters() {
  // the decision variables
  x_ = new double[n_];
  // the initial guess for Lagrange multipliers
  xmul_ = new double[n_];
  // the state of the variables (whether the optimal is likely to be on
  // the boundary or not)
  xstate_ = new integer[n_];
  F_ = new double[neF_];
  Fmul_ = new double[neF_];
  Fstate_ = new integer[neF_];
  nxnames_ = 1;
  nFnames_ = 1;
  xnames_ = new char[nxnames_ * 8];
  Fnames_ = new char[nFnames_ * 8];
}

void Subproblem::Solve(vector<double> &xsol,
                       vector<double> &fsol,
                       char *optimizationType,
                       Eigen::VectorXd centerPoint,
                       Eigen::VectorXd bestPointDisplacement,
                       Eigen::VectorXd startingPoint) {
  y0_ = centerPoint;
  y0 = y0_;
  bestPointDisplacement_ = bestPointDisplacement;
  yb_rel = bestPointDisplacement_;
  // Set norm specific constraints
  std::cout << "lower bounds\n" << xlowCopy_ << "\n";
  std::cout << "upper bounds\n" << xuppCopy_ << "\n";
  if (normType_ == INFINITY_NORM) {
    for (int i = 0; i < n_; ++i) {
      xlow_[i] = std::max(bestPointDisplacement_[i] - trustRegionRadius_,
       ↪  xlowCopy_[i] - y0_[i]);
      xupp_[i] = std::min(bestPointDisplacement_[i] + trustRegionRadius_,
       ↪  xuppCopy_[i] - y0_[i]);
    }
  } else if (normType_ == L2_NORM) {
```

```
    for (int i = 0; i < n_; ++i) {
      xlow_[i] = xlowCopy_[i];
      xupp_[i] = xuppCopy_[i];
    }
  }

  scale = 1; //computeScale();

  // The snoptHandler must be setup and loaded
  SNOPTHandler snoptHandler = initSNOPTHandler();
  snoptHandler.setProbName("SNOPTSolver");
  snoptHandler.setParameter(optimizationType);
  snoptHandler.initializeLagrangeVector(neF_ - 1);

  setOptionsForSNOPT(snoptHandler);
  snoptHandler.setRealParameter("Major step limit", trustRegionRadius_);
  ↪  //was 0.2
  snoptHandler.setRealParameter("Major feasibility tolerance", 1.0e-9);
  ↪  //1.0e-6

  snoptHandler.setRealParameter("Major optimality tolerance", 0.00001);

  ResetSubproblem();
  for (int i = 0; i < n_; i++) {
    //x_[i] = startingPoint[i];//bestPointDisplacement_[i];
    //x_[i] = 0.0;//bestPointDisplacement_[i];
    //x_[i] = startingPoint[i];
  }
  if (normType_ == Subproblem::L2_NORM) {
    Flow_[1] = 0;
    Fupp_[1] = trustRegionRadius_;
  }
  passParametersToSNOPTHandler(snoptHandler);
  integer Cold = 0, Basis = 1, Warm = 2;

  snoptHandler.solve(Cold, xsol, fsol);
  lastLagrangeMultipliers = snoptHandler.getLagrangeMultipliers();
  fsol[0] = fsol[0] * scale;
  integer exitCode = snoptHandler.getExitCode();

  if (exitCode != 40 && exitCode != 41 && exitCode != 1 && exitCode != 31
  ↪  && exitCode != 3 && exitCode != 32) {
    std::cout << "ExitCode is: " << exitCode << "\n";
    std::cin.get();
  }

  Eigen::VectorXd xvec(n_);
  for (int i = 1; i <= n_; ++i) {
    xvec(i - 1) = xsol[i - 1];
  }
  if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
    std::cout << "output from snopt is infeasible. ExitCode was: " <<
    ↪  exitCode << "\n";
    auto d = virtualSimulator.evaluateConstraints(xvec + y0);
    std::cout << d << "\n";
  }

```

```
175   Eigen::VectorXd tmp(n_);
176   if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
177
178     if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
179       std::cout << "output from snopt is infeasible. Will do random
          ↪  search\n";
180     }
181
182     double snopt_suggested_val = constant + gradient.transpose() * xvec +
       ↪  0.5 * xvec.transpose() * hessian * xvec;
183     Eigen::VectorXd yTry(n_); //Displacement from current center point
184     static std::random_device rd;
185     static std::mt19937 gen(rd());
186     std::uniform_real_distribution<> dis(-trustRegionRadius_,
       ↪  trustRegionRadius_);
187     int k = 0;
188
189     Eigen::VectorXd yBest = xvec;
190
191     double bestValue = constant + gradient.transpose() * xvec + 0.5 *
       ↪  xvec.transpose() * hessian * xvec;
192     double value = 0;
193
194     int ll = 0;
195     bool lock = false;
196     int fails = 0;
197     while (k < 5000) {
198       for (int i = 0; i < n_; ++i) {
199         yTry(i) = dis(gen) + bestPointDisplacement_[i];
200       }
201       if (virtualSimulator.IsFeasiblePoint(yTry + y0) == false) {
202         fails++;
203         continue;
204       } else {
205         if (lock == false) {
206           yBest = yTry;
207           bestValue = constant + gradient.transpose() * yTry + 0.5 *
             ↪  yTry.transpose() * hessian * yTry;
208           lock = true;
209         }
210       }
211       value = constant + gradient.transpose() * yTry + 0.5 *
         ↪  yTry.transpose() * hessian * yTry;
212       if (std::string(optimizationType) == "Maximize") {
213         if (value > bestValue) {
214           bestValue = value;
215           yBest = yTry;
216         }
217       }
218       if (std::string(optimizationType) == "Minimize") {
219         if (value < bestValue) {
220           bestValue = value;
221           yBest = yTry;
222         }
223       }
224
225       ++k;
```

```
226       }
227       std::cout << "Randomly generated point is infeasible, number of
          ↪  times: " << fails << "\n";
228       tmp = yBest;
229       for (int i = 1; i <= n_; ++i) {
230         xsol[i - 1] = yBest[i - 1];
231         fsol[0] = constant + gradient.transpose() * yBest + 0.5 *
          ↪  yBest.transpose() * hessian * yBest;
232       }
233     }
234
235     if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
236       std::cout << "Failed to find feasible points...\n";
237     }
238
239   }
240
241   void Subproblem::ResetSubproblem() {
242     for (int i = 0; i < n_; i++) {
243       Fstate_[i] = 0;
244       xstate_[i] = 0;
245       x_[i] = 0.0;
246       xmul_[i] = 0;
247     }
248
249     for (int h = 0; h < neF_; h++) {
250       F_[h] = 0.0;
251       Fmul_[h] = 0.0;
252     }
253
254   }
255
256   void Subproblem::passParametersToSNOPTHandler(SNOPTHandler &snoptHandler)
      ↪  {
257     snoptHandler.setProblemSize(n_, neF_);
258     snoptHandler.setObjective(objRow_);
259     snoptHandler.setA(lenA_, iAfun_, jAvar_, A_);
260     snoptHandler.setG(lenG_, iGfun_, jGvar_);
261     snoptHandler.setX(x_, xlow_, xupp_, xmul_, xstate_);
262     snoptHandler.setF(F_, Flow_, Fupp_, Fmul_, Fstate_);
263     snoptHandler.setXNames(xnames_, nxnames_);
264     snoptHandler.setFNames(Fnames_, nFnames_);
265     snoptHandler.setNeA(neA_);
266     snoptHandler.setNeG(neG_);
267     snoptHandler.setUserFun(SNOPTusrFG3_);
268   }
269
270   void Subproblem::setConstraintsAndDimensions() {
271     if (normType_ == Subproblem::L2_NORM) {
272       m_++;
273     }
274     neF_ = m_ + 1;
275     lenA_ = 0;
276     lenG_ = n_ + m_ * n_;
277     objRow_ = 0; // In theory the objective function could be any of the
      ↪  elements in F.
278     objAdd_ = 0.0;
```

```
279
280   iGfun_ = new integer[lenG_];
281   jGvar_ = new integer[lenG_];
282
283   iAfun_ = NULL;
284   jAvar_ = NULL;
285   A_ = NULL;
286
287   xlow_ = new double[n_];
288   xupp_ = new double[n_];
289
290   Flow_ = new double[neF_];
291   Fupp_ = new double[neF_];
292
293
294   // Objective function
295   Flow_[0] = -infinity_;
296   Fupp_[0] = infinity_;
297
298   // Trust region radius
299   if (normType_ == Subproblem::L2_NORM) {
300     Flow_[1] = 0;
301     Fupp_[1] = trustRegionRadius_;
302   }
303
304   Eigen::VectorXd x_lb = virtualSimulator.GetLowerBoundsForVariables();
305   if (x_lb.rows() < n_) {
306     int tmp1 = x_lb.rows();
307     x_lb.conservativeResize(n_);
308     for (int i = tmp1; i < n_; ++i) {
309       x_lb[i] = -infinity_;
310     }
311   }
312   Eigen::VectorXd x_ub = virtualSimulator.GetUpperBoundsForVariables();
313   if (x_ub.rows() < n_) {
314     int tmp2 = x_ub.rows();
315     x_ub.conservativeResize(n_);
316     for (int i = tmp2; i < n_; ++i) {
317       x_ub[i] = infinity_;
318     }
319   }
320   for (int i = 0; i < n_; ++i) {
321
322     if (std::isinf(x_lb[i])) {
323       xlow_[i] = -infinity_;
324     } else {
325       xlow_[i] = x_lb[i];
326     }
327     if (std::isinf(x_ub[i])) {
328       xupp_[i] = infinity_;
329     } else {
330       xupp_[i] = x_ub[i];
331
332     }
333
334     if (std::abs(x_lb[i] - x_ub[i]) <= 0.00000000001) {
335       xlow_[i] = x_lb[i];
```

```
      xupp_[i] = x_lb[i];
    }

    xlowCopy_[i] = xlow_[i];
    xuppCopy_[i] = xupp_[i];
  }

  int startI = 1;
  if (normType_ == Subproblem::L2_NORM) {
    startI++;
  }
  Eigen::VectorXd g_lb = virtualSimulator.GetLowerBoundsForConstraints();
  Eigen::VectorXd g_ub = virtualSimulator.GetUpperBoundsForConstraints();
  Flow_[startI] = 0;
  Fupp_[startI] = 500;
  startI++;
  for (int i = startI; i < neF_; ++i) {

    if (std::isinf(g_lb[i - startI])) {
      Flow_[i] = -infinity_;
    } else {
      Flow_[i] = g_lb[i - startI];
      smallTightning(Flow_[i], 1);
    }
    if (std::isinf(g_ub[i - startI])) {
      Fupp_[i] = infinity_;
    } else {
      Fupp_[i] = g_ub[i - startI];
      smallTightning(Fupp_[i], 0);

    }

  }

  // first the objective
  for (int i = 0; i < n_; i++) {
    iGfun_[i] = 0;
    jGvar_[i] = i;
  }

  if (m_ != 0) {
    // and then the constraints
    for (int j = 1; j <= m_; j++) {
      for (int i = 0; i < n_; i++) {
        iGfun_[i + j * n_] = j;
        jGvar_[i + j * n_] = i;
      }
    }
  }

  neG_ = lenG_;
  neA_ = lenA_;

}

Subproblem::~Subproblem() {
  delete[] iGfun_;
```

```
393   delete[] jGvar_;
394   delete[] x_;
395   delete[] xlow_;
396   delete[] xupp_;
397   delete[] xmul_;
398   delete[] xstate_;
399   delete[] F_;
400   delete[] Flow_;
401   delete[] Fupp_;
402   delete[] Fmul_;
403 }
404
405 void Subproblem::setOptionsForSNOPT(SNOPTHandler &snoptHandler) {
406
407   //if (settings_->verb_vector()[6] >= 1) // idx:6 -> opt (Optimization)
408   //cout << "[opt]Set options for SNOPT.---" << endl;
409
410   //snoptHandler.setParameter("Backup basis file              0");
411 //  snoptHandler.setRealParameter("Central difference interval", 2 *
    ↪  derivativeRelativePerturbation);
412
413   //snoptHandler.setIntParameter("Check frequency",           60);
414   //snoptHandler.setParameter("Cold Start                    Cold");
415
416   //snoptHandler.setParameter("Crash option                  3");
417   //snoptHandler.setParameter("Crash tolerance               0.1");
418   //snoptHandler.setParameter("Derivative level              3");
419
420 //  if ( (optdata.optimizationType == HISTORY_MATCHING) ||
    ↪  hasNonderivativeLinesearch )
421 //    snoptHandler.setParameter((char*)"Nonderivative linesearch");
422 //  else
423   //snoptHandler.setParameter((char*)"Derivative linesearch");
424   //snoptHandler.setIntParameter("Derivative option", 0);
425
426 //  snoptHandler.setRealParameter("Difference interval",
    ↪  optdata.derivativeRelativePerturbation);
427
428   //snoptHandler.setParameter("Dump file                     0");
429   //snoptHandler.setParameter("Elastic weight                1.0e+4");
430   //snoptHandler.setParameter("Expand frequency              10000");
431   //snoptHandler.setParameter("Factorization frequency       50");
432   //snoptHandler.setRealParameter("Function precision",
      ↪  sim_data.tuningParam.tstep.minDeltat);
433   //snoptHandler.setParameter("Hessian full memory");
434   //snoptHandler.setParameter("Hessian limited memory");
435
436 //  snoptHandler.setIntParameter("Hessian frequency",
    ↪  optdata.frequencyToResetHessian);
437   //snoptHandler.setIntParameter("Hessian updates", 0);
438   //snoptHandler.setIntParameter("Hessian flush", 1);  // Does NOT work
      ↪  in the current version of SNOPT!!!
439
440   //snoptHandler.setParameter("Insert file                   0");
441 //  snoptHandler.setRealParameter("Infinite bound",
    ↪  optdata.defaultControlUpperBound);
442
```

```
443   //snoptHandler.setParameter("Iterations limit");
444   //snoptHandler.setRealParameter("Linesearch tolerance",0.9);
445   //snoptHandler.setParameter("Load file                         0");
446   //snoptHandler.setParameter("Log frequency                    100");
447   //snoptHandler.setParameter("LU factor tolerance              3.99");
448   //snoptHandler.setParameter("LU update tolerance              3.99");
449   //snoptHandler.setRealParameter("LU factor tolerance", 3.99);
450   //snoptHandler.setRealParameter("LU update tolerance", 3.99);
451   //snoptHandler.setParameter("LU partial pivoting");
452   //snoptHandler.setParameter("LU density tolerance             0.6");
453   //snoptHandler.setParameter("LU singularity tolerance       3.2e-11");
454
455   //target nonlinear constraint violation
456   snoptHandler.setRealParameter("Major feasibility tolerance",
      ↪  0.0000001);
457   snoptHandler.setIntParameter("Major Iterations Limit", 1000);
458
459   //target complementarity gap
460   //snoptHandler.setRealParameter("Major optimality tolerance",
      ↪  0.000000000001);
461
462   snoptHandler.setParameter("Major Print level  00000"); //  00001"
463   snoptHandler.setRealParameter("Major step limit", 0.2); //was 0.2
464   //snoptHandler.setIntParameter("Minor iterations limit", 200); // 200
465
466   //for satisfying the QP bounds
467   snoptHandler.setRealParameter("Minor feasibility tolerance", 1.0e-3);
468   snoptHandler.setIntParameter("Minor print level", 0);
469   //snoptHandler.setParameter("New basis file                   0");
470   //snoptHandler.setParameter("New superbasics limit           99");
471   //snoptHandler.setParameter("Objective Row");
472   //snoptHandler.setParameter("Old basis file                   0");
473   //snoptHandler.setParameter("Partial price                    1");
474   //snoptHandler.setParameter("Pivot tolerance                3.7e-11");
475   //snoptHandler.setParameter("Print frequency                  100");
476   //snoptHandler.setParameter("Proximal point method            1");
477   //snoptHandler.setParameter("QPSolver Cholesky");
478   //snoptHandler.setParameter("Reduced Hessian dimension");
479   //snoptHandler.setParameter("Save frequency                   100");
480   snoptHandler.setIntParameter("Scale option", 1);
481   //snoptHandler.setParameter("Scale tolerance                  0.9");
482   snoptHandler.setParameter((char *) "Scale Print");
483   snoptHandler.setParameter((char *) "Solution  Yes");
484   //snoptHandler.setParameter("Start Objective Check at Column 1");
485   //snoptHandler.setParameter("Start Constraint Check at Column 1");
486   //snoptHandler.setParameter("Stop Objective Check at Column");
487   //snoptHandler.setParameter("Stop Constraint Check at Column");
488   //snoptHandler.setParameter("Sticky parameters                 No");
489   //snoptHandler.setParameter("Summary frequency                100");
490   //snoptHandler.setParameter("Superbasics limit");
491   //snoptHandler.setParameter("Suppress parameters");
492   //snoptHandler.setParameter((char*)"System information  No");
493   //snoptHandler.setParameter("Timing level                      3");
494   snoptHandler.setRealParameter("Unbounded objective value",
495                                 1.0e+18); // infinity_ = 1e20; "Unbounded
                                    ↪  objective value   1.0e+15"
```

```
  snoptHandler.setRealParameter("Unbounded step size", 1.0e+19);
   ↪  //"Unbounded step size              1.0e+18"
  //snoptHandler.setIntParameter("Verify level", -1); //-1
  //snoptHandler.setRealParameter("Violation limit", 1e-8); //1e-8

//  if (settings_->verb_vector()[6] >= 1) // idx:6 -> opt (Optimization)
//    cout << "[opt]Set options for SNOPT.---" << endl;

}

/*****************************************************
ADGPRS, version 1.0, Copyright (c) 2010-2015 SUPRI-B
Author(s): Oleg Volkov           (ovolkov@stanford.edu)
           Vladislav Bukshtynov (bukshtu@stanford.edu)
*****************************************************/

bool Subproblem::loadSNOPT(const string libname) {

//#ifdef NDEBUG
  if (LSL_isSNOPTLoaded()) {
    printf("\x1b[33mSnopt is already loaded.\n\x1b[0m");
    return true;
  }

  char buf[256];
  int rc;
  if (libname.empty()) {
    rc = LSL_loadSNOPTLib(NULL, buf, 255);
  } else {
    rc = LSL_loadSNOPTLib(libname.c_str(), buf, 255);
  }

  if (rc) {
    string errmsg;
    errmsg = "Selected NLP solver SNOPT not available.\n"
        "Tried to obtain SNOPT from shared library \"";
    errmsg += LSL_SNOPTLibraryName();
    errmsg += "\", but the following error occured:\n";
    errmsg += buf;
    cout << errmsg << endl;
    return false;
  }
//#endif

  return true;
}

void restrictNumberValueToMatchSNOPT(double &value) {
  if (value == -std::numeric_limits<double>::infinity() || value <=
   ↪  -1e20) {
    value = -1e20;
  } else if (value == std::numeric_limits<double>::infinity() || value >=
   ↪  1e20) {
    value = 1e20;
  }
}

```

```
550 double calculateSplineLenght(Eigen::VectorXd splineIsh) {
551   Eigen::VectorXd wellStart = Eigen::VectorXd::Zero(3);
552   Eigen::VectorXd wellEnd = Eigen::VectorXd::Zero(3);
553
554   for (int i = 0; i < 6; ++i) {
555     splineIsh[i] = splineIsh[i] * scaling[i];
556   }
557   for (int i = 0; i < 3; i++) {
558     wellStart[i] = splineIsh(mapNormalToFieldopt[i]);
559     wellEnd[i] = splineIsh(mapNormalToFieldopt[i + 3]);
560   }
561   return (wellStart - wellEnd).norm();
562 }
563
564 int SNOPTusrFG3_(integer *Status, integer *n, double x[],
565                  integer *needF, integer *neF, double F[],
566                  integer *needG, integer *neG, double G[],
567                  char *cu, integer *lencu,
568                  integer iu[], integer *leniu,
569                  double ru[], integer *lenru) {
570
571   Eigen::VectorXd xvec(*n);
572   for (int i = 0; i < *n; ++i) {
573     xvec[i] = x[i];
574   }
575
576   int m = *neF - 1;
577
578   // If the values for the objective and/or the constraints are desired
579   if (*needF > 0) {
580     /// The objective function
581     F[0] = (constant + gradient.transpose() * xvec + 0.5 *
      ↪  xvec.transpose() * hessian * xvec) / scale;
582     restrictNumberValueToMatchSNOPT(F[0]);
583     if (m) {
584       /// The constraints
585       int startI = 1;
586       if (normType == Subproblem::L2_NORM) {
587         F[1] = (xvec - yb_rel).norm();
588         startI++;
589       }
590
591       F[startI] = calculateSplineLenght(xvec + y0);
592       startI++;
593
594       Eigen::VectorXd constraints =
       ↪  virtualSimulator.evaluateConstraints(xvec + y0);
595       for (int i = 0; i < constraints.rows(); ++i) {
596         F[i + startI] = constraints[i];
597         restrictNumberValueToMatchSNOPT(F[i + startI]);
598       }
599     }
600   }
601
602   if (*needG > 0) {
603     /// The Derivatives of the objective function
604     Eigen::VectorXd grad(*n);
```

```
605       grad = gradient + hessian * xvec;
606       for (int i = 0; i < *n; ++i) {
607         G[i] = grad(i);
608         restrictNumberValueToMatchSNOPT(G[i]);
609       }
610
611       /// The derivatives of the constraints
612       if (m) {
613         Eigen::VectorXd gradConstraint(*n);
614         int startI = *n;
615         if (normType == Subproblem::L2_NORM) {
616           gradConstraint = (xvec - yb_rel) / ((xvec - yb_rel).norm() +
              ↪  0.000000000000001);
617           for (int i = 0; i < *n; ++i) {
618           }
619           startI += *n;
620         }
621         Eigen::MatrixXd constraintsGradients =
              ↪  virtualSimulator.evaluateConstraintGradients(xvec + y0);
622         for (int i = 0; i < constraintsGradients.rows(); ++i) { //for each
              ↪  contraint
623           for (int j = 0; j < constraintsGradients.cols(); ++j) { //for
                ↪  each variable.
624             G[startI + i * (*n) + j] = constraintsGradients(i, j);
625             restrictNumberValueToMatchSNOPT(G[startI + i * (*n) + j]);
626           }
627         }
628       }
629     }
630     return 0;
631   }
632
633   void Subproblem::setQuadraticModel(double c, Eigen::VectorXd g,
       ↪  Eigen::MatrixXd H) {
634     constant = c;
635     gradient = g;
636     hessian = H;
637   }
638
639   void Subproblem::setGradient(Eigen::VectorXd g) {
640     gradient = g;
641   }
642
643   void Subproblem::setHessian(Eigen::MatrixXd H) {
644     hessian = H;
645   }
646
647   void Subproblem::setConstant(double c) {
648     constant = c;
649   }
650   void Subproblem::setNormType(int type) {
651     normType = type;
652   }
653   void Subproblem::setCenterPointOfModel(Eigen::VectorXd cp) {
654     y0 = cp;
655   }
656   void Subproblem::setCurrentBestPointDisplacement(Eigen::VectorXd db) {
```

```
657   bestPointDisplacement_ = db;
658   yb_rel = bestPointDisplacement_;
659 }
660 void Subproblem::SetCenterPoint(Eigen::VectorXd cp) {
661   y0_ = cp;
662   y0 = y0_;
663 }
664 void Subproblem::SetBestPointRelativeToCenterPoint(Eigen::VectorXd bp) {
665
666 }
667 void Subproblem::printModel() {
668   std::cout << "The model of the subproblem \n";
669   std::cout << "constant = \n" << constant << std::endl;
670   std::cout << "gradient = \n" << gradient << std::endl;
671   std::cout << "hessian = \n" << hessian << std::endl;
672   std::cout << "trust region radius = " << trustRegionRadius_ <<
      ↪  std::endl;
673
674 }
675 Eigen::VectorXd Subproblem::GetInitialPoint() {
676
677   return virtualSimulator.GetInitialPoint();
678 }
679 Eigen::MatrixXd Subproblem::getGradientConstraints(Eigen::VectorXd point)
    ↪  {
680   return virtualSimulator.evaluateConstraintGradients(point);
681 }
682 int Subproblem::getNumberOfConstraints() {
683   return virtualSimulator.GetNumberOfConstraints();
684 }
685 void Subproblem::SolveVirtualSimulator() {
686   virtualSimulator.Solve();
687 }
688 Eigen::VectorXd Subproblem::FindFeasiblePoint() {
689   hessian = Eigen::MatrixXd::Zero(n_, n_);
690   gradient = Eigen::VectorXd::Zero(n_);
691   constant = 1;
692   yb_rel = Eigen::VectorXd::Zero(n_);
693   // The snoptHandler must be setup and loaded
694   SNOPTHandler snoptHandler = initSNOPTHandler();
695   snoptHandler.setProbName("SNOPTSolver");
696   snoptHandler.setParameter("Feasible point");
697   snoptHandler.initializeLagrangeVector(neF_ - 1);
698
699   setOptionsForSNOPT(snoptHandler);
700   snoptHandler.setRealParameter("Major step limit", 0.2); //was 0.2
701   snoptHandler.setRealParameter("Major feasibility tolerance", 1.0e-6);
      ↪  //1.0e-6
702
703   ResetSubproblem();
704   for (int i = 0; i < n_; i++) {
705     //x_[i] = startingPoint[i];//bestPointDisplacement_[i];
706     //x_[i] = 0.0;//bestPointDisplacement_[i];
707     //x_[i] = startingPoint[i];
708   }
709
710   if (normType_ == Subproblem::L2_NORM) {
```

```
    Flow_[1] = 0;
    Fupp_[1] = trustRegionRadius_;
  }
  passParametersToSNOPTHandler(snoptHandler);
  integer Cold = 0, Basis = 1, Warm = 2;

  vector<double> xsol;
  vector<double> fsol;
  snoptHandler.solve(Cold, xsol, fsol);
  lastLagrangeMultipliers = snoptHandler.getLagrangeMultipliers();
  fsol[0] = fsol[0] * scale;
  integer exitCode = snoptHandler.getExitCode();
  if (exitCode != 40 && exitCode != 41 && exitCode != 1 && exitCode != 31
   ↪  && exitCode != 3 && exitCode != 32) {
    std::cout << "ExitCode is: " << exitCode << "\n";
    std::cin.get();
  }

  Eigen::VectorXd xvec(n_);
  for (int i = 1; i <= n_; ++i) {
    xvec(i - 1) = xsol[i - 1];
  }
  return xvec;
}
void Subproblem::evaluateConstraints(Eigen::VectorXd point) {
  if (virtualSimulator.GetNumberOfConstraints() > 0) {
    auto d = virtualSimulator.evaluateConstraints(point + y0);
  }

}

void Subproblem::calculateLagrangeMultipliers(char *optimizationType,
                                              VectorXd centerPoint,
                                              VectorXd
                                               ↪  bestPointDisplacement,
                                              VectorXd startingPoint) {

  y0_ = centerPoint;
  y0 = y0_;
  bestPointDisplacement_ = bestPointDisplacement;
  yb_rel = bestPointDisplacement_;
  if (normType_ == INFINITY_NORM) {
    for (int i = 0; i < n_; ++i) {
      xlow_[i] = std::max(bestPointDisplacement_[i] - trustRegionRadius_,
       ↪  xlowCopy_[i] - y0_[i]);
      xupp_[i] = std::min(bestPointDisplacement_[i] + trustRegionRadius_,
       ↪  xuppCopy_[i] - y0_[i]);
    }
  } else if (normType_ == L2_NORM) {
    for (int i = 0; i < n_; ++i) {
      xlow_[i] = xlowCopy_[i];
      xupp_[i] = xuppCopy_[i];
    }
  }

  scale = 1;

```

```
764
765   // The snoptHandler must be setup and loaded
766   SNOPTHandler snoptHandler = initSNOPTHandler();
767   snoptHandler.setProbName("SNOPTSolver");
768   snoptHandler.setParameter(optimizationType);
769   snoptHandler.initializeLagrangeVector(neF_ - 1);
770
771   setOptionsForSNOPT(snoptHandler);
772   snoptHandler.setIntParameter("Major Iterations Limit", 2);
773   snoptHandler.setIntParameter("Iterations limit", 2);
774   snoptHandler.setRealParameter("Major step limit", trustRegionRadius_ *
      ↪  0.0001); //was 0.2
775   snoptHandler.setRealParameter("Major feasibility tolerance", 1.0e-9);
      ↪  //1.0e-6
776
777   snoptHandler.setRealParameter("Major optimality tolerance", 0.00001);
778   snoptHandler.setIntParameter("Minor iterations limit", 1); // 200
779
780   ResetSubproblem();
781   for (int i = 0; i < n_; i++) {
782     //x_[i] = startingPoint[i];//bestPointDisplacement_[i];
783     x_[i] = bestPointDisplacement_[i];
784     //x_[i] = startingPoint[i];
785   }
786   if (normType_ == Subproblem::L2_NORM) {
787     Flow_[1] = 0;
788     Fupp_[1] = trustRegionRadius_;
789   }
790   passParametersToSNOPTHandler(snoptHandler);
791   integer Cold = 0, Basis = 1, Warm = 2;
792
793   vector<double> xsol;
794   vector<double> fsol;
795   snoptHandler.solve(Cold, xsol, fsol);
796   lastLagrangeMultipliers = snoptHandler.getLagrangeMultipliers();
797
798   fsol[0] = fsol[0] * scale;
799   integer exitCode = snoptHandler.getExitCode();
800
801   if (exitCode != 40 && exitCode != 41 && exitCode != 1 && exitCode != 31
      ↪  && exitCode != 3 && exitCode != 32) {
802     std::cout << "ExitCode is: " << exitCode << "\n";
803     std::cin.get();
804   }
805
806   Eigen::VectorXd xvec(n_);
807   for (int i = 1; i <= n_; ++i) {
808     xvec(i - 1) = xsol[i - 1];
809   }
810   if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
811     std::cout << "output from snopt is infeasible. ExitCode was: " <<
      ↪  exitCode << "\n";
812     auto d = virtualSimulator.evaluateConstraints(xvec + y0);
813     std::cout << d << "\n";
814   }
815
816   if (virtualSimulator.IsFeasiblePoint(xvec + y0) == false) {
```

```
    std::cout << "Failed to find feasible points...\n";
  }

}
bool Subproblem::isPointFeasible(Eigen::VectorXd point) {
  return virtualSimulator.IsFeasiblePoint(point + y0);
}
void Subproblem::SetMappingVariables(Eigen::VectorXi map1,
 ↪  Eigen::VectorXd scaling1) {
  mapNormalToFieldopt = map1;
  scaling = scaling1;
}
}
}
```

GradientEnhancedModel.h

```
//
// Created by joakim on 24.04.18.
//

#ifndef FIELDOPT_GRADIENTENHANCEDMODEL_H
#define FIELDOPT_GRADIENTENHANCEDMODEL_H

#include <Eigen/Dense>
#include "FieldOpt-3rdPartySolvers/handlers/SNOPTHandler.h"
#include "FieldOpt-3rdPartySolvers/handlers/SNOPTLoader.h"
#include <QList>
#include <QString>
#include <QStringList>
#include <QJsonObject>
#include <Qt>

namespace Optimization {
namespace Optimizers {

class GradientEnhancedModel {
 private:
  Eigen::VectorXd h_old_;
  double constant_;
  Eigen::VectorXd gradient_;
  Eigen::MatrixXd hessian_;
  Eigen::MatrixXd hessian_old_;
  Eigen::MatrixXd points_;
  Eigen::VectorXd funcVals_;
  Eigen::MatrixXd v_;
  Eigen::VectorXd y0_;
  Eigen::MatrixXd D_;

  double alpha_;
  Eigen::VectorXd weights_derivatives_;
  Eigen::VectorXd weights_least_square_;
  Eigen::VectorXd best_point_;
  int n_;
  int m_;
  int ng_; // number_of_variables_with_gradient
```

```
40
41
42    void solveLinearSystem(Eigen::VectorXd funcVals, Eigen::VectorXd
      ↪  derivatives_at_y0, Eigen::VectorXd &ans);
43    int convert_h_ij_to_t_lsq(int i, int j);
44    int convert_h_ij_to_t_vectorized(int i, int j);
45    void convert_t_to_ij_lsq(int t, int &i, int &j);
46    void convert_t_to_ij_vectorized(int t, int &i, int &j);
47
48   public:
49    GradientEnhancedModel(int n,
50                          int m,
51                          int number_of_variables_with_gradient,
52                          QList<double> weights_derivatives,
53                          double weight_objective_minimum_change);
54    GradientEnhancedModel() {};
55    void GetConstant(double &c);
56    void GetGradient(Eigen::VectorXd &g);
57    void GetHessian(Eigen::MatrixXd &H);
58    void GetModel(double &c, Eigen::VectorXd &g, Eigen::MatrixXd &H);
59    void ComputeModel(Eigen::MatrixXd Y,
60                      Eigen::MatrixXd derivatives,
61                      Eigen::VectorXd derivatives_at_y0,
62                      Eigen::VectorXd funcVals,
63                      Eigen::VectorXd y0,
64                      Eigen::VectorXd best_point,
65                      double radius,
66                      double scaling_factor_r,
67                      int index_of_center_point);
68    void PrintParametersMatlabFriendly();
69
70    void isInterpolating();
71  };
72  }
73  }
74  #endif //FIELDOPT_GRADIENTENHANCEDMODEL_H
```

GradientEnhancedModel.cpp

```
1   //
2   // Created by joakim on 24.04.18.
3   //
4
5   #include <iostream>
6   #include "GradientEnhancedModel.h"
7   #include "EigenUtil.h"
8
9   namespace Optimization {
10  namespace Optimizers {
11
12  static Eigen::VectorXd _y0_;
13  static Eigen::MatrixXd _hessian_old_;
14  static Eigen::MatrixXd _D_;
15  static Eigen::MatrixXd _v_;
16  static double _alpha_;
17  static Eigen::VectorXd _weights_least_square_;
```

```
18  static Eigen::MatrixXd _points_;
19  static Eigen::VectorXd _best_point_;
20  static int _n_;
21  static int _ng_;
22  static int _m_;
23
24  void GradientEnhancedModel::GetConstant(double &c) {
25    c = constant_;
26  }
27  void GradientEnhancedModel::GetGradient(Eigen::VectorXd &g) {
28    g = gradient_;
29  }
30  void GradientEnhancedModel::GetHessian(Eigen::MatrixXd &H) {
31    H = hessian_;
32  }
33  void GradientEnhancedModel::GetModel(double &c, Eigen::VectorXd &g,
     ↪  Eigen::MatrixXd &H) {
34    c = constant_;
35    g = gradient_;
36    H = hessian_;
37  }
38  GradientEnhancedModel::GradientEnhancedModel(
39      int n, int m, int number_of_variables_with_gradient,
40      QList<double> weights_derivatives,
41      double weight_objective_minimum_change) {
42
43    _n_ = n;
44    n_ = n;
45    _m_ = m;
46    m_ = m;
47    _ng_ = number_of_variables_with_gradient;
48    ng_ = number_of_variables_with_gradient;
49    alpha_ = weight_objective_minimum_change;
50    weights_derivatives_ =
       ↪  Eigen::VectorXd::Zero(weights_derivatives.size());
51    int j = 0;
52    for (auto i = weights_derivatives.begin(); i !=
       ↪  weights_derivatives.end(); ++i) {
53      weights_derivatives_[j] = *i;
54      j++;
55    }
56    constant_ = 0;
57    gradient_ = Eigen::VectorXd::Zero(n_);
58    hessian_ = Eigen::MatrixXd::Zero(n_, n_);
59    hessian_old_ = Eigen::MatrixXd::Zero(n_, n_);
60    points_ = Eigen::MatrixXd::Zero(n_, m_);
61    funcVals_ = Eigen::VectorXd::Zero(n_);
62    v_ = Eigen::VectorXd::Zero(ng_ * m_);
63    y0_ = Eigen::VectorXd::Zero(n_);
64    best_point_ = Eigen::VectorXd::Zero(n_);
65    weights_least_square_ = Eigen::VectorXd::Zero(m_);
66
67    int t = 0;
68    int y = n_;
69    for (int i = 0; i < ng_; ++i) {
70      t += y;
71      y--;
```

```
  }
  h_old_ = Eigen::VectorXd::Zero(t);
  D_ = Eigen::MatrixXd::Zero((ng_) * m_, t);

}

void GradientEnhancedModel::ComputeModel(Eigen::MatrixXd Y,
                                         Eigen::MatrixXd derivatives,
                                         Eigen::VectorXd
                                          ↪  derivatives_at_y0,
                                         Eigen::VectorXd funcVals,
                                         Eigen::VectorXd y0,
                                         Eigen::VectorXd best_point,
                                         double radius, double
                                          ↪  scaling_factor_r,
                                         int index_of_center_point) {
  funcVals_ = funcVals;
  points_ = Y;

  y0_ = y0;
  best_point_ = best_point;

  // Set up _weights_least_square_
  for (int t = 0; t < m_; ++t) {
    double norm = (Y.col(t) - best_point).norm();
    if (norm <= radius) {
      weights_least_square_[t] = weights_derivatives_[0];
    } else if (norm <= scaling_factor_r * radius) {
      weights_least_square_[t] = weights_derivatives_[1];
    } else {
      weights_least_square_[t] = weights_derivatives_[2];
    }
  }

  // Set the _D_ matrix and the _v_ vector
  if (ng_ > 0) {
    int base_row = 0;
    for (int t = 0; t < m_; ++t) { // for each sample point

      int y = n_;
      int c0 = 0;
      for (int i = 0; i < ng_; ++i) { // for each row
        for (int k = 0; k < y; ++k) { // for each elem in row
          D_(base_row + i, c0 + k) = points_(k, t) *
           ↪  weights_least_square_[t];
        }
        c0 += y;
        y--;
      }
      y = n_ - 1;
      c0 = 0;
      for (int k = 0; k < (ng_ - 1); ++k) { // for each row that begins
       ↪  an inverse diagonal
        int ii = 1;
        int jj = 1;
        for (int i = k; i < (ng_ - 1); ++i) {//for each element in that
         ↪  inverse diagonal
```

```
124             D_(base_row + k + ii, c0 + y - jj) = points_(y, t) *
                ↪  weights_least_square_[t];
125             ii++;
126             jj++;
127           }
128           c0 += (y + 1);
129           y--;
130         }
131         base_row += ng_;
132
133         for (int i = 0; i < ng_; ++i) {
134           v_(t * ng_ + i) = (derivatives((ng_ - i - 1), t) -
              ↪  derivatives_at_y0(ng_ - i - 1)) * weights_least_square_[t];
135         }
136       }
137     }
138
139
140     // set the constraints
141     Eigen::VectorXd ans;
142     solveLinearSystem(funcVals, derivatives_at_y0, ans);
143
144     // calculate start indices;
145     int start_h_ij = 0;
146     int start_g_i = (int) ((n_ * n_ + n_) * 0.5);
147     int start_c = start_g_i + (n_ - ng_);
148     int start_lambda_i = start_c + 1;
149
150     //extract the results
151     constant_ = ans(start_c);
152     for (int i = 0; i < n_ - ng_; ++i) {
153       gradient_[i] = ans(start_g_i + i);
154     }
155     eigen_tail(gradient_, derivatives_at_y0, ng_);
156     for (int i = 1; i <= (int) ((n_ * n_ + n_) * 0.5); ++i) {
157       int ii = 0;
158       int jj = 0;
159       convert_t_to_ij_vectorized(i, ii, jj);
160
161       hessian_(ii - 1, jj - 1) = ans(start_h_ij + i - 1);
162       if (ii != jj) {
163         hessian_(jj - 1, ii - 1) = hessian_(ii - 1, jj - 1);
164       }
165     }
166
167     hessian_old_ = hessian_;
168   }
169
170   void GradientEnhancedModel::solveLinearSystem(Eigen::VectorXd funcVals,
171                                                 Eigen::VectorXd
                                                  ↪  derivatives_at_y0,
172                                                 Eigen::VectorXd &ans) {
173     int colsD = 0;
174     int y = n_;
175     for (int i = 0; i < ng_; ++i) {
176       colsD += y;
177       y--;
```

```
178   }
179   Eigen::MatrixXd A = Eigen::MatrixXd::Zero((int) (1 + (n_ - ng_) + m_ +
      ↪  (n_ * n_ + n_) * 0.5),
180                                             (int) (1 + (n_ - ng_) + m_ +
                                                ↪  (n_ * n_ + n_) * 0.5));
181   Eigen::VectorXd b = Eigen::VectorXd::Zero((int) (1 + (n_ - ng_) + m_ +
      ↪  (n_ * n_ + n_) * 0.5));
182   // calculate start indices;
183   int start_h_ij = 0;
184   int start_g_i = (int) ((n_ * n_ + n_) * 0.5);
185   int start_c = start_g_i + (n_ - ng_);
186   int start_lambda_i = start_c + 1;
187   int row = 0;
188
189
190
191
192   // dL/dhij,
193   for (int i = 1; i <= n_; ++i) {
194     for (int j = 1; j <= i; ++j) { // the derivative with respect to
       ↪  (i,j)
195
196       b(row) += hessian_old_(i - 1, j - 1) * alpha_; //right hand side
197       A(row, convert_h_ij_to_t_vectorized(i, j) - 1) = alpha_;
198
199       for (int t = 1; t <= m_; ++t) {
200         A(row, start_lambda_i + t - 1) += -0.5 * points_(i - 1, t - 1) *
          ↪  points_(j - 1, t - 1);
201       }
202
203       if (i > (n_ - ng_)) {
204         for (int p = 1; p <= ng_ * m_; p++) {
205           int t = convert_h_ij_to_t_lsq(i, j); // taking derivative
            ↪  w.r.t. h_t
206
207           b(row) += (1 - alpha_) * v_(p - 1) * D_(p - 1, t - 1); // right
            ↪  hand side
208
209           for (int k = 1; k <= colsD; ++k) {
210             int ii = 0;
211             int jj = 0;
212
213             convert_t_to_ij_lsq(k, ii, jj);
214             A(row, convert_h_ij_to_t_vectorized(ii, jj) - 1) += (1 -
              ↪  alpha_) * D_(p - 1, k - 1) * D_(p - 1, t - 1);
215           }
216         }
217
218       }
219
220       row++;
221     }
222   }
223   // dL/dgi,
224   for (int i = 1; i <= (n_ - ng_); ++i) {
225     for (int t = 1; t <= m_; ++t) {
226       A(row, start_lambda_i + t - 1) += points_(i - 1, t - 1);
```

```
227       }
228       row++;
229     }
230     // dL/dc
231     for (int t = 1; t <= m_; ++t) {
232       A(row, start_lambda_i + t - 1) = 1;
233     }
234     row++;
235
236     //interpolation conditions
237     for (int k = 1; k <= m_; ++k) {
238       b(row) = funcVals(k - 1) - derivatives_at_y0.dot((points_.col(k -
          ↪  1)).tail(ng_));
239       A(row, start_c) = 1;
240
241       for (int t = 1; t <= (n_ - ng_); ++t) {
242         A(row, start_g_i + t - 1) = points_(t - 1, k - 1);
243       }
244
245       for (int i = 1; i <= n_; ++i) {
246         for (int j = 1; j <= i; ++j) {
247           if (i == j) {
248             A(row, start_h_ij + convert_h_ij_to_t_vectorized(i, j) - 1) =
249                 0.5 * points_(i - 1, k - 1) * points_(i - 1, k - 1);
250           } else {
251             A(row, start_h_ij + convert_h_ij_to_t_vectorized(i, j) - 1) =
              ↪  points_(i - 1, k - 1) * points_(j - 1, k - 1);
252           }
253         }
254       }
255
256       row++;
257     }
258     ans = A.colPivHouseholderQr().solve(b);
259   }
260
261   int GradientEnhancedModel::convert_h_ij_to_t_lsq(int i, int j) {
262     int t = 0;
263     for (int p = n_; p > i; --p) {
264       t += p;
265     }
266     t += j;
267     return t;
268   }
269
270   int GradientEnhancedModel::convert_h_ij_to_t_vectorized(int i, int j) {
271     int t = 0;
272     int y = n_;
273     for (int k = 1; k < j; ++k) {
274       t += y;
275       y--;
276     }
277     t += (i - j + 1);
278     return t;
279   }
280
281   void GradientEnhancedModel::convert_t_to_ij_lsq(int t, int &i, int &j) {
```

```
  i = n_;
  j = 0;
  for (i = n_; t - i > 0; --i) {
    t -= i;
  }
  j = t;
}

void GradientEnhancedModel::convert_t_to_ij_vectorized(int t, int &i, int
 ↪  &j) {
  j = 1;
  for (int k = n_; t > k; k--) {
    t = t - k;
    j++;
  }
  i = j + (t - 1);
}

void GradientEnhancedModel::isInterpolating() {
  std::cout << "Checking if the gradient enhanced model actually
   ↪  interpolates the points\n";
  for (int i = 1; i <= m_; ++i) {
    double val = constant_ + gradient_.transpose() * points_.col(i - 1)
        + 0.5 * (points_.col(i - 1)).transpose() * hessian_old_ *
         ↪  points_.col(i - 1);
    std::cout << funcVals_[i - 1] << " == " << val << "\t" <<
     ↪  (std::abs(val - funcVals_[i - 1]) <= 0.000000001) << "\n";
  }
}

}
}
```

DFO.h

```
//
// Created by pcg1 on 12.01.18.
//

#ifndef FIELDOPT_DFO_H
#define FIELDOPT_DFO_H

#include "Optimization/optimizer.h"
#include "Subproblem.h"
#include "DFO_Model.h"
#include <ncurses.h>
#include <fstream>
#include <iostream>

namespace Optimization {
namespace Optimizers {

/*!
 * @brief This is a fantastic description of the DFO method, with
 ↪  references.
 */
```

```
class DFO : public Optimizer {
 public:
  DFO(Settings::Optimizer *settings,
      Case *base_case,
      Model::Properties::VariablePropertyContainer *variables,
      Reservoir::Grid::Grid *grid,
      Logger *logger);

  QString GetStatusStringHeader() const {};
  QString GetStatusString() const {};
  Model::Properties::VariablePropertyContainer *variables_;
  void set_next_step(int a);

  Eigen::VectorXd
   ↪  ScaleVariablesFromAlgorithmToApplication(Eigen::VectorXd point);
  Eigen::VectorXd
   ↪  ScaleVariablesFromApplicationToAlgorithm(Eigen::VectorXd point);
  void ConvertPointToCase(Eigen::VectorXd point, Optimization::Case
   ↪  *new_case);

  void CreateScalingVector();

 private:

  QList<QUuid> UUIDS_in_same_order_as_in_realvar_base_case_;
  bool terminated = false;
  Eigen::VectorXd realvectest_;
  QUuid mapIndexToCase;
  vector<QUuid> mapIndicesToCase;
  Eigen::VectorXd scaling_;

  std::string filenamePoint;
  std::string filenameType;
  std::string filenameTrr;

  std::string color_from = "31";
  std::string color_to = "33";
  Model::Properties::VariablePropertyContainer *varcont_;
  DFO_Model DFO_model_;
  void iterate() override;
  int number_of_interpolation_points_;
  int number_of_variables_;
  Optimization::Case *base_case_;
  int last_action_;
  int next_step_;
  std::string get_action_name(int a);
  Eigen::VectorXd weights_distance_from_optimum_lsq_;

  int number_of_points_first_set;
  bool multiple_new_points;
  double trust_region_radius_tilde;
  double trust_region_radius_icb;
  double rho;
  bool is_model_CFL;
  Eigen::VectorXd tmp_eval;

  int number_of_crit_step_finished_with_bad_poisedness;
```

```
  int number_of_crit_step_finished_without_checking_poisedness;
  int ng;
  double alpha;
  int number_of_tiny_improvements;
  double r;
  double w;
  double u;
  double beta;
  double epsilon_c;
  double tau;
  double eta1;
  double gamma;
  double gamma_inc;
  double trust_region_radius_inc;
  double trust_region_radius_max;
  double trust_region_radius_end;

  bool notConverged;

  int crit_steps;
  int accept_steps;
  int model_impr_steps;

  Eigen::VectorXd *refFuncVals;
  Eigen::MatrixXd *refY;
  Eigen::MatrixXd *refDerivatives;

  int number_of_new_points;
  int number_of_function_calls;
  int number_of_parallell_function_calls;
  Eigen::MatrixXd new_gradients;
  Eigen::MatrixXd new_points;
  Eigen::VectorXd new_point;
  Eigen::VectorXd last_trial_point;
  Eigen::VectorXd new_gradient;
  Eigen::VectorXi new_points_indicies;
  Eigen::VectorXd function_evaluations;
  double function_evaluation;
  int new_point_index;
  int criticality_step_iteration;

  bool force_criticality_step;

  bool is_successful_iteration() {};

  void handleEvaluatedCase(Case *c);

  TerminationCondition IsFinished() {
    cout << "JUST CALLED ISFINISHED" << endl;
    return TerminationCondition::NOT_FINISHED;
  };

 protected:

 private:
  int previous_iterate_type_;
  double initial_trust_region_radius_;
```

```
  double required_poisedness_;
  QList<QUuid> realvar_uuid_;
  Settings::Optimizer *settings_;
  int iterations_;

};
}
}

#endif //FIELDOPT_DFO_H
```

DFO.cpp

```
//
// Created by pcg1 on 12.01.18.
//

#include "DFO.h"
#include "GradientEnhancedModel.h"
#include "VirtualSimulator.h"
#include <iostream>
#include <iomanip>
#include <casadi/casadi.hpp>

#define FIND_POINTS1 0
#define FIND_POINTS2 1
#define INITIALIZE_MODEL 2
#define CRITICALITY_STEP_START 3
#define CRITICALITY_STEP_CHECK_CONVERGENCE 4
#define FIND_TRIAL_POINT 5
#define ACCEPTANCE_OF_TRIAL_POINT 6
#define MODEL_IMPROVEMENT_STEP_START 7
#define MODEL_IMPROVEMENT_STEP_END 8
#define TRUST_REGION_RADIUS_UPDATE_STEP 9
#define CRITICALITY_STEP_ADD_POINTS 10

VirtualSimulator vs;



Eigen::VectorXd evaluateFunctionVS(Eigen::VectorXd x, int ng) {
  Eigen::VectorXd result(1 + ng);
  result(0) = vs.evaluateFunction(x);

  Eigen::VectorXd gradients1 = vs.evaluateFunctionGradients(x);
  Eigen::VectorXd grads = gradients1.tail(ng);
  for (int i = 0; i < grads.rows(); ++i) {
    result(i + 1) = grads(i);
  }

  return result;
}



```

```
44  namespace Optimization {
45  namespace Optimizers {
46
47  DFO::DFO(Settings::Optimizer *settings,
48           Optimization::Case *base_case,
49           Model::Properties::VariablePropertyContainer *variables,
50           Reservoir::Grid::Grid *grid,
51           Logger *logger)
52      : Optimizer(settings, base_case, variables, grid, logger),
53        DFO_model_(
54            settings->parameters().number_of_interpolation_points,
55            settings->parameters().number_of_variables,
56            settings->parameters().number_of_variables_with_gradients,
57            base_case->GetRealVarVector(),
58            settings->parameters().initial_trust_region_radius,
59            settings->parameters().required_poisedness,
60            settings->parameters().
61            weight_model_determination_minimum_change_hessian,
62            settings->parameters().weights_distance_from_optimum_lsq,
63            settings) {
64    // Set parameters
65    if (settings->parameters().initial_trust_region_radius > 0.0)
66      initial_trust_region_radius_ =
        ↪  settings->parameters().initial_trust_region_radius;
67    else
68      initial_trust_region_radius_ = 600;
69
70    if (settings->parameters().number_of_interpolation_points > 0)
71      number_of_interpolation_points_ =
        ↪  settings->parameters().number_of_interpolation_points;
72    else
73      number_of_interpolation_points_ = 21;
74
75    if (settings->parameters().number_of_variables > 0)
76      number_of_variables_ = settings->parameters().number_of_variables;
77    else
78      number_of_variables_ = 10;
79
80    if (settings->parameters().required_poisedness > 0)
81      required_poisedness_ = settings->parameters().required_poisedness;
82    else
83      required_poisedness_ = 5;
84
85    settings_ = settings;
86    varcont_ = variables;
87    iterations_ = 0;
88    previous_iterate_type_ = 0;
89    base_case_ = new Case(base_case);
90    variables_ = variables;
91    base_case_->GetRealVarVector();
92    base_case_->GetRealVarIdVector();
93    last_action_ = -1;
94    weights_distance_from_optimum_lsq_ =
95        Eigen::VectorXd::Zero(settings->
96            parameters().weights_distance_from_optimum_lsq.size());
97    int j = 0;
```

```
98    for (auto i =
      ↪  settings->parameters().weights_distance_from_optimum_lsq.begin();
99         i !=
           ↪  settings->parameters().weights_distance_from_optimum_lsq.end();
           ↪  ++i) {
100     weights_distance_from_optimum_lsq_[j] = *i;
101     j++;
102     if (j >= 3) {
103       break;
104     }
105   }
106
107   UUIDS_in_same_order_as_in_realvar_base_case_ =
      ↪  base_case_->GetRealVarIdVector();
108   CreateScalingVector();
109   Eigen::VectorXd initialStartPoint = base_case->GetRealVarVector();
110   auto as = base_case->GetRealVarIdVector(); //QList<QUuid>
111   auto aa = base_case->real_variables(); // QHash<QUuid, double>
112   Eigen::VectorXd realvec = base_case_->GetRealVarVector();
113   DFO_model_.setInitialy0(ScaleVariablesFromApplicationToAlgorithm(
      ↪  base_case_->GetRealVarVector()));
114   vs = VirtualSimulator(settings_->parameters().test_problem_file);
115   DFO_model_.initLagrangeMultipliers(vs.GetNumberOfConstraints());
116   number_of_crit_step_finished_with_bad_poisedness = 0;
117   number_of_crit_step_finished_without_checking_poisedness = 0;
118   ng = settings_->parameters().number_of_variables_with_gradients;
119   alpha = settings_->parameters().
120       weight_model_determination_minimum_change_hessian;
121   QList<Case *> newCases = case_handler_->RecentlyEvaluatedCases();
122   number_of_tiny_improvements = 0;
123   r = settings_->parameters().r;
124   w = settings_->parameters().w;
125   u = settings_->parameters().u;
126   beta = settings_->parameters().beta;
127   epsilon_c = settings_->parameters().epsilon_c;
128   tau = settings_->parameters().tau;
129   eta1 = settings_->parameters().eta1;
130   gamma = settings_->parameters().gamma;
131   gamma_inc = settings_->parameters().gamma_inc;
132   trust_region_radius_inc =
      ↪  settings_->parameters().initial_trust_region_radius;
133   trust_region_radius_max =
      ↪  settings_->parameters().max_trust_region_radius;
134   trust_region_radius_end =
      ↪  settings_->parameters().end_trust_region_radius;
135   notConverged = true;
136   crit_steps = 0;
137   accept_steps = 0;
138   model_impr_steps = 0;
139   refFuncVals = DFO_model_.getFvalsReference();
140   refY = DFO_model_.getYReference();
141   refDerivatives = DFO_model_.getDerivativeReference();
142   for (int i = 0; i < number_of_interpolation_points_; ++i) {
143     (*refFuncVals)(i) = -1;
144   }
145   number_of_new_points = 0;
146   number_of_function_calls = 0;
```

```
147   number_of_parallell_function_calls = 0;
148   last_trial_point = Eigen::VectorXd::Zero(number_of_variables_);
149   new_point = Eigen::VectorXd::Zero(number_of_variables_);
150   function_evaluations =
      ↪  Eigen::VectorXd::Zero(number_of_interpolation_points_);
151   function_evaluations.setZero();
152   new_point_index = -1;
153   criticality_step_iteration = 0;
154   force_criticality_step = false;
155   {
156     Eigen::VectorXd tmp = DFO_model_.getCenterPoint();
157     for (int i = 0; i < number_of_variables_; ++i) {
158       last_trial_point(i) = tmp(i) + DFO_model_.GetTrustRegionRadius() *
          ↪  (2 * r) * 10; //far outside
159     }
160   }
161   next_step_ = FIND_POINTS1;
162   is_model_CFL = false; //false means we don't know.
163   trust_region_radius_tilde =
      ↪  settings_->parameters().initial_trust_region_radius;;
164   trust_region_radius_icb =
      ↪  settings_->parameters().initial_trust_region_radius;;
165   rho = 0;
166   multiple_new_points = false;
167   number_of_points_first_set = 0;
168   Eigen::VectorXd realvars = base_case_->GetRealVarVector();
169   DFO_model_.findVariableMeaning(realvars , scaling_);
170 }
171 void DFO::handleEvaluatedCase(Optimization::Case *c) {
172 }
173
174 void DFO::iterate() {
175   if (terminated){
176     return;
177   }
178
179   if (iterations_ != 0){
180     //Extract information from the new cases.
181     if (multiple_new_points) {
182       QList<Case *> cs = case_handler_->RecentlyEvaluatedCases();
183       for( int i=0; i<cs.count(); ++i )
184       {
185         for (int j = 0; j < number_of_new_points; j++) {
186           if (cs[i]->GetId() == mapIndicesToCase[j] ) {
187             function_evaluations[j] =
                ↪  -1*(cs[i]->objective_function_value());
188             break;
189           }
190         }
191       }
192
193     } else {
194       QList<Case *> c = case_handler_->RecentlyEvaluatedCases();
195       function_evaluation = -1*(c[0]->objective_function_value());
196     }
197
198     case_handler_->ClearRecentlyEvaluatedCases();
```

```
199   }
200
201
202   print:
203   std::cout << "\033[1;34;m " << " ---------- New iterate " <<
      ↪  iterations_ << " ---------- " << "\033[0m"
204             << std::endl;
205   std::cout << "\033[1;34;m " << "Fvals = \n" << "\033[0m" <<
      ↪  *refFuncVals << "\n";
206   std::cout << "\033[1;34;m " << "Y = \n" << "\033[0m" << *refY << "\n";
207   if (iterations_ != 0 && iterations_ != 1 && iterations_ != 2) {
208     std::cout << "\033[1;34;m " << "Best point index = \n" << "\033[0m"
        ↪  << DFO_model_.getBestPointIndex() << "\n";
209
210     DFO_model_.PrintSortedBestPoint(scaling_);
211   }
212   std::cout << "\033[1;34;m " << "y0 = \n" << "\033[0m" <<
      ↪  DFO_model_.getCenterPoint() << "\n";
213   std::cout << "\033[1;34;m " << "Trust region radius is: " << "\033[0m"
      ↪  << DFO_model_.GetTrustRegionRadius()
214             << std::endl;
215   std::cout << "\033[1;34;m " << "Trust region radius Tilde is: " <<
      ↪  "\033[0m" << trust_region_radius_tilde
216             << std::endl;
217
218
219   top:
220   if (next_step_ == FIND_POINTS1) {
221     new_points = DFO_model_.findFirstSetOfInterpolationPoints();
222     std::cout << "Y new points:\n" << new_points <<"\n";
223     std::cout << "Y:\n" << *refY << "\n";
224     new_points_indicies.resize(new_points.cols());
225     number_of_points_first_set = new_points.cols();
226
227     multiple_new_points = true;
228     set_next_step(FIND_POINTS2);
229     goto evaluate;
230   } else if (next_step_ == FIND_POINTS2) {
231
232
233
234     // add the points.
235     for (int i = 0; i < number_of_new_points; ++i) {
236       //DFO_model_.SetFunctionValue(i + 1, function_evaluations[i]);
237       DFO_model_.SetFunctionValueAndDerivatives(i + 1,
        ↪  function_evaluations(i), new_gradients.col(i));
238     }
239
240     set_next_step(INITIALIZE_MODEL);
241     // find the remaining points.
242     if (number_of_interpolation_points_ == number_of_function_calls) {
243       // All points are found.
244       number_of_new_points = 0;
245       goto print;
246     } else {
247       number_of_new_points = number_of_interpolation_points_ -
        ↪  number_of_function_calls;
```

```
248       new_points_indicies.resize(number_of_new_points);
249       new_points.resize(number_of_variables_, number_of_new_points);
250       new_points = DFO_model_.findLastSetOfInterpolationPoints();
251       multiple_new_points = true;
252       goto evaluate;
253     }
254   } else if (next_step_ == INITIALIZE_MODEL) {
255     if (number_of_new_points != 0) {
256       // Add the points
257       for (int i = 0; i < number_of_new_points; ++i) {
258         DFO_model_.SetFunctionValueAndDerivatives(
259             number_of_points_first_set + i + 1,
260             function_evaluations(i),
261             new_gradients.col(i));
262       }
263     }
264
265     DFO_model_.initializeModel();
266     set_next_step(CRITICALITY_STEP_START);
267     goto print;
268   } else if (next_step_ == CRITICALITY_STEP_START) {
269     Eigen::VectorXd gradient =
      ↪  DFO_model_.GetLagrangianGradient(DFO_model_.GetBestPoint());
270     if ((gradient.norm() > epsilon_c && force_criticality_step == false
271         && number_of_crit_step_finished_without_checking_poisedness <= 4)
          ↪  ) {
272       DFO_model_.SetTrustRegionRadius(trust_region_radius_icb);
273       set_next_step(FIND_TRIAL_POINT);
274       number_of_crit_step_finished_without_checking_poisedness++;
275       goto top;
276     } else {
277       number_of_crit_step_finished_without_checking_poisedness = 0;
278       bool is_poised =
279           DFO_model_.ModelImprovementAlgorithm(r *
            ↪  trust_region_radius_icb, new_points, new_points_indicies);
280       if ((!is_poised) || (trust_region_radius_icb > u *
        ↪  gradient.norm())) {
281         criticality_step_iteration = 0;
282         if (is_poised) {
283           force_criticality_step = false;
284           set_next_step(CRITICALITY_STEP_CHECK_CONVERGENCE);
285           goto top;
286         } else {
287           force_criticality_step = false;
288           set_next_step(CRITICALITY_STEP_ADD_POINTS);
289           multiple_new_points = true;
290           goto evaluate;
291         }
292       } else {
293         if (force_criticality_step) {
294           trust_region_radius_icb = trust_region_radius_icb * gamma;
295           force_criticality_step = false;
296           set_next_step(CRITICALITY_STEP_START);
297         } else {
298           DFO_model_.SetTrustRegionRadius(trust_region_radius_icb);
299           set_next_step(FIND_TRIAL_POINT);
300         }
```

```
        goto top;
      }
    }
  } else if (next_step_ == CRITICALITY_STEP_ADD_POINTS) {
    Eigen::VectorXi dummyI;
    Eigen::MatrixXd dummyP;
    bool is_poised = DFO_model_.ModelImprovementAlgorithm(r *
     ↪  trust_region_radius_tilde, dummyP, dummyI);
    DFO_model_.update(new_points,
                      function_evaluations,
                      new_gradients,
                      new_points_indicies,
                      number_of_new_points,
                      DFO_Model::IMPROVE_POISEDNESS);
    is_poised = DFO_model_.ModelImprovementAlgorithm(r *
     ↪  trust_region_radius_tilde, new_points, new_points_indicies);
    set_next_step(CRITICALITY_STEP_CHECK_CONVERGENCE);
    crit_steps++;
    goto print;
  } else if (next_step_ == CRITICALITY_STEP_CHECK_CONVERGENCE) {
    Eigen::VectorXd gradient =
     ↪  DFO_model_.GetLagrangianGradient(DFO_model_.GetBestPoint());

    bool is_poised =
        DFO_model_.ModelImprovementAlgorithm(r *
         ↪  trust_region_radius_tilde, new_points, new_points_indicies);

    if (trust_region_radius_tilde <= u * gradient.norm() && is_poised) {
      // The radius have been reduced, and the gradient is now sufficient
       ↪  large. Proceed to find trial point.
      is_model_CFL = true;
      double temp = max(trust_region_radius_tilde, beta *
       ↪  gradient.norm());
      double new_trust_region_radius = min(temp,
       ↪  trust_region_radius_icb);
      DFO_model_.SetTrustRegionRadius(new_trust_region_radius);
      set_next_step(FIND_TRIAL_POINT);
      goto print;
    } else {
      do {
        //decrease radius. check poisedness.
        criticality_step_iteration++;
        trust_region_radius_tilde = pow(w, (criticality_step_iteration -
         ↪  1)) * trust_region_radius_icb;
        is_poised =
            DFO_model_.ModelImprovementAlgorithm(r *
             ↪  trust_region_radius_tilde, new_points,
             ↪  new_points_indicies);
      } while (is_poised);

      if (trust_region_radius_tilde <= trust_region_radius_end){
        std::cout << "crit_steps: " << crit_steps << "\nmodel_impr: " <<
         ↪  model_impr_steps << "\nacceptance: "
                  << accept_steps << "\n";
        DFO_model_.Converged(iterations_, 0, number_of_function_calls,
         ↪  number_of_parallell_function_calls);
        for (int i = 0; i<7;i++){
```

```
346           eigen_col(new_points, DFO_model_.GetBestPoint(),i);
347           new_points_indicies(i) = i+1;
348         }
349         multiple_new_points = true;
350         terminated = true;
351         goto finished;
352
353       }
354
355       set_next_step(CRITICALITY_STEP_ADD_POINTS);
356       number_of_new_points = new_points_indicies.rows();
357       multiple_new_points = true;
358       goto evaluate;
359     }
360   } else if (next_step_ == FIND_TRIAL_POINT) {
361
362     Eigen::VectorXd gradient =
      ↪  DFO_model_.GetLagrangianGradient(DFO_model_.GetBestPoint());
363
364
365
366     if (DFO_model_.GetTrustRegionRadius() <= trust_region_radius_end) {
367       std::cout << "crit_steps: " << crit_steps << "\nmodel_impr: " <<
        ↪  model_impr_steps << "\nacceptance: "
368                 << accept_steps << "\n";
369       DFO_model_.Converged(iterations_, 0, number_of_function_calls,
        ↪  number_of_parallell_function_calls);
370       for (int i = 0; i<7;i++){
371         eigen_col(new_points, DFO_model_.GetBestPoint(),i);
372         new_points_indicies(i) = i+1;
373       }
374       terminated = true;
375       multiple_new_points = true;
376       goto finished;
377     }
378     new_point = DFO_model_.FindLocalOptimum();
379     if ((new_point - last_trial_point).norm() <=
      ↪  0.0001*DFO_model_.GetTrustRegionRadius()) {
380       force_criticality_step = true;
381       trust_region_radius_icb = gamma *
        ↪  DFO_model_.GetTrustRegionRadius();
382       set_next_step(CRITICALITY_STEP_START);
383       goto top;
384     } else {
385       last_trial_point = new_point;
386       double maxDistance =
        ↪  DFO_model_.findLargestDistanceBetweenPointsAndOptimum();
387       if (DFO_model_.norm((new_point - DFO_model_.GetBestPoint())) < tau
        ↪  * maxDistance) {
388         // Too close.
389         if (!is_model_CFL) {
390           is_model_CFL = DFO_model_.isPoised(new_point, new_point_index,
            ↪  DFO_model_.GetTrustRegionRadius());
391         }
392         if (is_model_CFL) {
393           set_next_step(TRUST_REGION_RADIUS_UPDATE_STEP);
394           goto top;
```

```
395         } else {
396           set_next_step(MODEL_IMPROVEMENT_STEP_START);
397           goto top;
398         }
399       } else {
400         set_next_step(ACCEPTANCE_OF_TRIAL_POINT);
401         number_of_new_points = 0;
402         multiple_new_points = false;
403         goto evaluate;
404       }
405     }
406   } else if (next_step_ == ACCEPTANCE_OF_TRIAL_POINT) {
407     accept_steps++;
408     DFO_model_.SetTrustRegionRadiusForSubproblem(
409         DFO_model_.GetTrustRegionRadius());
410     int t = DFO_model_.findPointToReplaceWithNewOptimum(new_point);
411     rho = (DFO_model_.GetFunctionValue(DFO_model_.getBestPointIndex()) -
       ↪  function_evaluation)
412         / (DFO_model_.evaluateQuadraticModel(DFO_model_.GetBestPoint())
413             - DFO_model_.evaluateQuadraticModel(new_point));
414
415     if (!is_model_CFL) {
416       Eigen::VectorXd dummyVec(number_of_variables_);
417       dummyVec.setZero();
418       int dummyInt = 0;
419       is_model_CFL = DFO_model_.isPoised(dummyVec, dummyInt,
         ↪  DFO_model_.GetTrustRegionRadius());
420     }
421
422     if ((rho >= eta1) || (is_model_CFL && rho > 0)) {
423       DFO_model_.update(new_point, function_evaluation, new_gradient, t,
         ↪  DFO_Model::INCLUDE_NEW_OPTIMUM);
424       set_next_step(TRUST_REGION_RADIUS_UPDATE_STEP);
425       goto print;
426     } else {
427       int index = DFO_model_.isPointAcceptable(new_point);
428       if (index != -1) {
429         DFO_model_.update(new_point, function_evaluation, new_gradient,
           ↪  index, DFO_Model::INCLUDE_NEW_POINT);
430       }
431       set_next_step(MODEL_IMPROVEMENT_STEP_START);
432       goto top;
433     }
434   } else if (next_step_ == MODEL_IMPROVEMENT_STEP_START) {
435     DFO_model_.modelImprovementStep(new_point, new_point_index);
436     if (new_point_index == -1) {
437       rho = 0;
438       force_criticality_step = true;
439       is_model_CFL = true;
440       set_next_step(TRUST_REGION_RADIUS_UPDATE_STEP);
441       goto top;
442     } else {
443       set_next_step(MODEL_IMPROVEMENT_STEP_END);
444     }
445   } else if (next_step_ == MODEL_IMPROVEMENT_STEP_END) {
446     DFO_model_.update(new_point, function_evaluation, new_gradient,
       ↪  new_point_index, DFO_Model::INCLUDE_NEW_POINT);
```

```
447       model_impr_steps++;
448       set_next_step(TRUST_REGION_RADIUS_UPDATE_STEP);
449       goto print;
450     } else if (next_step_ == TRUST_REGION_RADIUS_UPDATE_STEP) {
451       if (rho >= eta1) {
452         if (function_evaluation >
           ↪  DFO_model_.GetFunctionValue(DFO_model_.getBestPointIndex())) {
453           trust_region_radius_icb = gamma *
             ↪  DFO_model_.GetTrustRegionRadius();
454         } else {
455           double tmp = std::min(gamma_inc *
             ↪  DFO_model_.GetTrustRegionRadius(), trust_region_radius_max);
456           double weight = 0;
457           trust_region_radius_icb = weight *
             ↪  DFO_model_.GetTrustRegionRadius() + (1 - weight) * tmp;
458           rho = 0;
459         }
460       } else {
461         if (!is_model_CFL) {
462           Eigen::VectorXd dummyVec(number_of_variables_);
463           dummyVec.setZero();
464           int dummyInt = 0;
465           is_model_CFL = DFO_model_.isPoised(dummyVec, dummyInt,
             ↪  DFO_model_.GetTrustRegionRadius());
466         }
467         if (is_model_CFL) {
468           trust_region_radius_icb = gamma *
             ↪  DFO_model_.GetTrustRegionRadius();
469         } else {
470           trust_region_radius_icb = DFO_model_.GetTrustRegionRadius();
471         }
472       }
473       set_next_step(CRITICALITY_STEP_START);
474       goto top;
475
476     } else {
477       cout << "This should never be ran. \n";
478       cin.get();
479     }
480
481
482     evaluate:
483     if (multiple_new_points) {
484       number_of_new_points = new_points_indicies.rows();
485       new_gradients.resize(ng, number_of_new_points);
486       mapIndicesToCase.resize(number_of_new_points);
487       function_evaluations.resize(number_of_new_points);
488       for (int i = 0; i < number_of_new_points; ++i) {
489         number_of_function_calls++;
490         Optimization::Case *new_case = new Optimization::Case(base_case_);
491         ConvertPointToCase(new_points.col(i) +
           ↪  DFO_model_.getCenterPoint(),new_case);
492         mapIndicesToCase[i] =  new_case->GetId();
493         case_handler_->AddNewCase(new_case);
494       }
495       number_of_parallell_function_calls++;
496     } else {
```

```
497       Optimization::Case *new_case = new Optimization::Case(base_case_);
498       ConvertPointToCase(new_point + DFO_model_.getCenterPoint(),new_case);
499       mapIndexToCase =  new_case->GetId();
500       //add it to the handler!
501       case_handler_->AddNewCase(new_case);
502       number_of_function_calls++;
503       number_of_parallell_function_calls++;
504     }
505     iterations_++;
506
507     finished:
508     std::cout << "end of iterate\n";
509   }
510
511   std::string DFO::get_action_name(int a) {
512     switch (a) {
513       case 0: return "FIND_POINTS1                ";
514       case 1: return "FIND_POINTS2                ";
515       case 2: return "INITIALIZE_MODEL              ";
516       case 3: return "CRITICALITY_STEP_START              ";
517       case 4: return "CRITICALITY_STEP_CHECK_CONVERGENCE      ";
518       case 5: return "FIND_TRIAL_POINT              ";
519       case 6: return "ACCEPTANCE_OF_TRIAL_POINT         ";
520       case 7: return "MODEL_IMPROVEMENT_STEP_START       ";
521       case 8: return "MODEL_IMPROVEMENT_STEP_END          ";
522       case 9: return "TRUST_REGION_RADIUS_UPDATE_STEP     ";
523       case 10: return "CRITICALITY_STEP_ADD_POINTS       ";
524       default:return "Not a valid state                       ";
525     }
526   }
527
528   void DFO::set_next_step(int a) {
529     std::cout << "From " << "\033[1;" + color_from + "m " <<
      ↪  get_action_name(next_step_) << "\033[0m";
530     next_step_ = a;
531     std::cout << "\tTo " << "\033[1;" + color_to + ";m " <<
      ↪  get_action_name(next_step_) << "\033[0m" << std::endl;
532   }
533
534   Eigen::VectorXd
      ↪  DFO::ScaleVariablesFromAlgorithmToApplication(Eigen::VectorXd point)
      ↪  {
535     Eigen::VectorXd ret(point.rows());
536     for (int i = 0; i < number_of_variables_; i++){
537       ret[i] = scaling_[i]*point[i];
538     }
539     return ret;
540   }
541   Eigen::VectorXd
      ↪  DFO::ScaleVariablesFromApplicationToAlgorithm(Eigen::VectorXd point)
      ↪  {
542     Eigen::VectorXd ret(number_of_variables_);
543     for (int i = 0; i < number_of_variables_; i++){
544       ret[i] = point[i]/scaling_[i];
545     }
546     return ret;
547   }
```

```
548
549
550 void DFO::ConvertPointToCase(Eigen::VectorXd point,Optimization::Case*
    ↪  new_case) {
551   Eigen::VectorXd scaled(number_of_variables_);
552   scaled = ScaleVariablesFromAlgorithmToApplication(point);
553
554   new_case->SetRealVarValues(scaled);
555   new_case->set_objective_function_value(
556       std::numeric_limits<double>::max());
557 }
558
559 void DFO::CreateScalingVector() {
560   scaling_ = Eigen::VectorXd::Zero(number_of_variables_);
561   for (int i = 0; i < number_of_variables_; ++i) {
562     Model::Properties::Property::PropertyInfo
563         propinfo = variables_->
564         GetContinousVariable(
565         UUIDS_in_same_order_as_in_realvar_base_case_[i])->propertyInfo();
566     if (propinfo.coord == Model::Properties::Property::Coordinate::x) {
567       scaling_[i] = 10000.0;
568     } else if (propinfo.coord ==
    ↪  Model::Properties::Property::Coordinate::y) {
569       scaling_[i] = 10000.0;
570     } else if (propinfo.coord ==
    ↪  Model::Properties::Property::Coordinate::z) {
571       scaling_[i] = 10000.0 / 30.0;
572     } else {
573       std::cout << "The variable type is not coordinate xyz... no scaling
        ↪  applied\n";
574       std::cin.get();
575     }
576   }
577 }
578
579
580 }
581 }
```

VirtualSimulator.h

```
1  //
2  // Created by joakim on 12.06.18.
3  //
4
5  #ifndef FIELDOPT_VIRTUALSIMULATOR_H
6  #define FIELDOPT_VIRTUALSIMULATOR_H
7  #include <iostream>
8  #include <iomanip>
9  #include <casadi/casadi.hpp>
10 #include <Eigen/Core>
11
12 class VirtualSimulator {
13  private:
14   std::string problem;
15   casadi::NlpBuilder nl;
```

```
  std::vector<casadi::DM> input;
  casadi::Function f;
  casadi::Function fj;
  casadi::Function g;
  casadi::Function gj;
  int m_; //Number of constraints (linear and nonlinear);
  int n_; //Number of variables;
  int mb_; //Number of bounds.

 public:
  VirtualSimulator(std::string problemFile);
  VirtualSimulator();
  Eigen::VectorXd evaluateConstraints(Eigen::VectorXd point);
  Eigen::MatrixXd evaluateConstraintGradients(Eigen::VectorXd point);
  double evaluateFunction(Eigen::VectorXd point);
  Eigen::VectorXd evaluateFunctionGradients(Eigen::VectorXd point);
  casadi::DM convertEigenMatrixToCasadi(Eigen::MatrixXd point);
  Eigen::MatrixXd convertCasadiMatrixToEigen(casadi::DM casadiMatrix);
  int GetNumberOfVariables();
  int GetNumberOfConstraints();
  Eigen::VectorXd GetInitialPoint();
  Eigen::VectorXd GetLowerBoundsForVariables();
  Eigen::VectorXd GetUpperBoundsForVariables();
  Eigen::VectorXd GetLowerBoundsForConstraints();
  Eigen::VectorXd GetUpperBoundsForConstraints();
  bool IsFeasiblePoint(Eigen::VectorXd point);
  Eigen::VectorXd Solve();
};

#endif //FIELDOPT_VIRTUALSIMULATOR_H
```

VirtualSimulator.cpp

```
//
// Created by joakim on 12.06.18.
//

#include "VirtualSimulator.h"
VirtualSimulator::VirtualSimulator(std::string problemFile) {
  problem = problemFile;
  nl.import_nl(problem);
  std::vector<casadi::MX> f1 = {nl.f};
  std::vector<casadi::MX> f2 = {casadi::MX::vertcat(nl.x)};
  f = casadi::Function("obj", f2, f1);
  fj = f.factory("jacf", {f.name_in()}, {"jac:o0:i0", "o0"});
  auto gcat = {casadi::MX::vertcat(nl.g)};
  g = casadi::Function("obj", f2, gcat);
  gj = g.factory("jacg", {f.name_in()}, {"jac:o0:i0", "o0"});
  input = {casadi::DM(nl.x_init)}; // set the correct size of input.
  m_ = GetNumberOfConstraints();
}
VirtualSimulator::VirtualSimulator() {
}

Eigen::VectorXd VirtualSimulator::evaluateConstraints(Eigen::VectorXd
 ↪  point) {
```

```
23    input[0] = convertEigenMatrixToCasadi(point);
24    auto out = g(input);
25    Eigen::VectorXd output;
26    if (m_ > 0) {
27      output = convertCasadiMatrixToEigen(out[0]);
28
29    } else {
30      output = Eigen::VectorXd::Zero(0);
31    }
32    return output;
33  }
34
35  Eigen::MatrixXd
    ↪  VirtualSimulator::evaluateConstraintGradients(Eigen::VectorXd point)
    ↪  {
36    input[0] = convertEigenMatrixToCasadi(point);
37    auto out_gj = gj(input);
38    Eigen::MatrixXd output = convertCasadiMatrixToEigen(out_gj[0]);
39    return output;
40  }
41
42  double VirtualSimulator::evaluateFunction(Eigen::VectorXd point) {
43    input[0] = convertEigenMatrixToCasadi(point);
44    auto out = f(input);
45    Eigen::VectorXd output = convertCasadiMatrixToEigen(out[0]);
46    return output[0];
47  }
48
49  Eigen::VectorXd
    ↪  VirtualSimulator::evaluateFunctionGradients(Eigen::VectorXd point) {
50    input[0] = convertEigenMatrixToCasadi(point);
51    auto out_fj = fj(input);
52    Eigen::VectorXd output =
      ↪  (convertCasadiMatrixToEigen(out_fj[0])).transpose();
53
54    return output;
55  }
56
57  casadi::DM VirtualSimulator::convertEigenMatrixToCasadi(Eigen::MatrixXd
    ↪  point) {
58    casadi::DM casadiVector;
59    size_t rows = point.rows();
60    size_t cols = point.cols();
61
62    casadiVector.resize(rows, cols);
63    casadiVector = casadi::DM::zeros(rows, cols);
64    std::memcpy(casadiVector.ptr(), point.data(), sizeof(double) * rows *
      ↪  cols);
65
66    return casadiVector;
67  }
68
69  Eigen::MatrixXd VirtualSimulator::convertCasadiMatrixToEigen(casadi::DM
    ↪  casadiMatrix) {
70    auto casadiMatrixDense = casadi::DM::densify(casadiMatrix);
71    Eigen::MatrixXd eigenMatrix;
72    size_t rows = casadiMatrixDense.size1();
```

```
  size_t cols = casadiMatrixDense.size2();
  eigenMatrix.resize(rows, cols);
  eigenMatrix.setZero(rows, cols);
  std::memcpy(eigenMatrix.data(), casadiMatrixDense.ptr(), sizeof(double)
   ↪  * rows * cols);

  return eigenMatrix;
}
int VirtualSimulator::GetNumberOfVariables() {
  return (nl.x_init).size();;
}
int VirtualSimulator::GetNumberOfConstraints() {
  return (nl.g_ub).size();;
}
Eigen::VectorXd VirtualSimulator::GetLowerBoundsForVariables() {
  Eigen::VectorXd lb(GetNumberOfVariables());
  for (int i = 0; i < lb.rows(); ++i) {
    lb(i) = nl.x_lb[i];
  }
  return lb;
}

Eigen::VectorXd VirtualSimulator::GetUpperBoundsForVariables() {
  Eigen::VectorXd ub(GetNumberOfVariables());
  for (int i = 0; i < ub.rows(); ++i) {
    ub(i) = nl.x_ub[i];
  }
  return ub;
}
Eigen::VectorXd VirtualSimulator::GetLowerBoundsForConstraints() {
  Eigen::VectorXd lb(GetNumberOfConstraints());
  for (int i = 0; i < lb.rows(); ++i) {
    lb(i) = nl.g_lb[i];
  }

  return lb;
}

Eigen::VectorXd VirtualSimulator::GetUpperBoundsForConstraints() {
  Eigen::VectorXd ub(GetNumberOfConstraints());
  for (int i = 0; i < ub.rows(); ++i) {
    ub(i) = nl.g_ub[i];
  }
  return ub;
}
bool VirtualSimulator::IsFeasiblePoint(Eigen::VectorXd point) {
  if (GetNumberOfConstraints() <= 0) {
    return true;
  } else {
    Eigen::VectorXd constraints = evaluateConstraints(point);
    for (int i = 0; i < GetNumberOfConstraints(); ++i) {
      if (constraints[i] < nl.g_lb[i] * 0.9 || constraints[i] > 1.1 *
       ↪  nl.g_ub[i]) {
        return false;
      }
    }
  }
```

```
128   return true;
129 }
130 Eigen::VectorXd VirtualSimulator::GetInitialPoint() {
131   Eigen::VectorXd x_init(GetNumberOfVariables());
132   for (int i = 0; i < x_init.rows(); ++i) {
133     x_init(i) = nl.x_init[i];
134   }
135   return x_init;
136 }
137 Eigen::VectorXd VirtualSimulator::Solve() {
138   casadi::Dict opts;
139   opts["expand"] = true;
140   // Allocate NLP solver and buffers
141   casadi::Function solver = casadi::nlpsol("nlpsol", "ipopt", nl, opts);
142   std::map<std::string, casadi::DM> arg, res;
143   // Solve NLP
144   arg["lbx"] = nl.x_lb;
145   arg["ubx"] = nl.x_ub;
146   arg["lbg"] = nl.g_lb;
147   arg["ubg"] = nl.g_ub;
148   arg["x0"] = nl.x_init;
149   res = solver(arg);
150
151   for (auto &&s : res) {
152     std::cout << std::setw(10) << s.first << ": " <<
      ↪  std::vector<double>(s.second) << std::endl;
153   }
154   double fval = (convertCasadiMatrixToEigen(res["f"])(0));
155   Eigen::VectorXd ans = convertCasadiMatrixToEigen(res["x"]);
156
157   std::cout << "========================= THE ANSWERS
      ↪  ======================\n";
158   std::cout << "fval = " << fval << "\n";
159   std::cout << "x = \n" << ans << "\n";
160   return ans;
161
162 }
```