Norwegian University of
Science and Technology

Implementation of Linear Constraint Handling Techniques for Generating Set Search Method for Well Placement Optimization Problem on FieldOpt Platform

Bo Niu

Petroleum Engineering
Submission date: June 2017
Supervisor: Jon Kleppe, IGP
Co-supervisor: Mathias Bellout, IGP

Norwegian University of Science and Technology
Department of Geoscience and Petroleum

Abstract

Well placement optimization is important in Petroleum Field Development. Traditional optimization algorithms for well placement are stochastic algorithms. In this paper, we present a novel idea to solve such problem, which is using generating set search with linear constraint. The innovation point of our work is that we are able to provide a well-determined feasible domain of for well placement optimization by using piecewise linear constraints. We apply our linear constraint handler with generating set search method. This allows us to design a customized feasible domain that to a high degree, incorporates the reservoir engineering knowledge. Moreover, such a domain reduces the search area to accelerate the optimization speed. According to the literature review, few works can be found that attempt to solve the problem in this way.

Generating set search method is a local optimization method. Unlike stochastic algorithms, it is easy to trap into a local optimum. To alleviate the problem, we also discuss how to pick the parameters properly by testing our algorithm on example models, including the contraction factor, expansion factor, and search pattern. In addition to that, we present figures to illustrate how the algorithm work, and how the linear constraints can accelerate the optimization.

In the end, we present well placement optimization in OLYMPUS reservoir model and get satisfactory results. We optimize one producer out of sixteen wells. Our solution significantly increases the cumulative oil production of that well with a small number of iterations, which indicates a good performance and applicability of our algorithm.

Preface

This thesis is written as a part of the Master's degree in Petroleum Engineering at the Department of Geoscience and Petroleum at the Norwegian University of Science and Technology, NTNU. It was written during the spring semester of 2017 under the supervision of Prof. Jon Kleppe and co-supervised by Postdoc Mathias C. Bellout. This thesis is done in collaboration with the Petroleum Cybernetics Group at NTNU.

Acknowledgments

I would like to thank my supervisor Jon Kleppe who provides me the opportunity to work in the Petroleum Cybernetics Group at NTNU, enable me to collaborate with many talent researchers in the group. I also want to express my gratitude to Mathias Bellout for the tutoring, meeting and feedback during the thesis. He has spent so much time on me, which helped me a lot. Then, I would like to thank Einar Baumann, who helped me solve some technical problems on software. Finally, I would like to thank the other two master students in the group, Chingiz Panahli and Lingya Wang. The communication between us was always helpful and inspiring.

Table of Contents

1 Introduction **1**

1.1 Optimization Algorithm 1
1.2 Constraint handler for feasible domain 2
1.3 Contribution and Motivation of our work 2
1.4 FieldOpt Platform 4
1.5 Organization of this thesis 4

2 Generating Set Search Theory and Constraint Handling Methodology **5**

2.1 Description of the algorithm 5
2.2 Initializing the algorithm 6
2.3 Generating Sets for $\mathbb{R}^n$ without constraints 7
2.4 Update Formulas 8
2.5 Successful and unsuccessful iterations 9
2.6 Scaling factor 9
2.7 Linear constraint handler 9
2.7.1 Effect of the Linear Constraint handler 10
2.7.2 The Process of Straightforward Construction of $\mathcal{G}_k$ 11
2.8 Degenerate Case handler 13
2.8.1 An illustrative example of degenerate case 13
2.9 Summary of generating sets for $\mathbb{R}^n$ 14

3 Implementation Details in FieldOpt **17**

3.1 Algorithm 17
3.2 Miscellaneous 19
3.3 Details of the input in the driver file 19

4 Case Study **21**

4.1 Introduction of the two test reservoir models 21
4.2 Expansion factor and contraction factor picking strategy 22
4.2.1 Case Description 22
4.2.2 Optimization Solution 23

4.3 Comparison between two patterns . . . 26
4.3.1 Optimization Solution . . . 26
4.4 Optimization for two-dimensional model using GSS with linear constraints 28
4.4.1 Linear constraints picking strategy . . . 28
4.4.2 An illustrated example . . . 30
4.4.3 Optimization Solution . . . 32
4.4.4 Comparison between all the cases . . . 34
4.4.5 Interpretation of the solution . . . 35
4.5 Optimization for three-dimensional model using GSS with linear constraints 36
4.5.1 Case description . . . 36
4.5.2 Optimization without linear constraints and scaling factor . . . 38
4.5.3 Linear constraints and scaling factor picking strategy . . . 40
4.5.4 An illustrative example . . . 41
4.5.5 Optimization Solution . . . 43

5 Application of the Method for OLYMPUS Reservoir Model 47
5.1 Description of OLYMPUS model for optimization challenge . . . 47
5.1.1 The optimization challenge of the model . . . 48
5.2 Optimization problem description . . . 48
5.3 Parameters for GSS algorithm . . . 49
5.3.1 Linear constraints . . . 49
5.3.2 miscellaneous . . . 50
5.4 Optimization results . . . 51
5.5 Interpretation of the results . . . 53

6 Summary and Recommendations 57
6.1 Summary . . . 57
6.2 Recommendations for further work . . . 58
6.2.1 New way to define the location of a well . . . 58
6.2.2 More advanced degenerate case handler . . . 58
6.2.3 More algorithms to compare . . . 59
6.2.4 More search directions . . . 59
6.2.5 Parallel Computing . . . 59

Bibliography 61

Appendices 65

A How FieldOpt Works 65
A.1 Driver Files . . . 65
A.2 Optimizer . . . 65
A.3 Simulator . . . 66
A.4 Model . . . 66
A.5 Objective Function . . . 66
A.6 Constraints . . . 66
A.7 Bookkeeper . . . 67

A.8 The main Loop of Serial Runner 67

B An Example of the Driver File **69**

C GSS_Linear_Constraints.h **73**

D GSS_Linear_Constraints.cpp **77**

E gss_patterns.hpp **89**

F Initial Well placement **93**

G Optimized Well placement **95**

List of Tables

3.1 The name of parameters needed in driver file 19

4.1 Configurations for the two-dimensional model 23
4.2 Results of case 1-4 23
4.3 Results of case 5-8 26
4.4 Results of case 9-12 32
4.5 Comparison between unconstrained and constrained condition 34
4.6 Configurations for the three-dimensional model 38
4.7 Result of case 13 39
4.8 Result of case 14 43

5.1 Initial configurations for OLYMPUS case 51
5.2 Optimization results 52

List of Figures

2.1 Different search patterns . . . 8
2.2 Effects of linear constraints . . . 10
2.3 Nondegenerate Case handler . . . 14

3.1 GSS_Linear_Constraints Class . . . 18

4.1 Two test reservoir models . . . 22
4.2 Search map of case 1-4 . . . 24
4.3 Search map of case 5-8 . . . 27
4.4 Linear constraints for two-dimensional model . . . 29
4.5 Linear constraints proposed for the two-dimensional model . . . 29
4.6 An illustrative example of the iterations for two-dimensional case . . . 31
4.7 Search map of case 9-12 . . . 33
4.8 Number of simulations VS objective function value . . . 35
4.9 Oil distribution of the two-dimensional model . . . 36
4.10 Comparison between the base case and the best optimized case, case 11 . 37
4.11 Initial position of the wells . . . 38
4.12 Search map of case 13 . . . 39
4.13 Linear constraints proposed for the three-dimensional model . . . 40
4.14 The effect of linear constraints and scaling factor . . . 41
4.15 An illustrative example of the iterations for three-dimensional case . . . 42
4.16 Search map of case 14 . . . 44
4.17 Comparison between the base case and optimized case . . . 44
4.18 Cumulative oil and water production of the three-dimensional model . . . 45
4.19 Oil distribution of the three-dimensional model (first layer excluded) . . . 46

5.1 OLYMPUS reservoir model . . . 48
5.2 The challenge of the channels . . . 49
5.3 Oil saturation distribution for the initial case . . . 49
5.4 Linear constraints . . . 50
5.5 Linear constraint matrix for OLYMPUS model . . . 51

5.6 Evolution of the objection function value 52
5.7 Optimal well completion 53
5.8 Oil saturation distribution after optimization 53
5.9 Cumulative oil production of the field 54
5.10 Cumulative water production of the field 54
5.11 Cumulative oil production of PROD-10 54
5.12 Cumulative water production of PROD-10 55
5.13 Oil production rate of PROD-10 55
5.14 Water production rate of PROD-10 55

6.1 The disadvantage of using linear constraints in Z-direction 58

Chapter 1

Introduction

Determination of the location of wells is important to develop a reservoir optimally. The idea behind such problem is to try different locations generated by the optimization algorithm until the best one. This thesis concerns implementing generating set search algorithm with linear constraints for solving well placement optimization problems.

1.1 Optimization Algorithm

The discrete optimal well placement problem is commonly solved by using derivative-free optimization algorithms. Due to the effects of reservoir heterogeneity, these problems can display very rough optimization surfaces, with multiple local optima. Therefore most researchers have focused on using derivative-free stochastic search procedures. The most popular methods are Genetic Algorithms (GA) (Badru and Kabir, 2003; Bangerth et al., 2006; Artus et al., 2006). Another commonly-used stochastic search algorithm is Particle Swarm Optimization method (PSO) (Onwunalu and Durlofsky, 2010a; Jesmani Mansoureh and Foss, 2015). In the work of Onwunalu and Durlofsky (2010a), PSO was found to perform better than GA. Other stochastic search methods used by the researchers are Simulated Annealing (SA) (Beckner and Song, 1995), and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) (Bouzarkouna et al., 2012).

However, these stochastic search methods are not supported by solid convergence theory, and have tuning parameters that are difficult to determine. Besides the stochastic optimization procedures discussed above, there are also researchers who solve the well placement optimization problem using local search optimization methods. Isebor (2013) performed Mesh Adaptive Direct Search (MADS) as one of the algorithm in his paper. Bellout et al. (2012) performed Pattern Search for joint optimization of well placement and controls. These methods also gave satisfactory results with an acceptable number of simulations according to those works.

1.2 Constraint handler for feasible domain

Well placement optimization problems always come together with different kinds of constraints, including constraints to limit the well length, inter-well distance, feasible domain to place the well, etc. While dealing with the feasible reservoir domain, most researchers only define upper and lower bound for it, which means they perform the search process within a cubic space. To handle these bound constraints, for example, points that violate bound constraints are projected back onto the boundary of search space (T.D. Humphries, 2015). Onwunalu (2010) apply the absorb technique for PSO method, in which invalid particles are moved to the nearest bound and the affected velocity components are set to zero. These kinds of constraint handling techniques can also be called repair techniques since infeasible solutions are converted into feasible candidates.

Jesmani et al. (2016) enable the use of user-defined feasible regions of the model by approximating the boundary of each region by piecewise polynomials. In this way, all the constraints for well length, feasible domain, well orientation and inter-well distance are solved together. They apply penalty function and decoder to solve these constraints and compare. The decoder can convert any nonlinear constraints into boundary constraints. The idea of penalty functions is to transform a constrained optimization problem into an unconstrained one by adding a certain value to the objective function based on the amount of constraint violation present in a certain solution. It is the most common approach to handle constraints. Other related work involves Bouzarkouna et al. (2010) and Onwunalu and Durlofsky (2010b) who use penalty method.

1.3 Contribution and Motivation of our work

In this work, we first implement Generating Set Search (GSS) algorithm to resolve the well placement optimization problem. Our implementation of GSS algorithm, introduced by Kolda et al. (2003), denotes the generic term of all classical pattern search methods supported by solid convergence theory, including Generalized Pattern Search (GPS), Mesh Adaptive Direct Search (MADS) and Hooke-Jeeves Direct Search (HJDS). In our work, we use linear constraints to define the feasible reservoir domain. We then apply a linear constraint handling technique taken from Lewis et al. (2007a) for GSS algorithm to handle these linear constraints.

In our work, we approximate the boundary of the feasible reservoir domain by piecewise linear constraints. On one hand, the approximation by using linear constraints is usually precise enough to replace the irregularly shaped reservoir regions. And it is easier to implement than the approximation by piecewise polynomials in the work of Jesmani et al. (2016) in terms of mathematics. On the other hand, by introducing these linear constraints, we can narrow down the search space by using reservoir engineering knowledge. For example, instead of searching the whole reservoir within the reservoir boundary defined by bound constraints, we can divide out certain zones with high oil saturation by linear constraints, in which the best location of the producer is much more likely to be than the other places. In consequence, the size of the search space can be reduced considerably by eliminating the simulation of infeasible development scenarios. This means we can shorten the optimization time by incorporating reservoir engineering knowledge,

which is rarely discussed by other researchers.

From the literature review, Umut Ozdogan (2005) are the first who attempt on geometrical constraints of the well placement problem considering the irregular shape of the reservoir. They proposed Fixed Pattern Approach (FPA) which uses a hybrid GA to locate wells on user-defined patterns. Their methodology needs two primary parameters to define a line pattern, which are the distance of the wells on the line to the reservoir boundary and well spacing. With the two parameters, they can capture the shape of the feasible domain. Comparison with respect to the conventional window approach showed that FPA significantly reduced the CPU time and resulted in practical and economical well locations. However, their methodology did not fix the number of wells. In other words, they are searching the best well pattern instead of well placement for each individual, which is different from our work. And also, the methodology they design for picking feasible domain is different from us.

Theoretically, the advantage of solving the linear constraints using our method can be summarized as follows:

1. Compared with the penalty method, it can perform better since the performance of penalty method is highly problem-dependent. And penalty method requires additional tuning of several parameters, which makes the problem more complicated.

2. Compared with the decoder method, it is much easy to implement since the decoder method requires more advanced mathematical knowledge and is harder to implement when the reservoir model is complicated and irregular.

3. Compared with the repair technique that projects infeasible solutions onto its corresponding feasible space, the linear constraint handling technique implemented in this work can make use of the shape of the constraints and change the searching direction more efficiently to accelerate the convergence speed. On the contrary, the projection process will generate too many points lying on the boundary.

4. Our linear constraint handler can make use of the reservoir engineering knowledge to shorten the optimization time as discussed above.

The advantages of using GSS algorithm compared to stochastic search methods for well placement optimization problem are:

1. It has a solid mathematical foundation of convergence which can be found at Kolda et al. (2007).

2. It is compatible with any other constraint handling technique, which means we can still apply other constraint handling technique afterward to include other types of constraints.

We know that using GSS method may lead to getting trapped into local optima. Therefore, we will discuss how to alleviate such problem in chapter 4. Furthermore, in many well placement problems, finding a reasonable local optimum following some amount of global exploration is often sufficient (Bellout et al., 2012).

The disadvantage of the linear constraint handling technique implemented in this work is that it can only be applied to the GSS type of algorithms. That is, it may not be consistent with some popularity-based algorithms.

1.4 FieldOpt Platform

Our optimization platform for this thesis is FieldOpt, a software framework developed by Baumann (2015). FieldOpt is written in C++. It is designed to promote researches by efficiently integrating methodology from optimization theory and petroleum engineering. The main idea behind FieldOpt is to consolidate and make user-friendly much of the groundwork necessary to conduct optimization on a variety of petroleum problems.

FieldOpt has several different functionality sections, most of which can be treat as "black box" that are not necessary to touch. However, in order to integrate our algorithm into FieldOpt platform, one needs to have a general understanding of how FieldOpt works. In general, FieldOpt parses the input file which contains reservoir model and configurations for optimizer and wells, and then outputs the optimization results. Currently, FieldOpt is still being updated, and the functionality is still being improved. In appendix A, we present the current state of FieldOpt, and how FieldOpt works as a whole. We also introduce the important functionality modules we use in our work, including the reservoir simulator we choose, the objective function, the runner, and other types of constraints we use in this thesis. One can refer to appendix A for more detail.

Currently, FieldOpt can apply compass search method for well placement optimization problem. Compass search method can be seen as the simplest form of GSS method. In this thesis, we will extend it to the general GSS method and add linear constraint handler to it. And also, the compass search algorithm is always the one we use to compare with our algorithm in chapter 4.

1.5 Organization of this thesis

In chapter 2, we will introduce the theory of GSS algorithm with linear constraints in great detail. In chapter 3, we present the implementation details of our algorithm. That is, how we integrate our algorithm into FieldOpt platform. In chapter 4, we perform different test cases for the algorithm we implemented in order to show the performance of our method. Then in chapter 5, we will make use of the knowledge we gained from chapter 4 and solve the well placement optimization problem in OLYMPUS reservoir model. Finally, in chapter 6, we summarize our work and give recommendations for further work.

Chapter 2

Generating Set Search Theory and Constraint Handling Methodology

The linearly-constrained optimization problem considered is

$$\begin{array}{ll} minimize & f(x) \\ subject\ to & l \leq Ax \leq u \end{array} \tag{2.1}$$

$f : \mathbb{R}^n \to \mathbb{R}$ is the objective function. The linear constraints are in the form of a matrix of $A \in \mathbb{R}^{m \times n}$. Generating set search (GSS) is a class of direct search optimization methods that includes compass search and similar pattern search methods. The algorithm of GSS method for this thesis is provided in algorithm 1. This chapter will discuss the theoretical properties of the method based on the algorithm 1. The whole process of the optimization method, from the initialization to the termination, will be covered in this chapter. We aim at providing the solid theoretical foundation of our algorithm.

2.1 Description of the algorithm

As illustrated in algorithm 1, the basic idea behind GSS algorithm is to perform the search by calculated search pattern. Given an initial point, the algorithm will generate several new trial points around it according to the search pattern. If there exists a better point, then the iteration is called a successful iteration. The step length will expand, and the tentative best point will be updated to replace the initial one. Otherwise, the iteration is called unsuccessful. The step length will contract, and the tentative best point is still the initial one. In the following iterations, the search process will always be performed around the tentative best point using the same, or a new search pattern. Such search process will continue until the step length reaches the lower limit we set.

The main difference between our algorithm and the traditional unconstrained pattern search algorithm is that our search pattern in each iteration can change based on the linear

- **Initialization.**
  - Let $f : \mathbb{R}^n \to \mathbb{R}$ be given.
  - Let $x_0 \in \mathbb{R}^n$ be the initial guess.
  - Let $\Delta_{tol} > 0$ be the tolerance used to test for convergence.
  - Let $\Delta_0 > \Delta_{tol}$ be the initial value of the step-length control parameter.
- **Algorithm.** For each iteration $k = 1, 2, ...$

  Step 1. Let $\mathcal{D}_k = \mathcal{G}_k \cup \mathcal{H}_k$. Here $\mathcal{G}_k$ is the generating set for $\mathbb{R}^n$ defined by the search pattern or calculated by the linear constraint handler, and $\mathcal{H}_k$ is a user-defined finite (possibly empty) set of additional search directions.

  Step 2. If there exists $d_k \in \mathcal{D}_k$ such that $f(x_k + \Delta_k d_k) < f(x_k)$, then do the following:
  - Set $x_{k+1} = x_k + \Delta_k d_k$ (change the iterate).
  - Set $\Delta_{k+1} = \phi_k \Delta_k$ where $\phi_k \geq 1$ (optionally expand the step-length control parameter).

  Step 3. Otherwise, $f(x_k + \Delta_k d) \geq f(x_k)$ for all $d \in \mathcal{D}_k$, so do the following:
  - Set $x_{k+1} = x_k$ (no change to the iterate).
  - Set $\Delta_{k+1} = \theta_k \Delta_k$ where $0 < \theta_k < \theta_{max} < 1$ (contract the step-length control parameter).
  - If $\Delta_{k+1} < \Delta_{tol}$, then **terminate**.

Algorithm 1: Generating Set Search Algorithm

constraints nearby. Therefore, the key point of our algorithm is how we construct the search pattern in each iteration, which is indicated in step 1 of algorithm 1. In the following section, we will go through every part of algorithm 1 in great detail, especially for the process of construction of the search pattern.

2.2 Initializing the algorithm

A few comments regarding the initialization of the algorithm are in order. First, since we are dealing with the constrained optimization problem, the initial point x_0 must be in the feasible region. If not, according to our implementation, the search will either return an error or never enter the feasible region.

The parameter Δ_{tol} is problem-dependent and plays a major role in determining both the accuracy of the final solution and the number of iterations. Smaller choices of Δ_{tol} yield higher accuracy but the price is an increase in the number of iterations. For well placement problem, usually we set it to be the value of half size of a single grid block so that the point can move at least one grid block.

For the initial value of the step-length control parameter Δ_0, relatively speaking, it is better to set it to be a bigger one in case of getting trapped into local optimum too early.

2.3 Generating Sets for $\mathbb{R}^n$ without constraints

$\mathcal{D}_k$ is defined by a set of search directions which contains a $generating\ set$ $\mathcal{G}_k$ for $\mathbb{R}^n$. See Definition 1 for generating sets , which is also defined as $positive\ spanning\ set$. Pay attention that all the coefficients are nonnegative. $\mathcal{H}_k$ is any other additional direction that can be included, and is allowed to set to be empty. For our implementation, $\mathcal{H}_k$ is always empty with $\mathcal{G}_k$ changing iteration by iteration.

Definition 1. *Let $\mathcal{G} = d^{(1)}, ..., d^{(p)}$ be a set of $p \geq n+1$ vectors in $\mathbb{R}^n$. Then the set G generates (or positively spans) $\mathbb{R}^n$ if for any vector $v \in \mathbb{R}^n$, there exist $\lambda^{(1)}, ..., \lambda^{(p)} \geq 0$ such that*

$$v = \sum_{i=1}^{p} \lambda^{(i)} d^{(i)}.$$

A generating set must contain at least $n+1$ vectors. This means any vector in the search space can be constructed by a positive linear combination of the components of $\mathcal{G}$. If no constraints are nearby in one iteration, the problem can be treated as an unconstrained problem, and the generating set can always be the same. The generating set is also called search pattern. In this thesis, we implement two search patterns for the two-dimensional and three-dimensional problem. The patterns are illustrated in figure 2.1. The left ones are called *Compass* pattern and the right ones are called *Fast* pattern in our thesis. Compass pattern denotes searching along the coordinate axis. While Fast pattern has fewer trial points in each iteration compared with Compass pattern. In this way, we incorporate the built-in compass search method by adding the Compass pattern to our GSS algorithm.

The corresponding generating sets in vector form of vector for the Fast and Compass pattern are:

$$\mathcal{G}_{Fast} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \right\}$$

$$\mathcal{G}_{Compass} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\}$$

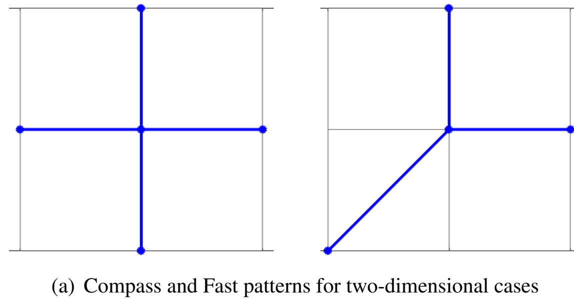
for two-dimensional case in figure 2.1(a), and

$$\mathcal{G}_{Fast} = \left\{ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \right\}$$

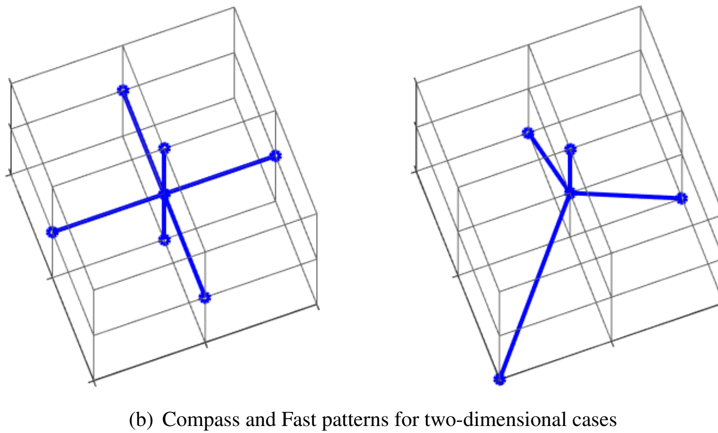
$$\mathcal{G}_{Compass} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \right\}$$

for three-dimensional case in figure 2.1(b).

When there are linear constraints nearby, the linear constraint handler is used to construct the generating sets. In this way, the generating sets will no longer be limited to the two type we introduced above. There will be diverse generating sets calculated by our linear constraint handler. This is the key point that makes our algorithm outperform other pattern search method, not only in the aspect of optimization results, but also in the aspect of the number of simulations, which is performed in chapter 4. The process of constructing generating sets when the point is near linear constraints will be discussed later in the following sections.



(a) Compass and Fast patterns for two-dimensional cases



(b) Compass and Fast patterns for two-dimensional cases

Figure 2.1: Different search patterns

2.4 Update Formulas

When the algorithm finds a better trial point along certain direction, it means that it is still on the way of the final optimum. Therefore, the step length will be expanded to search for a larger range of space. In contrast, if the algorithm fails to find a better trial point in an iteration, it tends to think that it has already found the final optimum. Thus the step length will contract. In sum, the rules for updating the step-length control parameter Δ_k are:

$$\Delta_{k+1} = \begin{cases} \phi_k \Delta_k, & k \in \mathcal{S} \\ \theta_k \Delta_k, & k \in \mathcal{U} \end{cases}$$

with $\phi_k \geq 1$ and $0 < \theta_k \leq \theta_{max} < 1$. How to better choose the value of ϕ_k and θ_k is discussed in chapter 4.

2.5 Successful and unsuccessful iterations

In one iteration, if there exists a $d_k \in \mathcal{D}_k$ for which

$$f(x_k + \Delta_k d_k) < f(x_k), \tag{2.2}$$

Then the iteration is called *successful*. In this case, we update to the best point. If the condition in equation 2.2 is not satisfied, then we call the iteration *unsuccessful*. In this case, the best point is unchanged.

2.6 Scaling factor

GSS method is extremely sensitive to scaling, so it is important to use an appropriate scaling to get the best performance. In this paper, we allow as an option the common technique of shifting and rescaling the variables so that they have a similar range and size (Lewis et al., 2007b). We work in a computational space whose variables ω are related to the original variables x via $x = D\omega + c$, where D is a diagonal matrix with positive diagonal entries and c is a constant vector, both provided by the user. In this way, the original problem in equation 2.1 is transformed to:

$$\begin{array}{ll} minimize & f(Dx + c) \\ subject\ to & l - Ac \leq AD\omega \leq u - Ac \end{array} \tag{2.3}$$

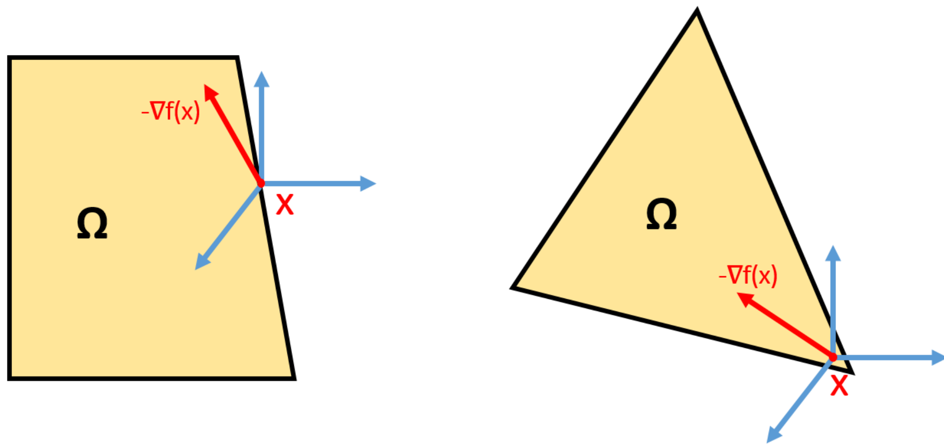
The graphical illustration and explanation of the scaling factor will be performed in chapter 4.

2.7 Linear constraint handler

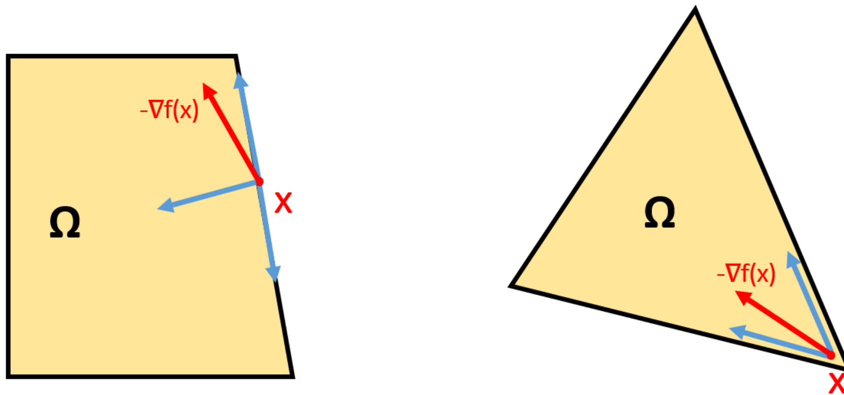
If there exist linear constraints, when the current best point is not near any boundary, the algorithm will search along the same directions as the unconstrained GSS method. When the current best point is near any boundary, the constraint handling technique will change the set of search directions, subject to the geometry of the constraints nearby. How the new generating set are constructed is discussed in this section.

2.7.1 Effect of the Linear Constraint handler

When the point is near linear constraints, if the algorithm has no linear constraint handling technique, the optimizer will proceed to search as in the unconstrained situation. Then it may encounter two issues shown in figure 2.2. Here we are taking the example of using Fast pattern in figure 2.1(a). The area in orange color denotes the feasible domain. The lines in black are the linear constraints. And the red arrow refers to the direction where the objective function value decreases. Therefore we expect the search can always be performed along this direction, or at least along an direction close to this direction.



(a) No feasible step and inefficient step



(b) Search direction after linear constraint handler

Figure 2.2: Effects of linear constraints

However, the constraints may prevent the search from taking a feasible step that can

reduce the objective function. Such situation is illustrated in the left one in figure 2.2(a). Note that only the search direction whose angle with the red line is less than 90 degree is regarded as a feasible search direction that can reduce the objective function value. However, the only one such search direction in the figure is totally outside the feasible region since the point located on the linear constraint.

Furthermore, even though there may exist a pattern search direction in the feasible space that yields decrease, the feasible step length may only be so short. This is illustrated in the right one in figure 2.2(a). We note that two of the three search directions in the figure are feasible to reduce the objective function value. However, the feasible step length can only be so short because the point is too close to the linear constraints. Therefore, only a small improvement in the objective function can be realized in either case.

In consequence, what is needed is directions that allow the search to move along any nearby boundaries, taking feasible steps that are sufficiently long. This is how our linear constraint handler works. Figure 2.2(a) illustrates the search directions after our linear constraint handler. We can see that the search directions changed according to the nearby constraints. The case on the left has three search directions, two of whom are along the linear constraint and one of whom is perpendicular to it. The case on the right has two search directions which are parallel to the nearby constraints. It is evident that the search for both two cases is much better now. In the following section, we will introduce our linear constraint handler in detail.

2.7.2 The Process of Straightforward Construction of $\mathcal{G}_k$

Let a_i^T denote the rows of A in equation 2.1. Then partition all the constraints into sets of equalities and inequalities as

$$\mathcal{E} = \{\ i\ |l_i = u_i\} \text{ and } \mathcal{I} = \{\ i\ |l_i < u_i\}. \tag{2.4}$$

We denote by $C_{l,i}$ and $C_{u,i}$ the faces of the feasible polyhedron:

$$C_{l,i} = \{y|a_i^T y = l_i\} \text{ and } C_{u,i} = \{y|a_i^T y = u_i\}. \tag{2.5}$$

The distance to nearby inequality constraints along directions that also satisfies equality constraints are:

$$dist_{\mathcal{N}}(x, S) = \begin{cases} |a^T x - b|/\|Z^T a\| & \text{if } Z^T a \neq 0, \\ \quad 0 & \text{if } Z^T a = 0 \text{ and } a^T x = b, \\ \quad \infty & \text{if } Z^T a = 0 \text{ and } a^T x \neq b. \end{cases} \tag{2.6}$$

Where $\mathcal{N}$ denotes the nullspace of the equality constraints (the matrix whose rows are a_i^T for $i \in \mathcal{E}$). Let Z be an $n \times r$ orthogonal matrix whose columns are a basis for $\mathcal{N}$. And $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$, $S = \{\ y\ |a^T y = b\}$.

For those who are not familiar with nullspace, see Definition 2.

Definition 2. *Let A be an m by n matrix, and consider the homogeneous system*

$$Ax = 0$$

the set of all vectors x which satisfy this equation forms a subset of $\mathbb{R}^n$. This subset actually forms a subspace of $\mathbb{R}^n$, called the nullspace of the matrix A and denoted $N(A)$.

Then define the sets of inequalities at their bounds that are within distance ε of x:

$$\mathcal{I}_l(x,\varepsilon) = \{i \in \mathcal{I} | dist_{\mathcal{N}}(x, C_{l,i}) \leq \varepsilon\},$$

$$I_u(x,\varepsilon) = \{i \in \mathcal{I} | dist_{\mathcal{N}}(x, C_{u,i}) \leq \varepsilon\}.$$

Let $\mathcal{I}_E(x,\varepsilon) = \mathcal{I}_l(x,\varepsilon) \cap \mathcal{I}_u(x,\varepsilon)$. Given a feasible x and an $\varepsilon > 0$, let

$$\mathcal{W}_E(x,\varepsilon) = \{a_i | i \in \mathcal{E}\} \cup \{a_i | i \in \mathcal{I}_E(x,\varepsilon)\},$$

which corresponds to the equality constraints together with inequality constraints for which the faces for both lower and upper bounds are within the distance ε of x.

$$\mathcal{W}_I(x,\varepsilon) = \{-a_i | i \in \mathcal{I}_l(x,\varepsilon) \backslash \mathcal{I}_E(x,\varepsilon)\} \cup \{a_i | i \in \mathcal{I}_u(x,\varepsilon) \backslash \mathcal{I}_E(x,\varepsilon)\},$$

which corresponds to the set of **outward-pointing** normals to the inequalities for which the faces of exactly one of their lower or upper bounds is within distance ε of x. In this thesis, these linear constraints are called *active* constraints in the corresponding iteration. Then, algorithm 2 is proceeding to attempt the straightforward construction of $\mathcal{G}_k$.

Step 1. Let L be the linear subspace spanned by the vectors in $\mathcal{W}_E(x_k, \varepsilon_k)$. Compute a basis Z for $L^{\perp}$.

Step 2. Check whether the set of vectors $\{Z^T p | p \in \mathcal{W}_I(x_k, \varepsilon_k)\}$ is linearly independent. If so, proceed to Step 3. If not, proceed to the degenerate case handler in explained in algorithm 3 the next section.

Step 3. Let Q be the matrix whose columns are the vectors $Z^T p$ for $p \in \mathcal{W}_I(x_k, \varepsilon_k)$. Compute a right inverse R for Q^T and a matrix N whose columns are a positive spanning set for the nullspace of Q^T .

Step 4. $\mathcal{G}_k$ is then the columns of ZR together with the columns of ZN.

Algorithm 2: Linear constraint handler

We note that for our work, we are dealing with well placement optimization problem. Therefore, we do not consider equality constraints since it makes no sense to force a well to be located on a certain plane. For this reason, the basis Z for $L^{\perp}$ is always the identity matrix.

2.8 Degenerate Case handler

As illustrated in step 2 of algorithm 2, if the set of vectors $\{Z^T p | p \in \mathcal{W}_I(x_k, \varepsilon_k)\}$ is not linearly independent, Then we have a degenerate case. Mathematically speaking, degenerate case in our work means the case that cannot be solved by algorithm 2. In short, step 3 and 4 will not work for some situation. That is to say, if we still use algorithm 2 to calculate the generating sets in such situation, it will give very weird results. Therefore, we need a degenerate case handler to deal with the degenerate case. In this subsection, first, we give a graphical understanding of the degenerate case. Then we introduce our degenerate case handler.

2.8.1 An illustrative example of degenerate case

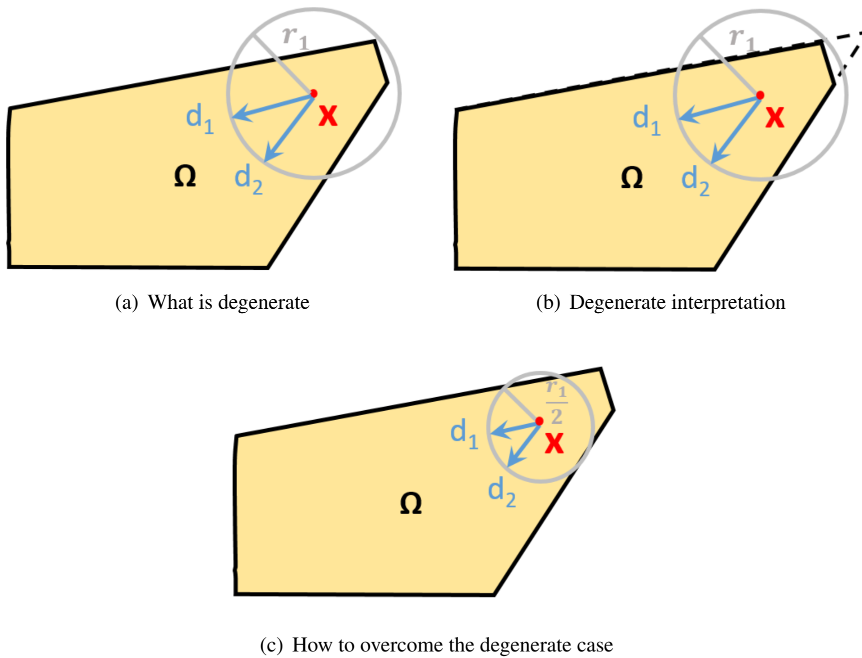
An illustrative example of degenerate case is shown in figure 2.3(a), where there are *three* active linear constraints for the *two*-dimensional problem. For this case, we expect the two search directions shown in the figure. However, such two search directions cannot be calculated by algorithm 2. If we force to use algorithm 2, we will get some illogical search directions. However, we realize that one of the boundaries does not play any role in determining the search directions as illustrated in figure 2.3(b). To be more specific, when we extrapolate the two linear constraints until they intersect, things become more clear: The two linear constraints we extrapolate are the two which actually determine the search directions. If the rest one linear constraint is removed, the search direction should remain the same. Therefore, how to remove the unnecessary nearby constraints is the key of our degenerate case handler. Another typical example for a degenerate case can be a point near the apex of the pyramid, where we have *four* boundary planes under the *three*-dimensional problem.

It is very complicated to find out which constraint has no effect and which one has from mathematic. In the paper by Lewis et al. (2007b), a C-library package named *cddlib* package was used to solve the degenerate case. In this thesis, we use a much simpler method illustrated in algorithm 3 to overcome such problem. Whenever the degenerate case occurs, we will contract the step length until a nondegenerate case is detected as shown in figure 2.3(c).

With a lot of experiments, we find that our degenerate case handler is sufficient enough to deal with the well placement optimization problem.

do
| $\Delta_{k+1} = \theta_k \Delta_k$;
while $\{$ $\{Z^T p | p \in \mathcal{W}_I(x_k, \varepsilon_k)\}$ *is not linearly independent* $\}$;

Algorithm 3: Degenerate case handler



(a) What is degenerate (b) Degenerate interpretation

(c) How to overcome the degenerate case

Figure 2.3: Nondegenerate Case handler

2.9 Summary of generating sets for $\mathbb{R}^n$

In summary, The algorithm of generating set for $\mathbb{R}^n$ in step 1 of algorithm 1 in this thesis is listed in algorithm 4:

Check distance to all linear constraints;

if *Active constraints detected* **then**
  $\mathcal{G}_k \leftarrow$ Linear constraint handler in algorithm 2;
else
  $\mathcal{G}_k \leftarrow$ Unconstrained search pattern in figure 2.1;
end

Algorithm 4: Generating Sets for $\mathbb{R}^n$

All above exhaustively explain how our GSS algorithm work. The C++ code of our algorithm is provided in appendix C-E, which contains all the functionalities we discuss in this chapter. Furthermore, in the next chapter, we discuss the implementation details of our algorithm in C++ to explain how our algorithm is implemented in codes, for sake of making things clear for other researchers who want to make changes based on it.

Also, the work of Kolda et al. (2007) and Griffin et al. (2008) contains illustrative

figures of the search process. In chapter 4, we will present our own figures to show the iterations graphically for two-dimensional and three-dimensional reservoir cases.

Chapter 3

Implementation Details in FieldOpt

In the previous chapter, we have discussed the theoretical properties of our algorithm. Still, there are a lot of work that needs to be done to integrate our algorithm into FieldOpt platform. In this chapter, we discuss the implementation details of our algorithm. Figure 3.1 illustrates the class of our algorithm. In the figure. # denotes the protected member in the class, and - denotes the private ones. In the following section, we will give necessary explanations to the implementation.

3.1 Algorithm

GSS_Linear_Constraints Class includes all the functionalities of our algorithm. As mentioned in the main loop of the serial runner in Appendix A.8, while the optimizer is not finished, it will fetch new case according to the algorithm used. It is here that the program will jump to our own class, the GSS_Linear_Constraints Class. The following section explains the most important functions in the class. We note that all the bold font in the following section denotes the function name in the class.

- The **generate_trial_points()** implements the algorithm 1 in chapter 2. It defines how to generate new points in the iteration, which is the fundamental function of our algorithm. All other functions related to the algorithm is inside **generate_trial_points()** function.
- The **satisfy_constraints()** implements the process of checking distance between the current point and all the linear constraints illustrated in equation 2.6. It will return TRUE if all the distance is larger than the step length, which means all the new points are within the feasible domain. Otherwise, it will return FALSE.
- If there will be points outside linear constraints when using unconstrained search pattern, **change_pattern_conform_to constraints()** function will take over to change the search pattern. It implements the full functionalities of our linear constraint handler. **check_linear_independence()** is inside the function, which implements our

GSS Linear Constraints
actual_dimensions_for_each_point: int
linear_constraints_: MatrixXd
bounding_: VectorXd
unsatisfied_constraints_index_: vector<int>
pattern_: vector<VectorXd>
initial_pattern_: vector<VectorXd>
welltype_: Settings::Optimizer::WellType
step_length_of_pattern_: double
spline_point_id_map_: vector<QUuid>
working_points_: vector<QUuid>
num_of_well_: int
constraint_flag_: bool
scaling_factor_D: vector<double>
scaling_factor_C: vector<double>
satisfy_constraints(): bool
constraint_flag_: bool
scaling_factor_D: vector<double>
scaling_factor_C: vector<double>
satisfy_constraints(): bool
change_pattern_conform_to_constraints(): void
check_linear_independence(): bool
handleEvaluatedCase(Case *c) override: void
generate_trial_points(): QList<Case *>
contract(): void
expand(): void
is_converged(): bool
IsFinished(): TerminationCondition
select_pattern(Settings::Optimizer *): void
transfer_value_to_linear_constraints(QString): void
transfer_value_to_bounding(QString): void
transfer_value_to_scaling_factor_D(QString): void
transfer_value_to_scaling_factor_C(QString): void
set_spline_point_id_map_(Model::Properties::VariablePropertyContainer*): void
get_working_points(Model::Properties::VariablePropertyContainer *): void
scale_variable(double, double, double): double
revert_scaled_variable(double, double, double): double
- iterate(): void
- is_successful_iteration(): bool

Figure 3.1: GSS_Linear_Constraints Class

degenerate case handler. It will read the index of unsatisfied linear constraints. If they form the degenerate case, it will contract the step length as shown in algorithm 3 in chapter 2.

3.2 Miscellaneous

There are other parameters and functions that are irrelevant to the algorithm. We will discuss them in this section.

- The linear constraints we set are supposed to be a matrix and respectively. However, They cannot be in the form of a matrix in the JSON file because JSON format does not support matrix. In consequence, we change the form of the matrix to be a string of number separated by a space. And then we use **transfer_value_to_linear_constraints** to restore it to the original state. Such is the same in **transfer_value_to_bounding**, **transfer_value_to_scaling_factor_D**, and **transfer_value_to_scaling_factor_C**.
- In fieldOpt, all the variables has their own ID for the optimizer to distinguish what kind of variables they are, such as heel.x or toe.z. **set_spline_point_id_map_** will store all the IDs in sequential order in **spline_point_id_map_**. For one well, the order is heel.x, heel.y, heel.z, toe.x, toe.y, toe.z. Only in this way can we manipulate on a certain variable or change the value of it correctly.

Table 3.1: The name of parameters needed in driver file

Parameters
MaxEvaluations
InitialStepLength
MinimumStepLength
ContractionFactor
ExpansionFactor
ActualDimensionsForEachPoint
WellType
ConstraintFlag
LinearConstraints
Bounding
SearchPatternWhenNoConstraintNearby
ScalingFactorD
ScalingFactorC

3.3 Details of the input in the driver file

In the driver file, there are 13 parameters in total that are necessary to initialize the optimizer. Table 3.1 provides the name of these parameters. Most parameters are as indicated

by their name. They are assigned to the corresponding parameters in GSS_Linear_Constraints Class. Besides that, **ConstraintFlag** determines if there are constraints or not. If it is set to be false, our algorithm will work as an unconstrained situation. **SearchPatternWhen-NoConstraintNearby** denotes the search pattern our algorithm use when no active linear constraint is detected in the current iteration. Normally it is set to be “compass”.

Note that to test our implementation, we perform optimization problem not only on three-dimensional models, but also on a two-dimensional model, in which there is no Z dimension. Therefore, we add extra options to determine the actual dimensions for each point and well type, indicated by **ActualDimensionsForEachPoint** and **WellType** in table 3.1. For example, if we want the well to be vertical well in the two-dimensional model, the **actual_dimensions_for_each_point** and **welltype_** should be set to be 2 and “Vertical” respectively. Note that such option is used only for the test in our two-dimensional model. For the normal optimization problem, The two parameter should always be set to be 3 and “Horizontal”.

In appendix B, we provide an example driver file. Also, an example configuration of our algorithm can be found in the driver file.

Chapter 4

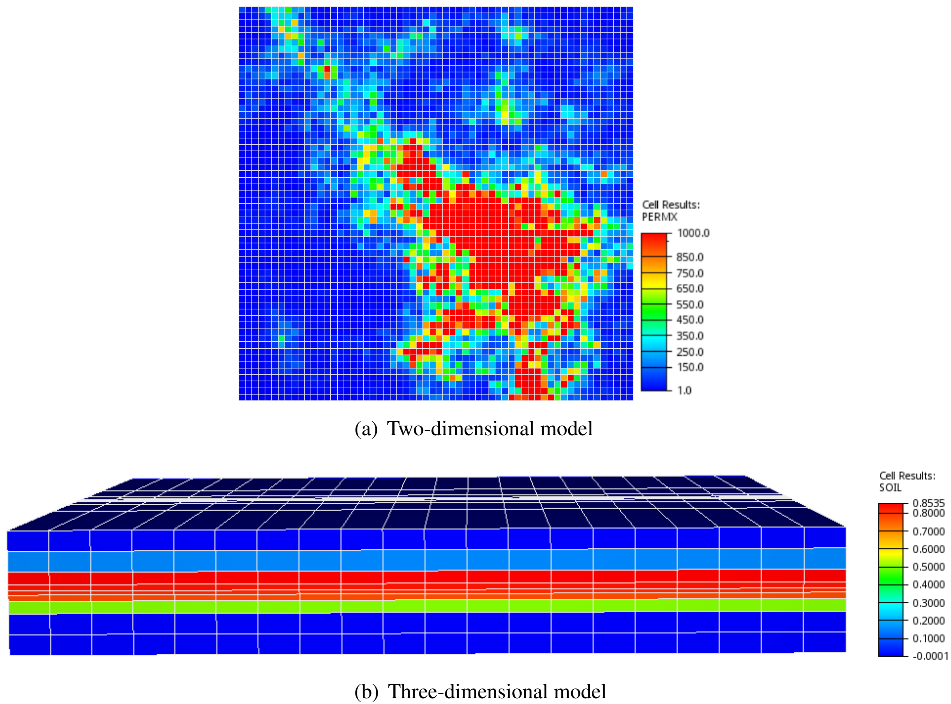
Case Study

In this chapter, we perform different case studies using GSS method with linear constraints. The reservoir model used is described in the following section. We will always compare the results with the built-in compass search method. We expect a better performance because our method adds more features on the basis of compass search. A large number of tests have been made and in this chapter. The most representative cases are shown and discussed. The aim of this chapter is, on the one hand, to verify that our implementation of the algorithm is correct, and on the other hand to test the performance of GSS algorithm with linear constraints.

4.1 Introduction of the two test reservoir models

Two-dimensional reservoir model The two-dimensional model is shown in figure 4.1(a). It consists of 60*60*1 grid blocks. Each grid block has dimensions of $24m \times 24m \times 24m$. The reservoir is 100% saturated with oil. For the most part of the reservoir, the porosity varied from 16% to 36% and permeability varied from $1mD$ to $1000mD$.

Three-dimensional reservoir model The three-dimensional model is shown in figure 4.1(b). It consists of $20 \times 9 \times 9$ grid blocks. The grid blocks are not uniform, making the reservoir model to be $2000ft \times 990ft \times 300ft$. The reservoir has a uniform 25% porosity, and a uniform permeability, which are $100mD, 100mD, 5mD$ for kx, ky and kz. In the Z direction, the first layer is gas layer, and the last two layers are water layers which contain no oil. The second layer is the oil-gas transition layer which contains a little amount of oil. Oil saturation in the other layers is very high.



(a) Two-dimensional model

(b) Three-dimensional model

Figure 4.1: Two test reservoir models

4.2 Expansion factor and contraction factor picking strategy

In this section, we perform four cases with different expansion factor and contraction factor. We change the expansion factor and contraction factor for each case and keep the other configurations the same. The aim is to find the most appropriate strategy for picking expansion factor and contraction factor by comparing the results of these different cases.

4.2.1 Case Description

The reservoir model used in this section is the two-dimensional model described above. For the test in this section, we consider four wells, two producers and two injectors. They are both vertical wells. The two injectors are fixed at bottom left corner and upper right corner. The two producers can be positioned anywhere inside reservoir, which means the optimizer has four variables to handle. The four wells are all in bottom hole pressure control mode. The bottom hole pressure of the injector at bottom left corner and upper right corner are kept all the time at $500bar$ and $250bar$ correspondingly and the wells are kept open from the first day of the simulation. The bottom hole pressure of one producer is maintained at $100bar$ and the other one kept at $80bar$. The simulation time for all the cases

is $1500days$. For the optimizer, the initial step length is 300m and minimum step length is set to be $10m$. The search pattern used is configured to be the Fast pattern introduced in chapter 2. The bookkeeper tolerance is set to be $5m$, and the objective function is set to be $FOPT - 0.2 * FWPT$. In this way, the initial objective function value is 1211840. All the configurations that are kept constant are summarized in table 4.1.

Table 4.1: Configurations for the two-dimensional model

Parameter	Value
Control mode for injector	BHP
Control mode for producer	BHP
Simulation time (days)	1500
Initial step length (m)	300
Minimum step length (m)	10
Bookkeeper tolerance (m)	5
SplinePoint-PROD1-x-Init (m)	400
SplinePoint-PROD1-y-Init (m)	1000
SplinePoint-PROD2-x-Init (m)	500
SplinePoint-PROD2-y-Init (m)	900
Initial objective function value	1211840

Table 4.2: Results of case 1-4

case	1	2	3	4
Expansion factor	2	2	2.5	4
Contraction factor	0.5	0.6	0.5	0.8
Final objective function value	1242690	1250250	1252510	1252470
Total number of reservoir simulations	169	205	247	721
Simulated reservoir simulations	158	205	243	540
Bookkeeped reservoir simulations	11	0	4	181
SplinePoint-PROD1-x	400	498.8	462.1	464.8
SplinePoint-PROD1-y	1000	974.9	972.7	965.2
SplinePoint-PROD2-x	1175	1079.1	1098.4	1123.2
SplinePoint-PROD2-y	562.5	672.7	675.9	666.9

4.2.2 Optimization Solution

Table 4.2 shows the results of the four cases and figure 4.2 shows all the searched points during the optimization process for each case graphically. The grid blocks in the background denote the reservoir grid blocks. The red circle and blue square denote the position of PROD1 and PROD2, respectively. The marker filled in black refers to the initial position, and the magnified marker filled in white refers to the solution of the optimizer.

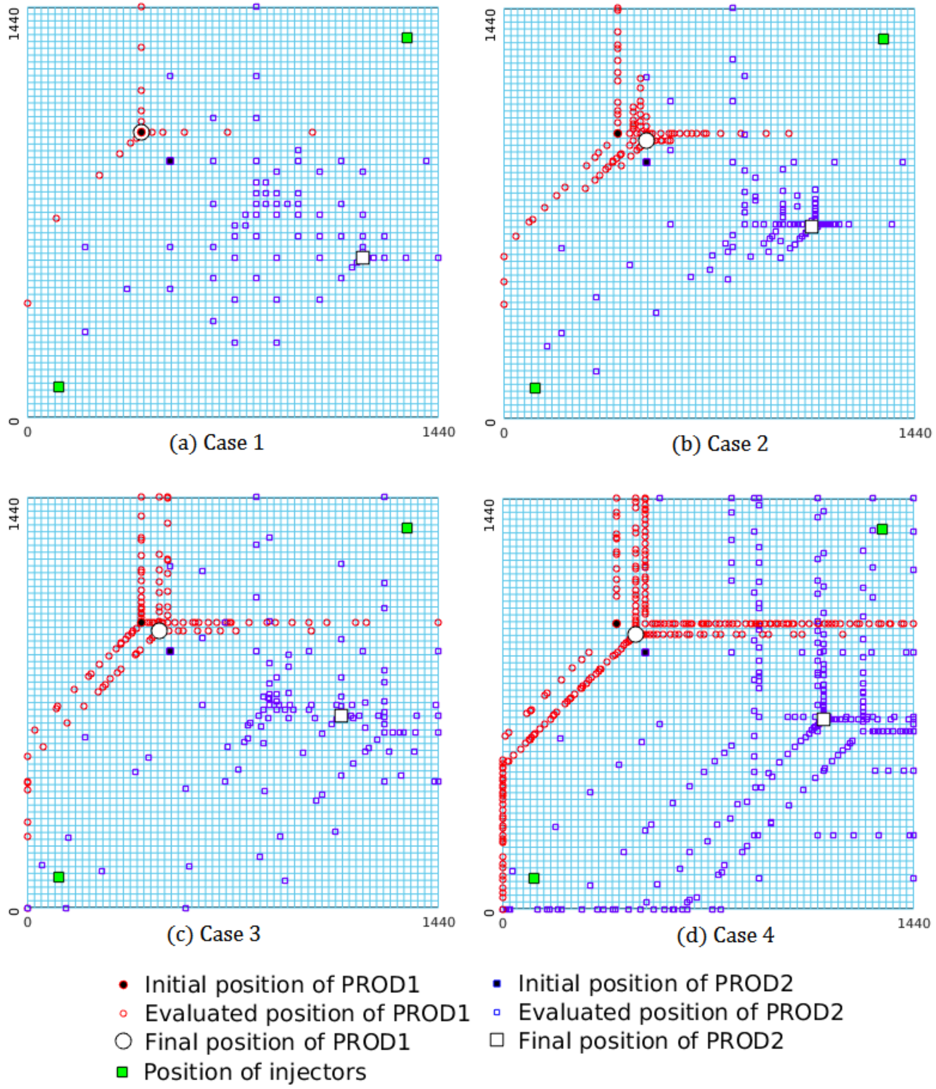


Figure 4.2: Search map of case 1-4

The results in table 4.2 show that the GSS method can be easily get trapped into the local optimum in this reservoir model due to its searching feature. When the expansion factor and contraction factor changes, The optimizer can get different results. While comparing case 1 with case 2 and 3, we see that higher expansion factor and contraction factor can lead to a better solution. The reason is that with higher factor, the optimizer can always search more points before it reaches the minimum step length.

We can draw the same conclusion from the search map for case 1. Here, the position of PROD1 is trapped into the initial position from the beginning. Therefore, while the tentative best position of PROD2 keeps changing during the optimization process, the tentative best position of PROD1 stays at the initial point, resulting in a bad solution. For case 2 and case 3, even though we know that the initial position of PROD1 is a local optimum hard to jump out, thanks to the higher expansion and contraction factor, the optimizer now has the chance to search more points to move away from it. It is evident that case 2 and case 3 went through a larger portion of the whole reservoir, especially for case 3. In consequence, they both find better solution than case 1. So in general, the more points the algorithm searches, the more likely it has a chance to find a better optimum.

However, that is not always the case. For example, for case 4, which has the highest expansion factor and contraction factor, even though the total number of cases evaluated is almost three times that of the other three, its objective function value is still less than case 2 and case 3. From figure 4.2, we can also see that actually the solution of case 4 is pretty close to case 3, but it just never reaches the position of case 3 even though we reduce the bookkeeper tolerance and minimum step length of the optimizer. There are mainly two reasons. The first reason is the search pattern we select has only three search directions, which are too few. The second reason is due to the huge geological heterogeneity of the reservoir model shown in figure 4.1(a).

To be more specific, in order to reach the solution of case 3, the position of PROD1 has to move a little bit upward, and the position of PROD2 has to move a little bit upper left. It is easy for PROD1 to go upward because we have the search pattern pointing upwards. But it is not easy for PROD2 to move upper left because the search pattern contains directions that only point upward and bottom left. Therefore, in order to move left, it has to move downward at the same time, which is against the direction we want. Therefore, in order to reach upper left, it needs at least two steps: the first step to move a little bit bottom left and the second step to move a little bit more upper.

Moreover, since it can not find it in one iteration, several steps needed means that it has to find a new tentative best cases in each step to reach the last optimum. However, the huge geological heterogeneity of the reservoir model will normally make the objective function largely non-smooth. In consequence, the tentative best cases required are always hard to be captured on the rough surface, or even not exist, making the process even harder.

Note that for case 4 in figure 4.2, which has a high expansion factor, many points reach the boundaries. Therefore, if we select large expansion factor, we need good constraint handling technique to avoid generating so many points on the boundaries. The results of using our constraint handling technique show a better performance, which are listed in table 4.4 and will be discussed later.

A large number of tests has been done besides the four cases above. We can sum up the following picking strategies for expansion factor and contraction factor: First, no

strict mathematical law can be found between the value of the factor we pick and the fact whether the optimizer can find global optimum or not due to the irregularity of the reservoir properties. But in general, the higher the contraction and expansion factors are, the more likely the optimizer can find better solution. Unfortunately, greater expansion and contraction factor will take more time since it will definitely generate more cases for simulator to run, whether or not it can find better solution.

We can also look at the issue from another perspective. As stated above, for example, in figure 4.2(a), we observe that the optimizer does not touch a large portion of the reservoir. Therefore, We can simply put the initial position to the area that is not searched and run the optimizer again. Usually this is the easiest way to get rid of being trapped into the local optimum.

In summary, our recommendation is that it is better to pick the two factor as high as possible under the limited computing resources. If we do not satisfy with the result, just rerun the optimizer with several different starting points.

4.3 Comparison between two patterns

In this section, we perform another four cases with the same optimization configuration as table 4.1 in last section except using the Compass pattern introduced in chapter 2. The reservoir model is still the two-dimensional model. The aim of this section is to compare the performance of the two search pattern, on the basis of which we can find out their advantages and disadvantages.

4.3.1 Optimization Solution

Table 4.3: Results of case 5-8

Case	5	6	7	8
Expansion factor	2	2	2.5	4
Contraction factor	0.5	0.6	0.5	0.8
Final objective function value	1242690	1323330	1322960	1323330
Total number of reservoir simulations	217	361	297	1017
Simulated reservoir simulations	168	318	264	762
Bookkeeped reservoir simulations	49	43	33	255
SplinePoint-PROD1-x	400	741	768.7	738.7
SplinePoint-PROD1-y	1000	625.9	601.7	630.8
SplinePoint-PROD2-x	1175	1129.4	1132	1132.5
SplinePoint-PROD2-y	562.5	584.6	618.7	586.8

Table 4.3 summarizes the results of the four new cases. In the table we see GSS with Compass pattern will always simulate more cases before finishing the optimization process. However, this can also reduce the possibility of getting trapped into local optimum to some extent. Except case 5, which also trapped into the same local optimum with case 1,

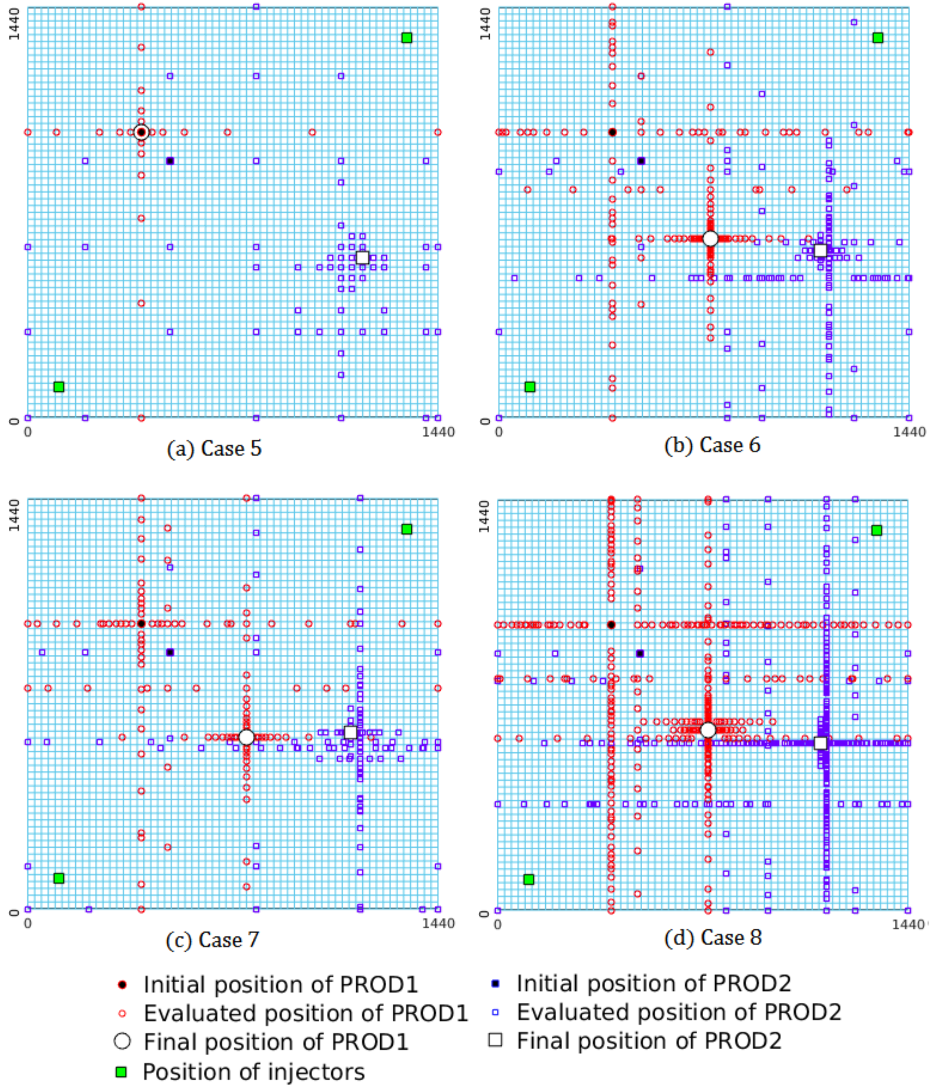


Figure 4.3: Search map of case 5-8

the other three cases have a clear improvement on the objective function value compared with case 1 - 4. From the total number of cases in table 4.3 we can find that GSS with Compass pattern has about 1/4 more cases than GSS with Fast pattern. This is because, in one iteration, GSS with Compass pattern will generate four new points for each well, while GSS with Fast pattern will generate three. It is for this reason that the search can jump out of the local optimum which trapped case 1 to case 4. To be more specific, In figure 4.3, the main difference between case 6, 7, 8 and case 2, 3, 4 is the position of PROD1 which is denoted by red circle. In case 2, 3 and 4, due to the small number of search directions in each iteration, the optimal position of PROD1 can only search in the area around the initial position. That is why although the objective function value is getting improved from case 2 to case 4, it still stays in the level of 1.25e6 but never reaches 1.32e6 as case 6, 7, 8 did. With the help of the additional search directions PROD1 is able to move out of the small area which trapped case 2 to 4. And finally, the algorithm found a better solution.

In summary, the more number of search directions the pattern has and the more various the search they are, the less possible the point will get trapped into the local optimum. Although it is faster using Fast pattern, it is still not recommended because most real reservoir conditions can be more complicated than the two-dimensional model. Three search directions for each point are too few to be representative enough for the conditions around the point. In other words, we cannot be convinced that there is no better solution around the point if the search along only three directions cannot find a better solution. We recommended that at least the Compass pattern, or other pattern containing more search directions than Compass pattern, to be applied to the well placement optimization problems.

4.4 Optimization for two-dimensional model using GSS with linear constraints

In this section, we perform the same optimization problem using GSS with linear constraints. Firstly, how to determine the linear constraints is discussed. Then, the first three iterations of one of the new case we propose are illustrated graphically in order to visualize the searching process and verify our implementation is correct. The results are discussed in detail. We expect the same or better results compared with case 1-8 which have no linear constraints. Finally, the interpretation of the results from reservoir point of view is discussed.

4.4.1 Linear constraints picking strategy

The aim of the linear constraints is basically to reduce the search area so that it will be more likely to find the optimum with fewer iterations. The more specific the feasible region is, the fewer iterations the optimizer will have before finding the optimum. But in order not to exclude the global optimum out of our feasible region, we have to pick it carefully.

Normally our first concern is to make sure that all the producers should be located at oil zone, not water zone. In this model, we have no water zone. All grid blocks are fully saturated with oil. Almost no water exists in the reservoir at all. Therefore, we do not

need to worry about it in this model. Another concern is that we should keep producers far away from the injector to avoid the early water breakthrough. In this model, The two water injectors are located at the bottom left corner and upper right corner. So it is better to keep our producers somewhere around the central part of the reservoir.

The most important thing to configure this model is, if we consider the simulation time, 1500 days. It is not long enough for the injected water to breakthrough as long as the producer is not too close to the injector. So the optimization problem actually becomes how to produce as much oil as possible within the limited production time. In this sense, our focus is to place the producers at the position which has very good reservoir properties in order to produce more oil. Looking at the permeability of the reservoir shown in figure 4.1(a), we found that the permeability around the main diagonal of the reservoir tends to be higher than anywhere else. Higher permeability means that producers have higher production rate according to Darcy's law. Higher porosity means this place has more oil than anywhere else. Therefore, this part is our target region.

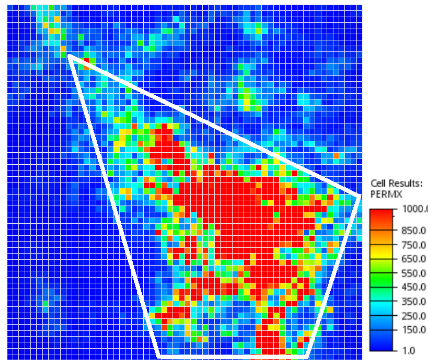


Figure 4.4: Linear constraints for two-dimensional model

$$\begin{bmatrix} 0.4831 & 1 & 0 & 0 \\ 3.0455 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ -3.4444 & -1 & 0 & 0 \\ 0 & 0 & 0.4831 & 1 \\ 0 & 0 & 3.0455 & -1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -3.4444 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} \leq \begin{bmatrix} 1366 \\ 3645 \\ -10 \\ -2077 \\ 1366 \\ 3645 \\ -10 \\ -2077 \end{bmatrix}$$

Figure 4.5: Linear constraints proposed for the two-dimensional model

In summary, taking into account all the factors above, we propose the following linear constraints illustrated in figure 4.4. The linear constraints in matrix form is presented in figure 4.5, where the huge matrix in the left is our linear constraint matrix denoted by A

in equation 2.1 in chapter 2. The matrix in the middle is the variable matrix where x_1, y_1 denote the position of PROD1. x_2, y_2 denote the position of PROD2. The matrix in the right denotes u in equation 2.1.

4.4.2 An illustrated example

In this section, we illustrate the first five iterations of case 11 in table 4.4, which corresponds to case 3 and case 7, with expansion factor equal to 2.5 and contraction factor equal to 0.5. The aim is to verify the correctness of our implementation and give a visualized understanding of the method.

The first iteration is shown in figure 4.6(a). The light blue grid blocks in the background denote the reservoir. The feasible region is demonstrated by the quadrilateral. The constraints in dashed line are the active constraints. We calculate the search directions (shown as lines emanating from the current best point to corresponding trial points) based on the active constraints. The red circle and blue rectangle denote the position of PROD1 and PROD2 respectively. The marker in bigger size refers to the tentative best case in this iteration. For the first iteration, two constraints are the active constraints. (Actually, those are four constraints. The first, fourth, fifth and eighth lines in the constraint matrix in figure 4.5. They overlapped because PROD1 and PROD2 have the same constraints.) According to the algorithm 1 in chapter 2, they both have two search directions which are parallel to the constraints, indicating four new cases in this iteration. The first iteration is a successful iteration because it found a new tentative best case, which is marked by the bigger sign in the figure. After the first iteration, PROD1 changed to a new position with PROD2 left unchanged.

Figure 4.6(b) shows the next iteration. The step length has updated to 750 since the last iteration is successful. The same two constraints remain active. Everything is the same with the first iteration except the step length. This iteration is successful since a better case is detected as shown.

In Figure 4.6(c), the optimizer reaches the third iteration, where step length becomes 1875. But in this case, the step length is too long that the distance between the current best point and all the constraints is less than the step length, which means all the four linear constraints become active constraints. According to the algorithm 1 in chapter 2, the degenerate case occurs, in which the optimizer cannot find the correct search direction. How to deal with the degenerate case is illustrated in the next figure.

According to our degenerate case handler, the optimizer will keep multiplying the step length by the contraction factor until the appearing of the nondegenerate case. In this situation, the step length has been reduced twice to 468.75. Under such step length, three new constraints become active constraints (Actually four constraints, two for the heel and two for the toe, denoted by the first and forth lines in the constraint matrix for PROD1, and the fifth and sixth lines in the constraint matrix for PROD2). In consequence, the search directions also changed as shown in figure 4.6(d). Unfortunately, this iteration is unsuccessful since no better point is found.

Figure 4.6(e) illustrates the fourth iteration. Again we have new search directions conform to the new active linear constraints. Note that in this iteration we have only one active constraint for PROD1, which is the first constraint in the matrix. The one active constraint leads to three search directions, two of which are parallel to the linear constraint

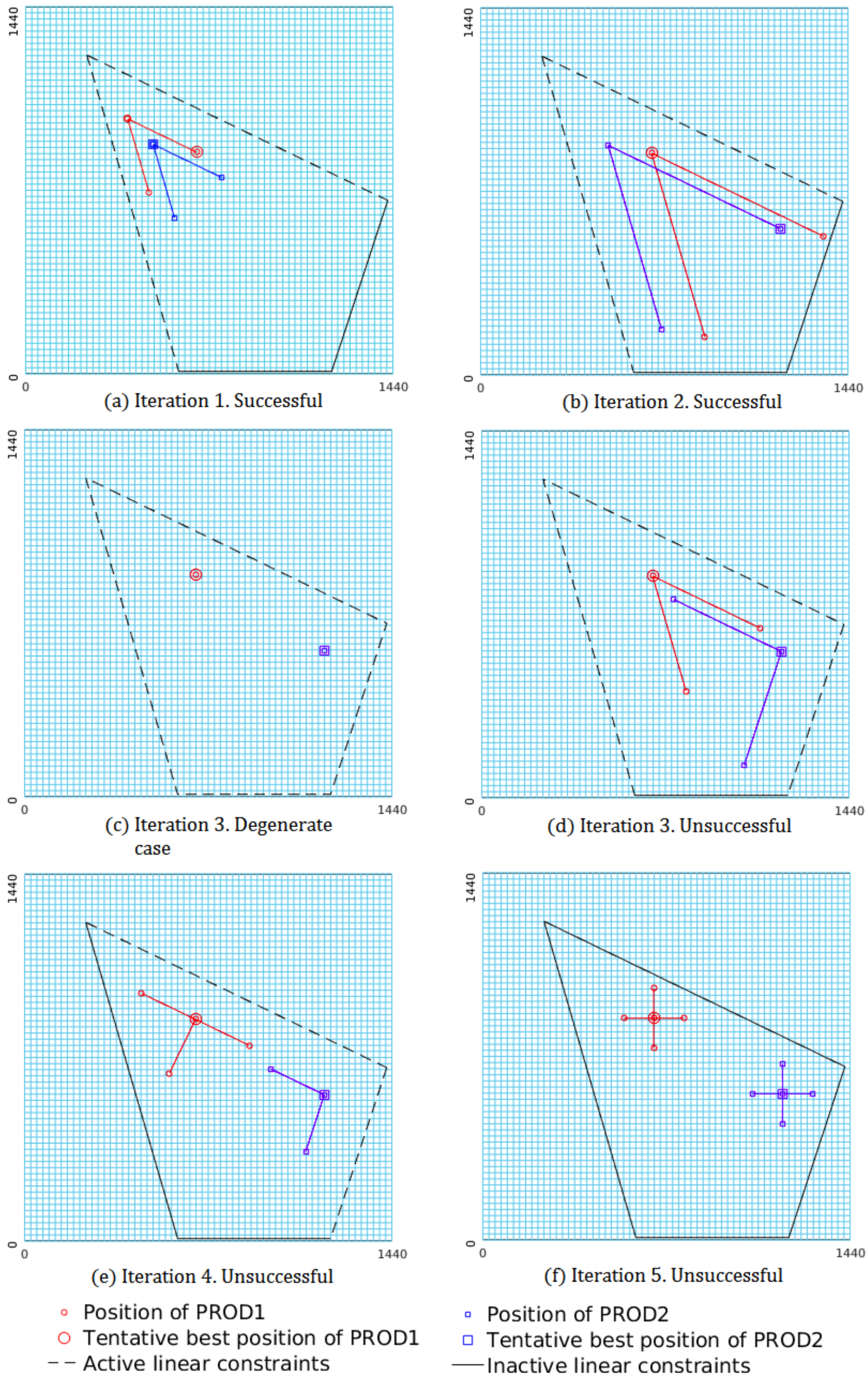


Figure 4.6: An illustrative example of the iterations for two-dimensional case

and the other one is perpendicular to it. This iteration is also unsuccessful without any better point.

Figure 4.6(f) shows the fifth iteration. In this iteration, the step length has been reduced to 117.19 because of the two consecutive unsuccessful iterations. The short step length leads to the situation that no active linear constraint is nearby. The algorithm will continue searching as unconstrained GSS algorithm. We set the search pattern for unconstrained cases to be Compass pattern. Therefore, four new points will be generated for each well. And the iterations will continue.

4.4.3 Optimization Solution

Table 4.4: Results of case 9-12

case	9	10	11	12
Expansion factor	2	2	2.5	4
Contraction factor	0.5	0.6	0.5	0.8
Final objective function value	1318510	1323330	1323360	1323360
Total number of reservoir simulations	190	183	194	784
Simulated reservoir simulations	163	183	194	782
Bookkeeped reservoir simulations	27	0	0	2
SplinePoint-PROD1-x	707.6	739.7	727.3	743.8
SplinePoint-PROD1-y	794.5	633.3	634.8	626.3
SplinePoint-PROD2-x	1096.4	1138.3	1088.7	1082.7
SplinePoint-PROD2-y	657.7	595.3	704.1	710.3

From the results in table 4.4, it is evident that overall GSS algorithm with linear constraints gives better results compared with all cases above. All the four new cases get better solution compared with the corresponding cases before. Furthermore, the most important thing is that there is a huge reduction in the number of simulations, which significantly increases the efficiency of the optimization.

We can sum up two main advantages of using linear constraints compared with all the previous cases. Firstly, with the restriction of the linear constraints, the searching process can only be carried out in a smaller region, which can get rid of plenty of unnecessary points. In figure 4.3, we found that it evaluated too many points near the boundary of the reservoir, or close to the injector. These points are definitely unnecessary points which cost a lot of time. However, the amount of searched points when using linear constraints is much less as shown in figure 4.7. Secondly, with the linear constraints nearby, the search directions can always change iterations by iterations. For example, for iteration 3, 4 and 5 in figure 4.6, we can see that for PROD1, all the nine new points are generated in different directions. Without linear constraints, the search process can be only in the four directions along the coordinate axis with Compass pattern, or three directions with the Fast pattern. We believe that the more different search directions it has, the more possible it can find new better points.

Note that for case 9 in figure 4.7, PROD1 no longer gets trapped in the initial position

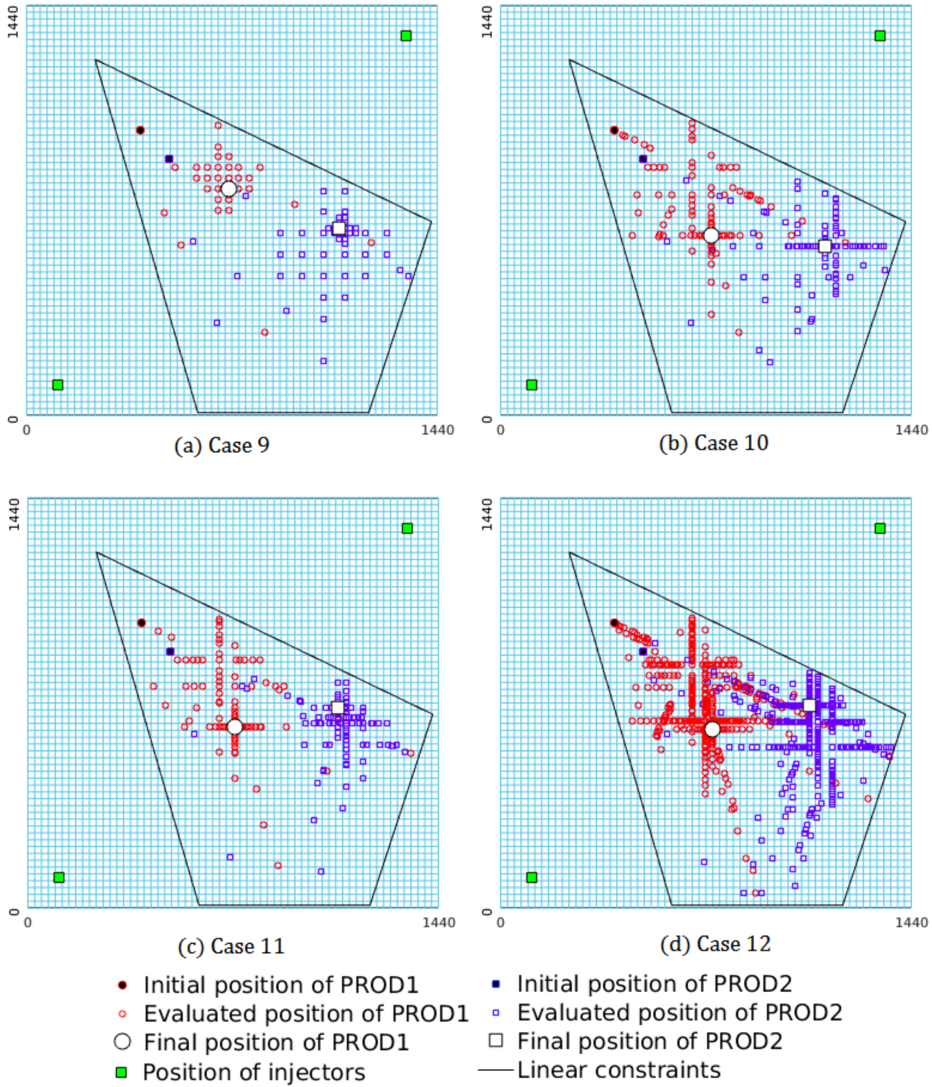


Figure 4.7: Search map of case 9-12

as case 1 and case 5 does. This is due to the variety of search directions as we stated above. Even though it simulated only 163 cases in total, its objective function value is even higher than the best case we have when using the Fast pattern without constraints. Case 10 has already found the same best result as table 4.3 found, with only 183 simulations simulated. While case 6, which use unconstrained Compass pattern, simulates 318 times until it found the same result. Furthermore, case 11 found even better results than the best result in table 4.3 with only 194 simulations, indicating a much better performance than not using linear constraints. An even higher expansion factor and contraction factor is carried out in case 12, and no better result is found.

One interesting thing we found in figure 4.7 is that, the area covered by very dense searched points has higher permeability than the area not covered by many searched points. For example, for the search map of case 12 in figure 4.7, the bottom left part and bottom right part of the feasible region, which has almost no searched point, corresponds to pretty low permeability region in figure 4.4. This relationship also meets with our linear constraints picking strategy illustrated above.

4.4.4 Comparison between all the cases

Table 4.5 summarizes the number of simulations and objective function value for case 5-8 and case 9-12. Both the two groups of cases use Compass pattern. (case 5-8 uses unconstrained Compass pattern. case 9-12 uses Compass pattern when no linear constraint is nearby correspondingly.) It is more clear to see the effect of linear constraints in this table. As stated above, we find that using linear constraints can always find a better solution with less number of iterations.

Table 4.5: Comparison between unconstrained and constrained condition

Condition	Case	Number of simulations	Objective function value
Unconstrained	5	217	1242690
	6	361	1323330
	7	297	1322960
	8	1017	1323330
Constrained	9	190	1318510
	10	183	1323330
	11	194	1323360
	12	784	1323360

Figure 4.8 gives the optimization curves for all cases. The curves in different shades of blue denote case 1-4, which use unconstrained Fast pattern. Curves in different shades of green denote case 5-8, which use unconstrained Compass pattern. Curves in different shades of red denote case 9-12, which use linear constraints. It is clear to see that case 1-4 cannot find satisfactory solutions compared with others. Their curves always lie in the bottom half of the figure. Although case 5-8 find good solutions, the number of simulations is too many. We got pretty satisfactory results from case 9-11, which has good solutions and also few number of simulations. The curves of case 9-11 lie in the pretty upper-left

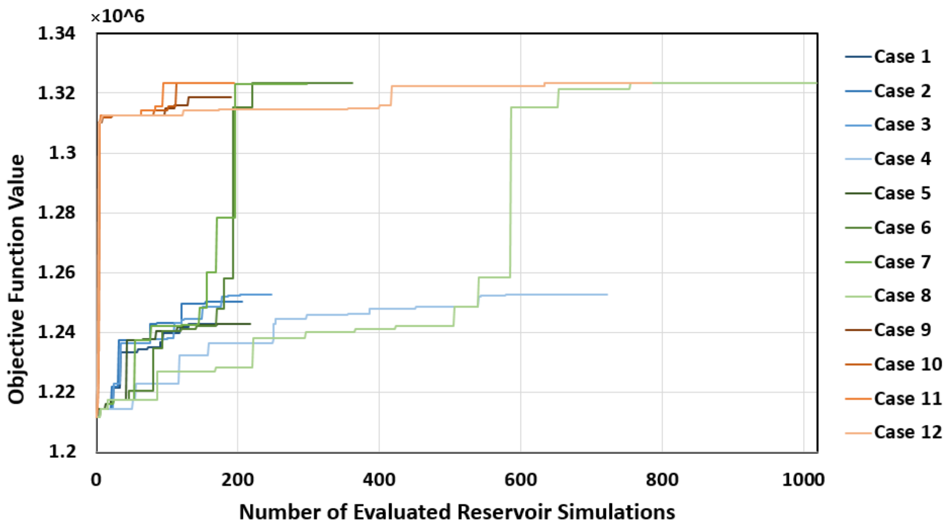


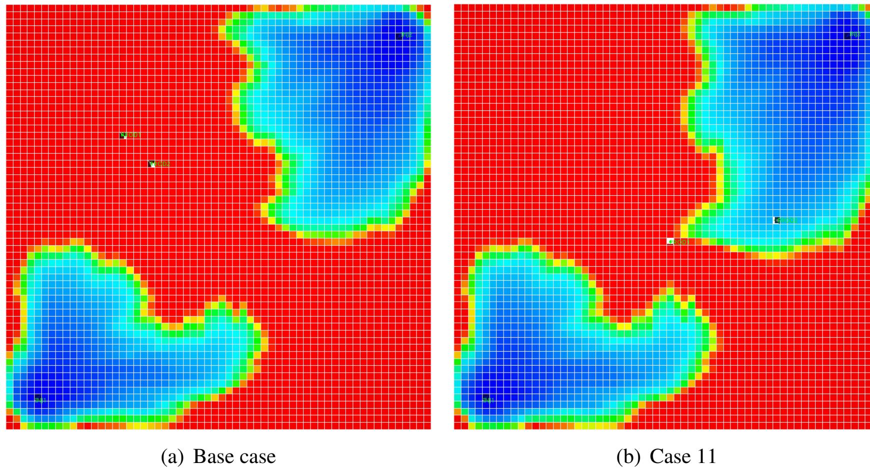
Figure 4.8: Number of simulations VS objective function value

side of the figure, indicating high objective function value and few number of iterations. Case 12 is just for the test since it has too high expansion and contraction factor.

4.4.5 Interpretation of the solution

Figure 4.9 displays the oil saturation distribution at the end of the production time frame for the base case and case 11, the best case we got. Figure 4.10 presents the cumulative oil production and water injection for the field and for the two producers. Since both cases have no water breakthrough, cumulative water production can be neglected. The results show that our optimized well placement, case 11, has a slightly better reservoir sweep efficiency than the base case. We notice that the difference is not too great. The reason is that on the one hand, the initial position of the producer is in the region of our selected area defined by linear constraints, which means it is already a pretty good position. On the other hand, the short control time and small size of our model always lead to the similar results for different locations of wells. The results can be seen much more clear in figure 4.10(a), which shows that our optimized well placement has a higher cumulative oil production than base case.

Note that one of the two producers seems to be in the water-swept area in figure 4.9(b). Actually, that well shuts very early before the injected water reaches its position as presented in figure 4.10(c). The reason why it shuts so early is still due to the small size of the reservoir model. Since the two producers are controlled by different BHP, when they are both in the pretty high permeability zone and the distance between them is short, the reservoir pressure in both place will always tend to become the value of the lower BHP. Therefore, the well controlled by the higher BHP will automatically shut according to the ECLIPSE simulator. But still, since the aim of our cases in this chapter is just to test the



(a) Base case (b) Case 11

Figure 4.9: Oil distribution of the two-dimensional model

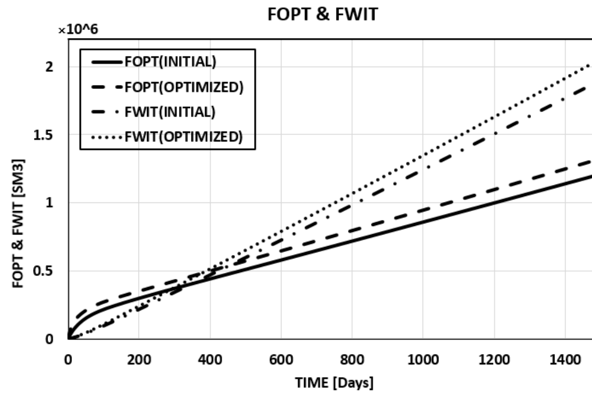
performance of our algorithm, we do not care too much about if it is implementable in practice or not.

4.5 Optimization for three-dimensional model using GSS with linear constraints

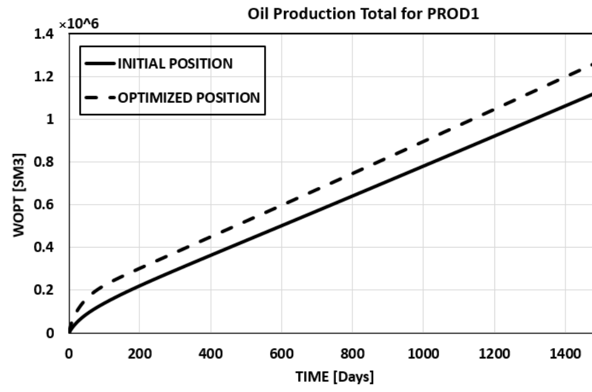
In this section, we perform two optimizations for the three-dimensional model, the first one using unconstrained Compass pattern without linear constraints and scaling factor, and the other one with linear constraints and scaling factor. (Also Compass pattern when no linear constraint is nearby.) The aim of this section is to explain how our method works for the three-dimensional model, and then to test its performance by setting the proper linear constraints and scaling factors.

4.5.1 Case description

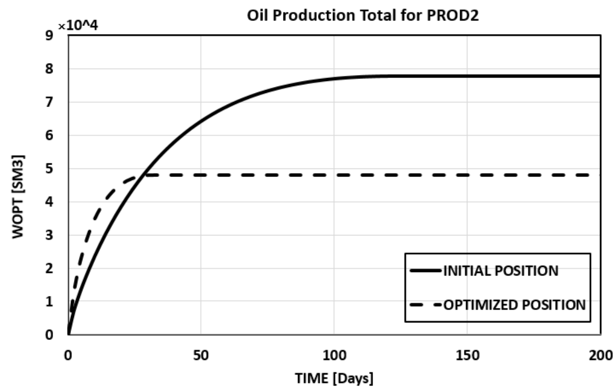
The reservoir model used in this section is the three-dimensional model described at the beginning of this chapter. In this section, we consider two wells, one horizontal injector and one horizontal producer. The injector is fixed at southwest corner with all nine grids blocks in the north direction perforated at bottom layer. Therefore the optimization problem has six variables to handle. The injector is controlled by a constant bottom hole pressure at $4100psi$. The producer is controlled by a constant bottom hole pressure with an upper limit of liquid rate, which is $1500psi$ and $1000rb/day$ respectively. The simulation time is $2400days$. The initial objective function value is 366027. All the necessary configurations for the model and optimizer are summarized in table 4.6. The initial well position for injector and producer is presented in figure 4.11. The two cases we perform will be discussed later.



(a) Cumulative oil production and water injector for the whole field



(b) Cumulative oil production for PROD1



(c) Cumulative oil production for PROD2

Figure 4.10: Comparison between the base case and the best optimized case, case 11

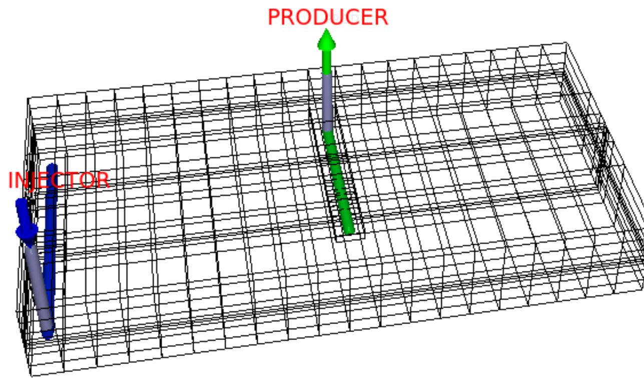


Figure 4.11: Initial position of the wells

Table 4.6: Configurations for the three-dimensional model

Parameter	Value
Control mode for injector	BHP
Control mode for producer	BHP
Simulation time (days)	2400
Initial step length (ft)	400
Minimum step length (ft)	5
Bookkeeper tolerance (ft)	4
Expansion factor	2.5
Contraction factor	0.5
SplinePoint-PROD-heel-x-Init (ft)	1050
SplinePoint-PROD-heel-y-Init (ft)	840
SplinePoint-PROD-heel-z-Init (ft)	7180
SplinePoint-PROD-toe-x-Init (ft)	1050
SplinePoint-PROD-toe-y-Init (ft)	360
SplinePoint-PROD-toe-z-Init (ft)	7180
Initial objective function value	366027

4.5.2 Optimization without linear constraints and scaling factor

Case 13 is the one using GSS method with Compass pattern. For this case, neither linear constraints nor scaling factors are used. Table 4.7 summarizes the results of case 13. Overall the result is good as the objective function value increased significantly from 366027 to 987446. However, we found some parts that could still be improved. Firstly, the total number of iterations is too many. According to the search map shown in figure 4.12 we observe that we have too many searched points at the boundary of the reservoir model. This is because without introducing scaling factor, the step length along each search direction is the same. However, the size of our reservoir model along different directions is

not in the same order of magnitude. Typically for a reservoir model, the size of the longitudinal dimension is much smaller than the horizontal dimension. Under such situation, the search along Z coordinate will always exceed the range of the reservoir model, which will be projected back to the boundary according to the FieldOpt's bound constraint handler. Therefore, scaling factors must be introduced to make each parameter in the same magnitude.

We observe that according to the result, the heel of the optimized producer is on the same side as the injector, and the toe of the optimized producer is on the other side. Such well placement strategy does not follow the principle of reservoir engineering because the too short distance between the heel of producer and injector will lead to early water breakthrough. In order to prevent the heel of producer from being too close to the injector, linear constraints need to be introduced to limit the range of feasible region of heel and toe of the producer.

Table 4.7: Result of case 13

case	13
description	GSS without constraints and scaling factor
Final objective function value	987446
Initial objective function value	366027
Total number of reservoir simulations	1309
Simulated reservoir simulations	1012
Bookkeeped reservoir simulations	297
SplinePoint-PROD-heel-x	24.5
SplinePoint-PROD-heel-y	858.1
SplinePoint-PROD-heel-z	7141.6
SplinePoint-PROD-toe-x	1968
SplinePoint-PROD-toe-y	258.4
SplinePoint-PROD-toe-z	7150.6

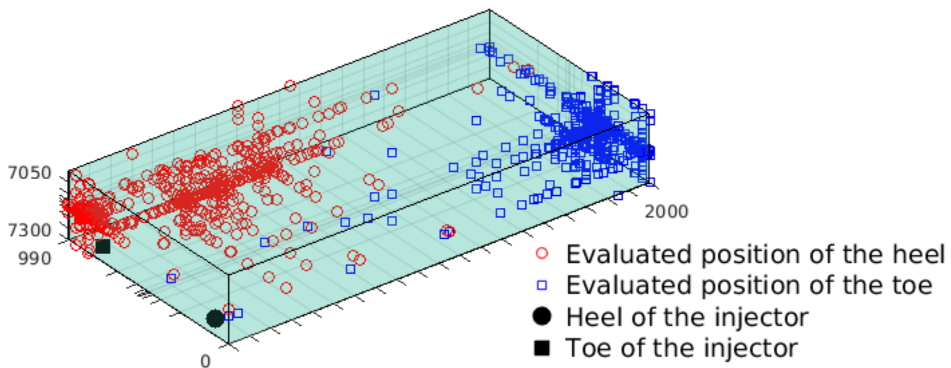


Figure 4.12: Search map of case 13

4.5.3 Linear constraints and scaling factor picking strategy

The reservoir model has nine layers in the z-direction. we observe that the first layer is gas layer and the last two layers are water layer. Therefore, in the z-direction, we propose bound constraints to make both the heel and toe only move in oil layer. Then we propose one linear constraint in the middle to separate the entire reservoir into two parts, the east part and the west part. In this way, the heel of producer can only move on the side which is away from the injector and the toe can only move on the other side of the reservoir. The other constraints in the matrix are the reservoir boundary in x and y directions.

In summary, The linear constraints we propose is listed in figure 4.14(a). The right part of the reservoir is the feasible domain for the heel of the producer. While the left part is for the toe. Furthermore, the linear constraints is shown in figure4.13 in the matrix form, where the huge matrix in the left is our linear constraint matrix denoted by A in equation 2.1 in chapter 2. The matrix in the middle is the variable matrix where subscript 1 denotes heel and subscript 2 denotes toe. The matrix in the right denotes u in equation 2.1.

$$\begin{bmatrix} -2.475 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.475 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix} \leq \begin{bmatrix} -2970 \\ 0 \\ 990 \\ 2000 \\ -7050 \\ 7200 \\ 2970 \\ 0 \\ 990 \\ 0 \\ -7050 \\ 7200 \end{bmatrix}$$

Figure 4.13: Linear constraints proposed for the three-dimensional model

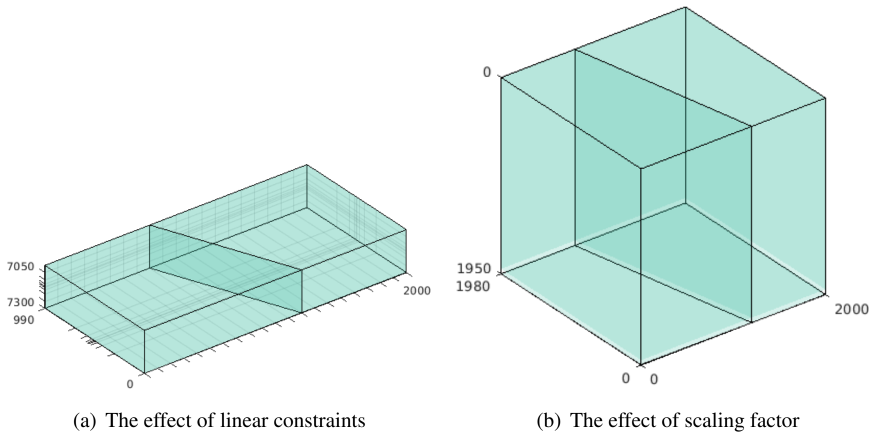
In order to make all variables in the same magnitude, the diagonal of the matrix D and the scaling factor c we pick are

$$D_{diag} = 1, 2, 13, 1, 2, 13$$

$$c = 0, 0, -91650, 0, 0, -91650$$

In this way, x is in the range between 0 and 2000, y is in the range between 0 and 1980, and z is in the range between 0 and 1950. It is evident that they are in the same magnitude now. Therefore, we transform the search space from figure 4.14(a) to figure 4.14(b).

In summary, this is how we define the linear constraints and scaling factor for the three-dimensional model. Case 14 is proposed by the configurations discussed above. The first three iterations of it is discussed in the next section.



(a) The effect of linear constraints (b) The effect of scaling factor

Figure 4.14: The effect of linear constraints and scaling factor

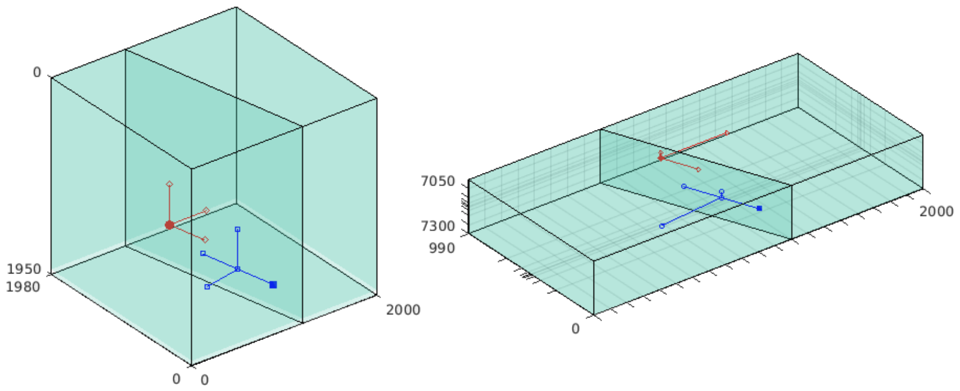
4.5.4 An illustrative example

In this section, we perform the first three iterations of case 14 as shown in figure 4.15, aiming at showing graphically how linear constraints and scaling factors work for this three-dimensional model.

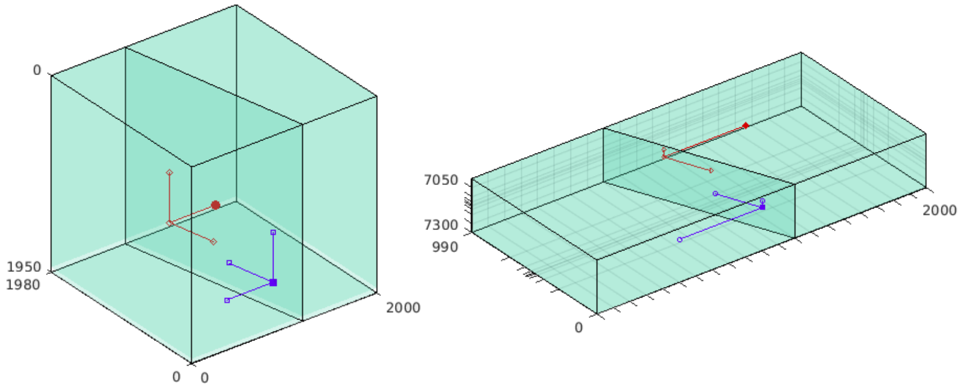
In figure 4.15, the left part is the reservoir model after scaling. As mentioned above, in this space, the three dimensions has the same magnitude. Actually, our searching algorithm is working in this space, and the new points are mapped back to the real reservoir model according to the one-to-one relationship between this space and the real model. In both spaces, the plane in the middle is the linear constraint that separates the feasible domain of heel and toe of producer. The red circle and blue rectangle denote the position of heel and toe respectively. The marker filled with color refers to the tentative best case in that iteration.

Figure 4.15(a) is the first iteration. The initial step length is 400ft, which makes the first, third, sixth, seventh and twelfth lines in the linear constraint matrix (figure 4.13) become active constraints. According to our algorithm, the starting point has seven search directions, three for heel and four for toe. As shown in figure, the step length is all the same for each search direction after scaling. However, when it is mapped to the real reservoir, the step length is longer in the x-direction and shorter in z-direction according to our scaling factor to better match the size of the reservoir. However, all the search directions are still parallel to the corresponding constraint plane, though the angle of the plane may change due to the scaling factor. The first iteration is successful since it found new better points which are filled with color.

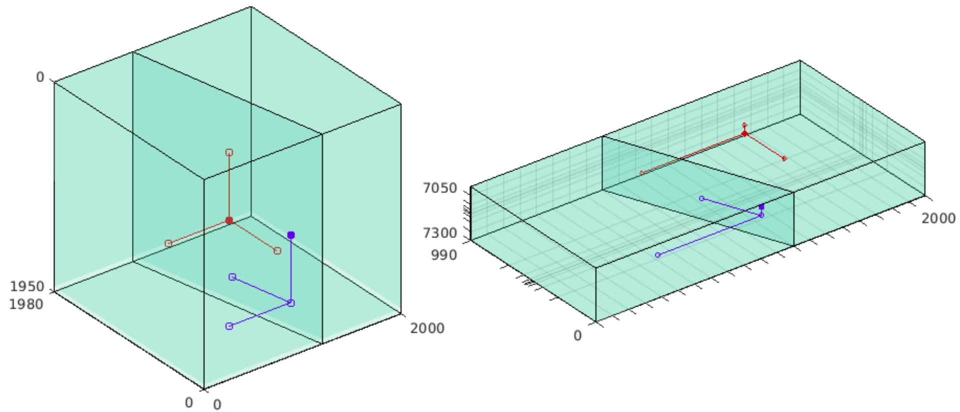
The second iteration is shown in figure 4.15(b). Since the first iteration is successful, the step length becomes 1000ft according to the expansion factor we set. Then, the degenerate case occurs because we have so many active constraints under this step length that the algorithm cannot find any search direction. According to our degenerate case handler introduced in chapter 2, the step length will be multiplied by the contraction factor until it



(a) Iteration 1. Successful



(b) Iteration 2. Successful



(c) Iteration 3. Successful

- ○ Position of the heel
- ● Tentative best position of the heel
- □ Position of the toe
- ■ Tentative best position of the toe

Figure 4.15: An illustrative example of the iterations for three-dimensional case

is no longer degenerate case. In this iteration, the step length is multiplied by the contraction factor one time to become $500ft$. For the second iteration, the active constraints are the first, third, sixth, seventh, eighth and twelfth lines in linear constraints matrix (figure 4.13). In consequence, six search directions are generated by the algorithm. Again, the second iteration is successful as shown.

The third iteration in figure 4.15(c) is mostly the same as the first two iterations, including the same successful iteration and degenerate case occurrence. But the active constraints in this iteration become the third, fourth, sixth, seventh, eighth and twelfth lines. There are six search directions correspondingly and the step length in this iteration is $625ft$.

4.5.5 Optimization Solution

The results of case 14 is presented in table 4.8. While comparing the results between table 4.7 and table 4.8, it is evident that case 14, which uses GSS with linear constraints and scaling factor, gave better results. Although case 14 is only 1.3% higher than case 13 in terms of the objective function value. There is a huge reduction in the number of simulations compared with case 13, from 1309 to 493. The reason, as implied above, is that by introducing linear constraints and scaling factor, a significant amount of unnecessary points can be avoided, such as the points in the water layer, points that are projected onto the reservoir boundary. Figure 4.16 shows the search map of case 14. Unlike the messy searching process in case 13 in figure 4.12, case 14 capture the area of best points very fast, and then find a solution by gradually contracting the step length.

Table 4.8: Result of case 14

case	14
Description	GSS with constraints and scaling factors
Final objective function value	1000420
Initial objective function value	366027
Total number of reservoir simulations	493
Simulated reservoir simulations	493
Bookkeeped reservoir simulations	0
SplinePoint-PROD-heel-x	1928.4
SplinePoint-PROD-heel-y	766.2
SplinePoint-PROD-heel-z	7144.2
SplinePoint-PROD-toe-x	150.7
SplinePoint-PROD-toe-y	0.9
SplinePoint-PROD-toe-z	7129.6

This can also be seen in figure 4.17, where case 14 outperforms case 13 in terms of the number of reservoir simulations. For case 14, the objective function value increases sharply at the beginning of the optimization. It has already been pretty close to the final solution within 200 runs. The curve of the following runs is very flat with a small improvement in objective function value. In contrast, the curve of case 13 increases slowly and gradually until the end.

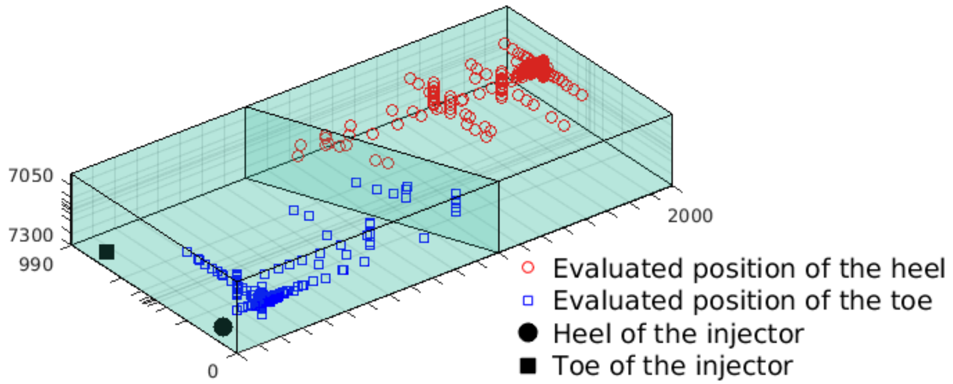


Figure 4.16: Search map of case 14

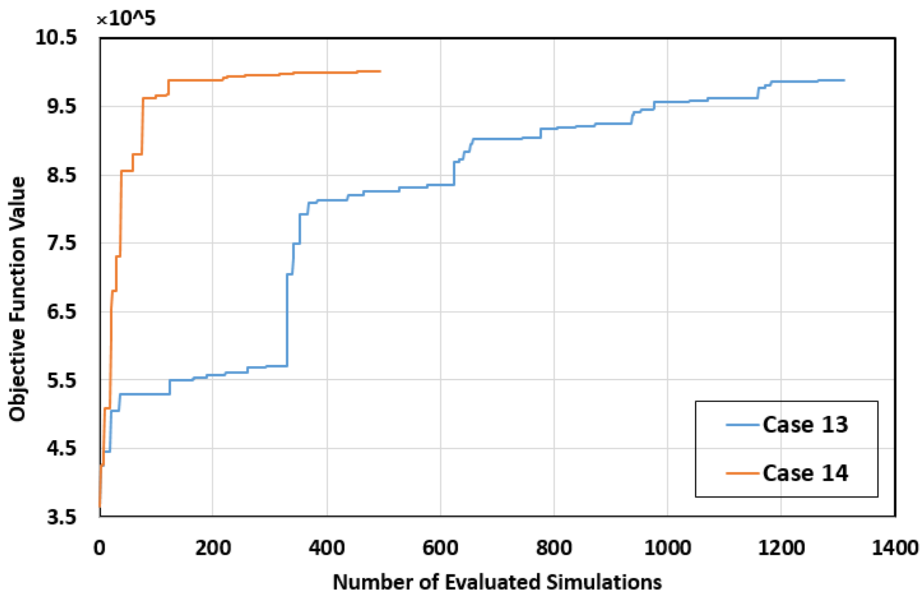
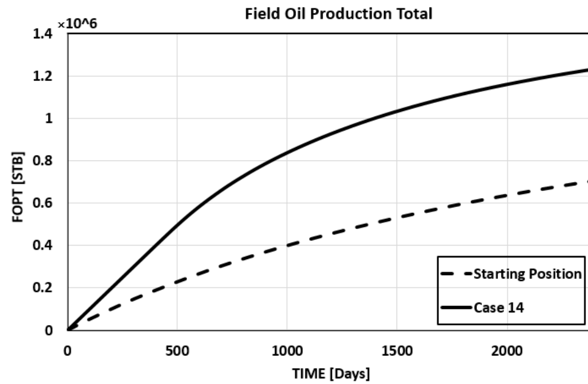
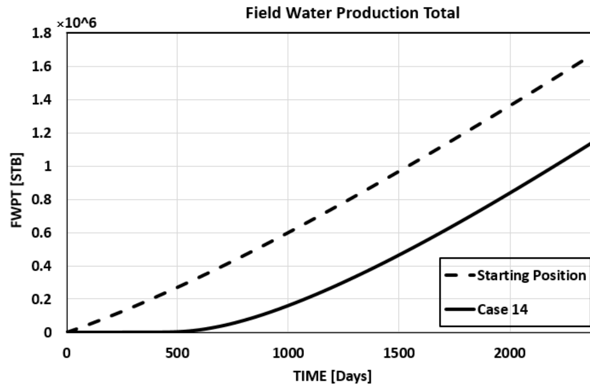


Figure 4.17: Comparison between the base case and optimized case

We perform the comparison between the initial position of the producer and case 14, the best position our algorithm has got. As can be seen in figure 4.18, case 14 has much higher cumulative oil production and much lower cumulative water production than initial position. Figure 4.19 displays the oil saturation distribution of both cases at the end of the production time frame. (Note that the first layer is excluded out since it can block the oil distribution in oil layer.)



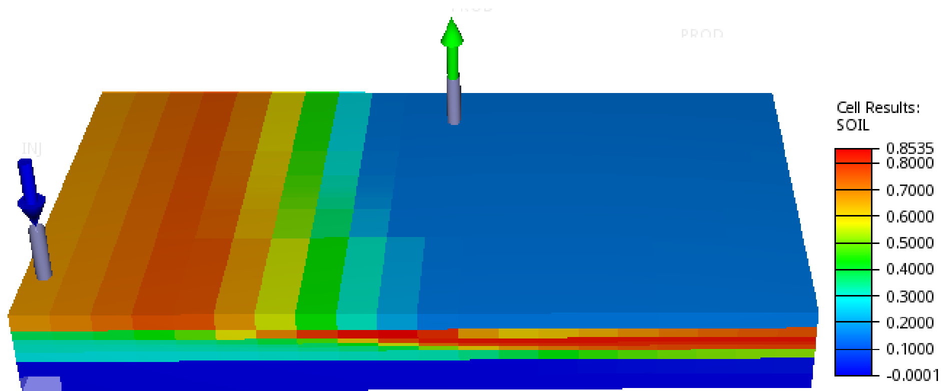
(a) Cumulative oil production



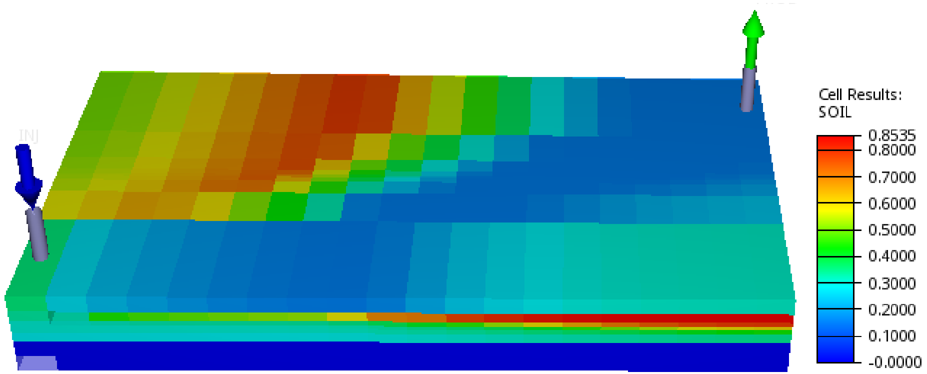
(b) Cumulative water production

Figure 4.18: Cumulative oil and water production of the three-dimensional model

It is evident that our optimized case has a much better sweep efficiency than the initial position. For the initial position, which is in the middle of the reservoir, a significant amount of oil on the side away from injector still remains in the reservoir. Even on the same side of the injector, a lot of oil is still left in the reservoir because of the early water breakthrough. Whereas such situation improved a lot in our optimized position, which is shown in the figure.



(a) Oil distribution of the base case



(b) Oil distribution of case 14

Figure 4.19: Oil distribution of the three-dimensional model (first layer excluded)

Chapter 5

Application of the Method for OLYMPUS Reservoir Model

In this chapter, we will make use of the knowledge we gained from the previous chapter and perform the well placement optimization on OLYMPUS model, a complicated reservoir model. The reservoir model used is described in the next section. Limited by the time available and the performance of our computer, we will optimize the location of one well out of the sixteen wells. Our aim is to improve the position of a producer by using our algorithm.

5.1 Description of OLYMPUS model for optimization challenge

OLYMPUS is a synthetic reservoir model inspired by a virgin oil field in the North Sea, developed for the purpose of a benchmark study for field development optimization (R.M. Fonseca and Leeuwenburgh, 2017). The field is $9km \times 3km$ wide and is bounded on one side by a boundary fault. The reservoir model is shown in figure 5.1. It is $50m$ thick for which 16 layers have been modeled. In addition to the boundary fault, six minor faults are present in the reservoir. The reservoir consists of two zones, separated by an impermeable shale layer. The top reservoir zone contains fluvial channel sands embedded in floodplain shales. The bottom reservoir zone consists of alternating layers of coarse, medium and fine sands with a predetermined dip similar to a clinoformal stratigraphic sequence (R.M. Fonseca and Leeuwenburgh, 2017).

For the model, it consists of grid cells of approximately $50m \times 50m \times 3m$ each. 16 wells are drilled in the given model as shown in figure 5.1, 10 of which are producers and 6 of which are water injectors.

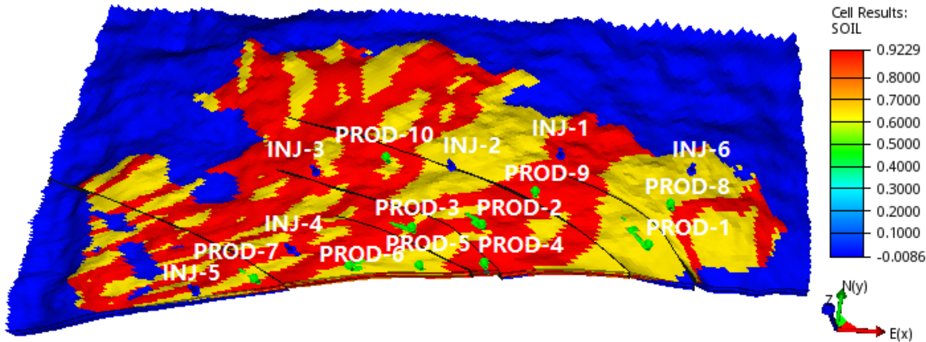


Figure 5.1: OLYMPUS reservoir model

5.1.1 The optimization challenge of the model

The following geological reservoir characteristics are identified as sources of complexity in OLYMPUS model for a good field development optimization challenge:

Faulting The presence of faults makes regular well patterns suboptimal such that placement of individual wells needs to be optimized. Some compartmentalization would be preferred, but compartments should not be regular in shape, again to prevent the optimality of regular well patterns (Geosciences, 2017).

Barriers The No.8 layer of OLYMPUS model is a completely sealing horizontal barrier which separates the whole reservoir into two. The existence of the sealing barrier will make it more difficult to best develop two separate reservoirs with a limited set of wells.

Channels The presence of channels will introduce the challenge of high-connectivity and undesired fast water breakthrough. In the OLYMPUS model, there exists multiple channels that can be found on figure 5.2. Note that the permeability values in the X and Y directions are identical.The permeability in the Z direction is 10% of the permeability in the X direction.

5.2 Optimization problem description

For the wells in the model, all the producers are controlled by a constant bottom hole pressure at $175bar$, and all the injectors are controlled by a constant bottom hole pressure at $235bar$. The production time is $20years$, along which all producers and injectors are open. For the completion of wells, most producers are horizontal wells while all injectors are vertical wells. The oil saturation distributions at the end of the production time are given in figure 5.3. We observe that PROD-10 is a vertical well. (The completion of PROD-10 is listed in appendix F.) In addition, there is still a large amount of remaining

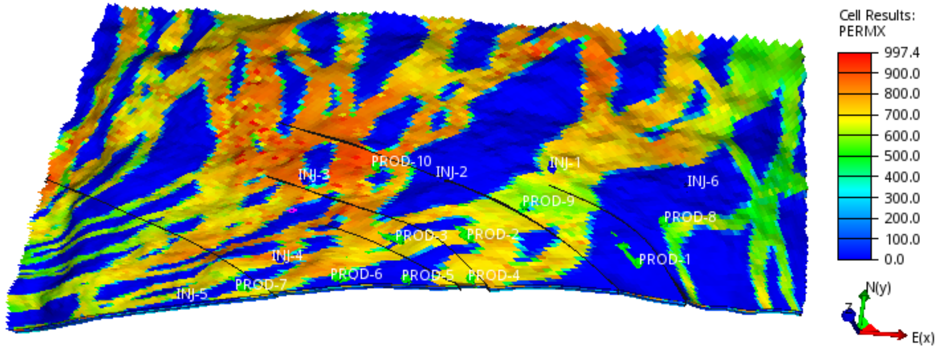


Figure 5.2: The challenge of the channels

oil around the well at the end of production. Therefore, we believe that by optimizing the well placement of PROD-10 to make it become a horizontal well, PROD-10 has a great potential to produce much more oil than before.

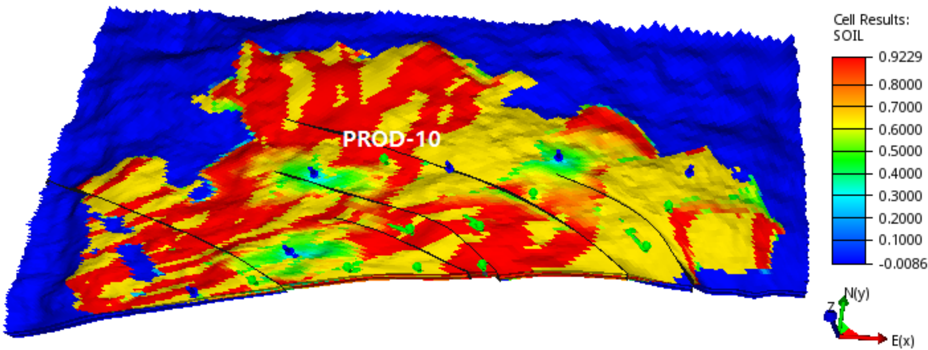


Figure 5.3: Oil saturation distribution for the initial case

Though other producers also have potential, due to the limited time available and computing resources, the scope of our work in this thesis focuses only on PROD-10.

5.3 Parameters for GSS algorithm

5.3.1 Linear constraints

The reservoir model is so huge for placing one well. Given an initial distribution of wells in the reservoir, our work direction is to split the entire reservoir into small segments around each well, so that the well can be optimized in the corresponding segment. In this

way, the overall well distribution will not change significantly. The original well pattern still remains to some extent after optimization. In doing so, illogical well placements can be avoided, such as two wells intersecting, or wells that cross the faults, etc. The linear constraints we set for PROD-10 is shown in figure 5.4. Our strategy is as follows.

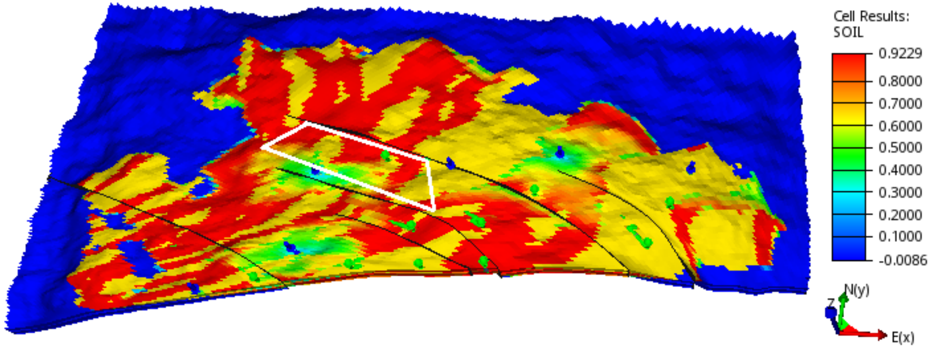


Figure 5.4: Linear constraints

As shown in figure 5.3, two faults and two injectors are near PROD-10. We do not want PROD-10 to have any intersection with them. Therefore, we set two nearly parallel linear constraints; one is exactly along the fault, the other one is one grid block away from the injector, to exclude the faults and injectors out.

Furthermore, the west side of PROD-10 is water zone as indicated in blue. The east side of PROD-10 is in the range of another producer: PROD-2. Thus, we pick another two linear constraints to cut our own region out. In this way, we have enclosed our space.

Then, we propose two planes to define the range in the Z-direction. The water-oil contact for OLYMPUS model is about $2090m$, which means oil zone is above it. The oil zone is our target zone for placing PROD-10. We note that the top of the reservoir is not a flat plane. Therefore, we pick an average depth of the top in this region, which is $2050m$. In this way, we have included both two parts of oil zone separated by the barrier. The channel problem is not necessary to be taken into consideration since the feasible space for PROD-10 is already small enough.

In sum, the above are how we define the six linear constraints. (12 linear constraints in total for heel and toe.) The matrix of linear constraints is listed in figure 5.5.

5.3.2 miscellaneous

We know that higher contraction factor and expansion factor can alleviate the problem of getting trapped in a local optimum. However, considering the time spent for running, our contractor factor and expansion factor are 0.5 and 2.5, respectively. The starting point of PROD-10 is the initial position of it. The diagonal of the matrix D and the scaling factor c we use are listed below to make all variables in the same magnitude and start from 0.

$$\begin{bmatrix} 0.27398 & 1 & 0 & 0 & 0 & 0 \\ 2.81057 & 1 & 0 & 0 & 0 & 0 \\ -0.28063 & -1 & 0 & 0 & 0 & 0 \\ -0.58525 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.27398 & 1 & 0 \\ 0 & 0 & 0 & 2.81057 & 1 & 0 \\ 0 & 0 & 0 & -0.28063 & -1 & 0 \\ 0 & 0 & 0 & -0.58525 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ x_2 \\ y_2 \\ z_2 \end{bmatrix} \leq \begin{bmatrix} 6324060 \\ 7651897 \\ -6326966 \\ 5875644 \\ 2090 \\ -2050 \\ 6324060 \\ 7651897 \\ -6326966 \\ 5875644 \\ 2090 \\ -2050 \end{bmatrix}$$

Figure 5.5: Linear constraint matrix for OLYMPUS model

$$D_{diag} = 1, 2.5, 40, 1, 2.5, 40$$

$$c = -521227, -15450000, -82000, -521227, -15450000, -82000$$

In addition to that, we also include a nonlinear constraint handler built in FieldOpt: *WellSpline Length*, which sets the upper limit for the length of the well. In our work, we set it to be $1000m$ because it is not possible to drill a well that is so long. The summary of our configuration is listed in table 5.1.

Table 5.1: Initial configurations for OLYMPUS case

Parameter	Value
Initial step length (m)	500
Minimum step length (m)	20
Bookkeeper tolerance (m)	2
SplinePoint-PROD-10-heel-x (m)	522937
SplinePoint-PROD-10-heel-y (m)	6180617
SplinePoint-PROD-10-heel-z (m)	2062
SplinePoint-PROD-10-toe-x (m)	522965
SplinePoint-PROD-10-toe-y (m)	6180606
SplinePoint-PROD-10-toe-z (m)	2081
Initial objective function value	5659246

5.4 Optimization results

Table 5.2 lists the results of the optimization. The objective function value for base case is 5659246. It is increased by 11% after optimization. We note that the optimization of one

Table 5.2: Optimization results

Parameter	Value
Final objective function value	6287516
Total number of cases	194
Valid cases	186
Bookkeeped cases	0
Optimization duration (hours)	39.3
SplinePoint-PROD-heel-x (m)	522169.2
SplinePoint-PROD-heel-y (m)	6180948.6
SplinePoint-PROD-heel-z (m)	2064.3
SplinePoint-PROD-toe-x (m)	523126.5
SplinePoint-PROD-toe-y (m)	6180659.7
SplinePoint-PROD-toe-z (m)	2066.2

well out of sixteen wells can yield 11% increase in term of the whole field, which is an apparent improvement. There are eight invalid cases during optimization. Those are the cases out of reservoir boundary. Since the top of the reservoir is not a flat plane, some part of the top surface is lower than the plane defined by the upper linear constraint. Therefore, some points that satisfy the linear constraints can still be outside the reservoir, which are treated as invalid points.

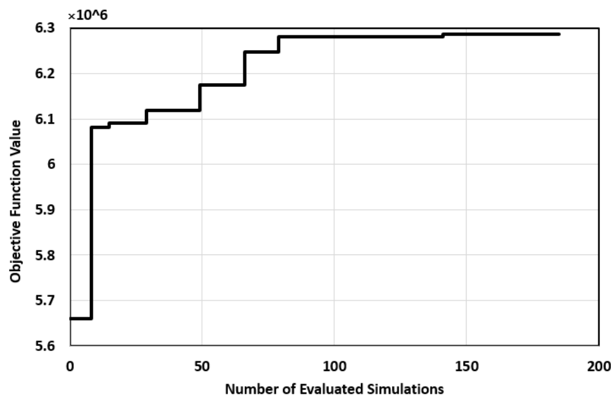


Figure 5.6: Evolution of the objection function value

The optimization process is further illustrated in figure 5.6, where the evolution of the objective function versus the number of evaluated cases is represented. Huge improvements are gained in the early iterations since the well length becomes longer and longer, resulting in more and more oil production. The improvements are gradually smaller in the late iterations since the limit of well length has reached. Only adjusting the location of heel and toe yields small improvements. The optimization converges with an acceptable number of evaluated cases. Therefore, it is reasonable and viable to expect to extend the

optimization to all sixteen wells using our algorithm, with better computing resources and parallel runner.

5.5 Interpretation of the results

The completion of the well after optimization is shown in figure 5.7. The detailed completion is given in appendix G. As shown in figure 5.7, the well is a horizontal well along the strike direction of the fault. Figure 5.8 illustrates oil saturation distribution of the new completion. It is evident that, compared with the previous oil saturation distribution shown in figure 5.3, the oil saturation near the well is significantly reduced, indicated by the big green area in the figure.

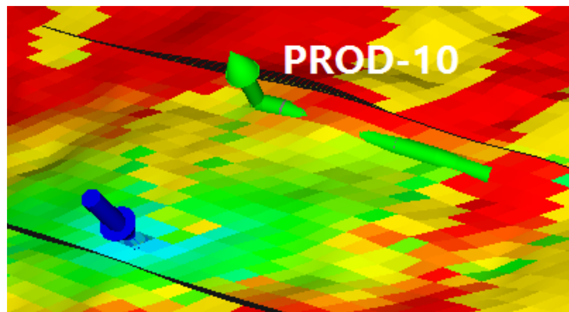


Figure 5.7: Optimal well completion

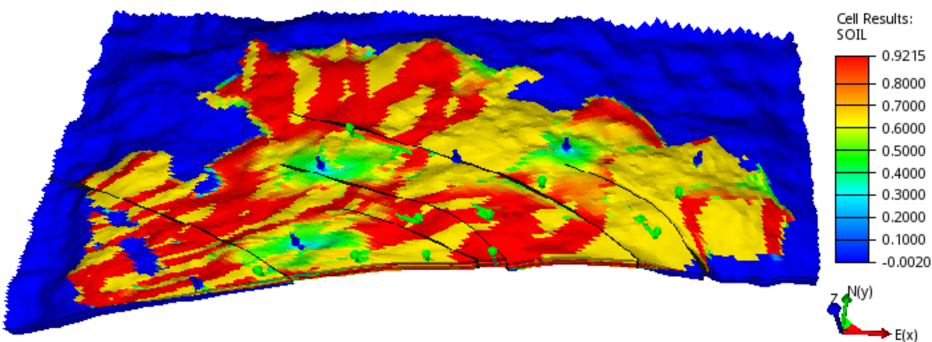


Figure 5.8: Oil saturation distribution after optimization

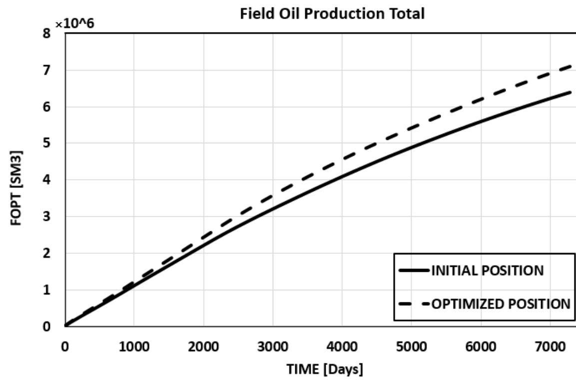


Figure 5.9: Cumulative oil production of the field

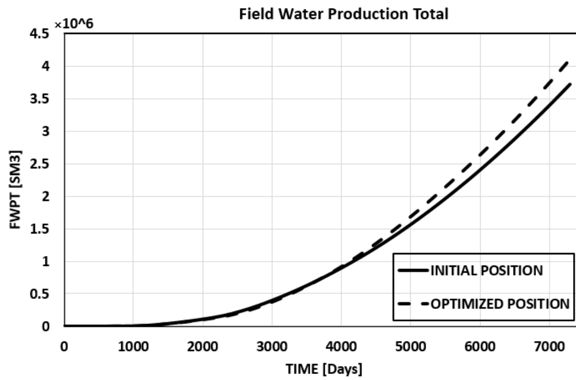


Figure 5.10: Cumulative water production of the field

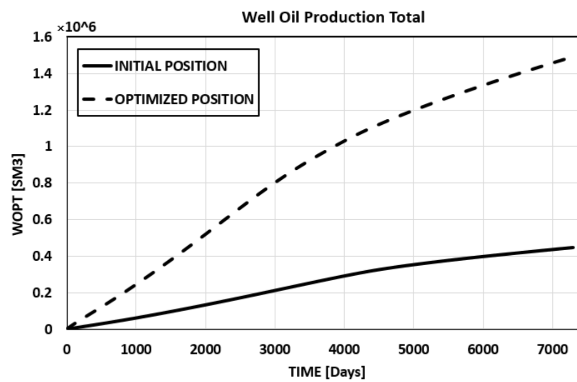


Figure 5.11: Cumulative oil production of PROD-10

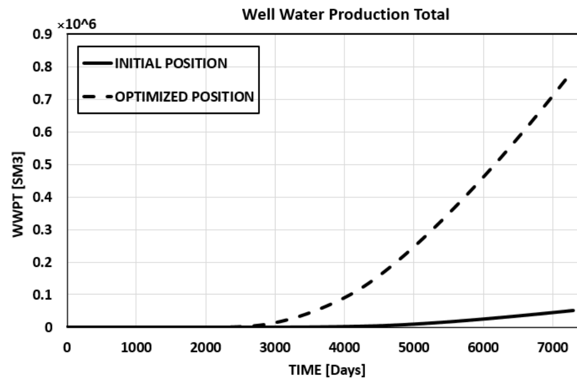


Figure 5.12: Cumulative water production of PROD-10

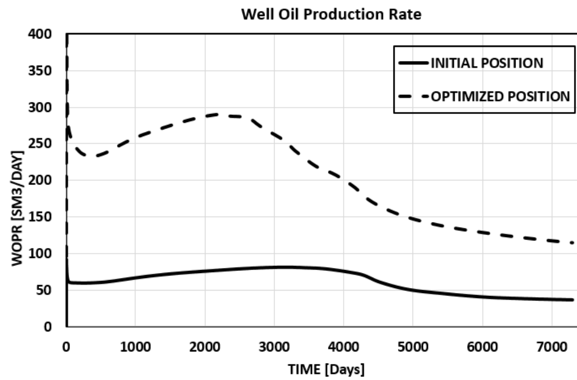


Figure 5.13: Oil production rate of PROD-10

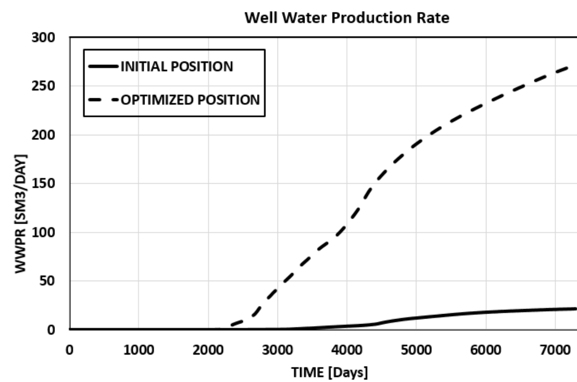


Figure 5.14: Water production rate of PROD-10

Figure 5.9 to figure 5.14 display the production data of OLYMPUS field. For the whole field, the cumulative oil production is increased by 11%. The cumulative water production is also increased by 10%. However, this is more than compensated for by the increase in cumulative oil. The difference is more clear in figure 5.11 and figure 5.12, where oil production for PROD-10 is 3.3 times as much as the initial case, and water production is even 15.5 times of the initial one. We note that, for the initial case, there is almost no water production at all. It is because the production rate for the initial case is so low that water breakthrough occurs very late. Even so, such case is not preferable since oil production is also very low, leading to a low objective function value. For the oil production rate in figure 5.13, at first, it goes up slowly for both initial and optimized cases. This part represents the production rate before water breakthrough. As water is displacing oil, since the viscosity of water is much less than oil, and both producers and injectors are controlled by the constant bottom hole pressure, the production rate will gradually increase according to Darcy's law. Once water breakthrough occurs, oil production rate will decline drastically. Our result shows that at the end of the production time frame, the water cut has reached 70%, which is 37% in the base case.

Chapter 6

Summary and Recommendations

6.1 Summary

This thesis has dealt with solving the well placement optimization problem using GSS algorithm with linear constraints. The main goal of our work is to provide new ideas for such problems. Our work can be summarized as:

- Literature review has shown that few researchers have focused on GSS method to solve well placement optimization problem since it is a local optimization method. Even fewer researchers have approximated the feasible domain with piecewise linear constraints.

- According to the test cases we made, GSS method can easily get trapped into a local optimum for well placement problem. However, configuring proper parameters and making use of linear constraints can alleviate such problem.

- With higher contraction factor and expansion factor, our algorithm can find better solution. The reason is that higher contraction and expansion factor enable the algorithm to search more trial points before termination.

- Compass pattern has a better performance than Fast pattern since it is so easy for the Fast pattern to trap into a local optimum. Three search directions of the Fast pattern are not representative enough for the conditions around a point.

- By introducing linear constraints to divide the feasible region out, the total number of iterations can be reduced significantly with the same, or even better solution.

- We have applied our algorithm to OLYMPUS model, and the solution increases the oil production significantly, which indicates a good performance and applicability of our algorithm.

6.2 Recommendations for further work

The methodology introduced in this work is quite general. Thus there are still issues that should be investigated further. Here we recommended the following areas for future research:

6.2.1 New way to define the location of a well

In this thesis, we define one well by six real values, which are the x, y, z coordinates of the heel and toe. However, there is a disadvantage of such definition. The shape of a typical reservoir is always not flat in Z-direction as illustrated in figure 6.1, where the planes indicated by the black lines are the linear constraints, and the surfaces indicated by the blue lines are the real reservoir layers we intend to place the well. It is evident that the area covered by the two linear constraints vary a lot with the actual region.

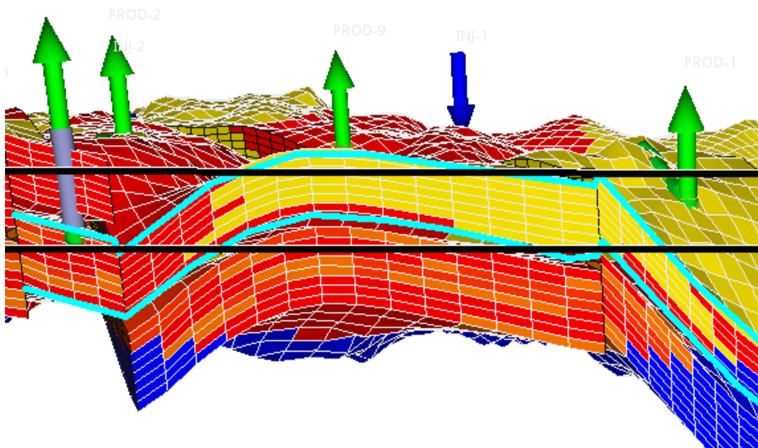


Figure 6.1: The disadvantage of using linear constraints in Z-direction

One way to overcome it is to divide the area more finely so that we can better approximate it by linear constraints. However, it will definitely make the process more complicated. One better way we propose is to change the way how we define a well. Instead of defining a well by six real values, it is better to define a well by four real values, which denote the x and y coordinates, and two integers, which refers to the index of reservoir layer (index of Z). In this way, the well can always be placed into the reservoir layer we want, no matter how ups and downs the reservoir is.

However, this requires a large amount of modification to the FieldOpt platform, which is out of our scope. This is why it is not implemented in our work.

6.2.2 More advanced degenerate case handler

In our work, we use a simplified degenerate case handler, which it to contract the step length when the degenerate case occurs. Although it works well according to our test,

we still expect a more advanced degenerate case handler which has a strong mathematical foundation.

As mentioned in chapter 2, a C-library package named *cddlib* package was used to solve the degenerate case in the work of Lewis et al. (2007b). We recommend to integrate such package into our FieldOpt platform. Again, this requires a strong mathematical and programming background to implement.

6.2.3 More algorithms to compare

Currently, our FieldOpt is still lacking other optimization algorithms which are commonly used for solving the well placement optimization problem, though some researchers are working on it. In this thesis, we have gained seemingly good results from our algorithm. Thus we expect to compare our algorithm with other algorithms for the same problem, especially with those commonly-used algorithms.

6.2.4 More search directions

As mentioned in chapter 2, Without nearby linear constraints, the searching process can be only in the four directions along the coordinate axis with compass pattern, or three directions with the fast pattern. The search along such fixed directions can easily lead to trapping in a local optimum. In the work of Bellout et al. (2012), they make the stencil orientation randomly modified after each polling. Therefore, we recommend adding the same feature to our algorithm, such as adding the same random search directions to $\mathcal{H}_k$ in unsuccessful iteration in algorithm 1, chapter 2. We believe that it can reduce the possibility of getting trapped into the local optimum.

6.2.5 Parallel Computing

Our GSS algorithm is easily parallelized in theory which means that we can speed up the optimization by evaluating several coordinates simultaneously. Still, in practice, it needs some efforts to parallelize the algorithm.

Bibliography

Artus, V., Durlofsky, L. J., Onwunalu, J., Aziz, K., 2006. Optimization of nonconventional wells under uncertainty using statistical proxies. Computational Geosciences 10 (4), 389–404.

Badru, O., Kabir, C., 2003. Well placement optimization in field development. In: SPE Annual Technical Conference and Exhibition, 5-8 October, Denver, Colorado. Society of Petroleum Engineers.

Bangerth, W., Klie, H., Wheeler, M. F., Stoffa, P. L., Sen, M. K., 2006. On optimization algorithms for the reservoir oil well placement problem. Computational Geosciences 10 (3), 303–319.

Baumann, E. J. M., 2015. Fieldopt: Enhanced software framework for petroleum field optimization - development of software support system for the integration of oil production problems with optimization methodology. Master's thesis, NTNU.

Beckner, B., Song, X., 1995. Field development planning using simulated annealing - optimal economic well scheduling and placement. In: SPE Annual Technical Conference and Exhibition, 22-25 October, Dallas, Texas. Society of Petroleum Engineers.

Bellout, M. C., Echeverra Ciaurri, D., Durlofsky, L. J., Foss, B., Kleppe, J., 09 2012. Joint optimization of oil well placement and controls. Computational Geosciences 16 (4), 1061–1079.

Bouzarkouna, Z., Ding, D. Y., Auger, A., 2010. Using evolution strategy with meta-models for well placement optimization. In: ECMOR XII-12th European Conference on the Mathematics of Oil Recovery.

Bouzarkouna, Z., Ding, D. Y., Auger, A., 2012. Well placement optimization with the covariance matrix adaptation evolution strategy and meta-models. Computational Geosciences 16 (1), 75–92.

Geosciences, T. A., 2017. Isapp - integrated systems approach for petroleum. URL `http://www.isapp2.com/optimization-challenge/reservoir-model-description`

Griffin, J. D., Kolda, T. G., Lewis, R. M., 2008. Asynchronous parallel generating set search for linearly constrained optimization. SIAM Journal on Scientific Computing 30 (4), 1892–1924.

Isebor, O. J., 2013. Derivative-free optimization for generalized oil field development. Ph.D. thesis, Stanford University.

Jesmani, M., Bellout, M. C., Hanea, R., Foss, B., 2016. Well placement optimization subject to realistic field development constraints. Computational Geosciences 20 (6), 1185–1209.

Jesmani Mansoureh, Bellout Mathias C., R. H., Foss, B., 2015. Particle swarm optimization algorithm for optimum well placement subject to realistic field development constraints. In: SPE Reservoir Characterisation and Simulation Conference and Exhibition, 14-16 September, Abu Dhabi, UAE. Society of Petroleum Engineers.

Kolda, T. G., Lewis, R. M., Torczon, V., 2003. Optimization by direct search: New perspectives on some classical and modern methods. SIAM Review 45 (3), 385–482.

Kolda, T. G., Lewis, R. M., Torczon, V., 2007. Stationarity results for generating set search for linearly constrained optimization. SIAM Journal on Optimization 17 (4), 943–968.

Lewis, R. M., Shepherd, A., Torczon, V., 2007a. Implementing generating set search methods for linearly constrained minimization. SIAM Journal on Scientific Computing 29 (6), 2507–2530.

Lewis, R. M., Shepherd, A., Torczon, V., 2007b. Implementing generating set search methods for linearly constrained minimization. SIAM Journal on Scientific Computing 29 (6), 2507–2530.

Magnusson, H., 2016. Development of constraint handling techniques for well placement optimization in petroleum field development. Master's thesis, NTNU.

Onwunalu, J., 2010. Optimization of field development using particle swarm optimization and new well pattern descriptions. Ph.D. thesis, Stanford University.

Onwunalu, J. E., Durlofsky, L. J., 2010a. Application of a particle swarm optimization algorithm for determining optimum well location and type. Computational Geosciences 14 (1), 183–198.

Onwunalu, J. E., Durlofsky, L. J., 2010b. Application of a particle swarm optimization algorithm for determining optimum well location and type. Computational Geosciences 14 (1), 183–198.

R.M. Fonseca, C. G., Leeuwenburgh, O., March 2017. Description of olympus reservoir model for optimization challenge. Tech. rep., TNO, Delft University of Technology, ENI, Statoil and Petrobras.

Schlumberger, 2014.
URL `https://www.software.slb.com/products/eclipse`

T.D. Humphries, R. H., 2015. Joint optimization of well placement and control for non-conventional well types. Journal of Petroleum Science and Engineering 126, 242 – 253.

Umut Ozdogan, Akshay Sahni, B. Y. B. G. W. H. C., 2005. Efficient assessment and optimization of a deepwater asset using fixed pattern approach. In: SPE Annual Technical Conference and Exhibition, 9-12 October, Dallas, Texas. Society of Petroleum Engineers.

Appendix A

How FieldOpt Works

FieldOpt is a software framework developed by Baumann (2015) that aims at being a common platform for MSc and Ph.D. students to conduct research. Generally speaking, it takes the input containing all the configurations of the run and gives the output of the optimized well placement and other result files. FieldOpt itself comprises more than sixteen thousand lines of C++ source code, and currently, the codes are still being improved. It contains several different modules. While implementing optimization algorithm in FieldOpt, the understanding of the necessary modules is needed in order to make changes to them and create new optimization method class. The other modules which do not influence the optimization method can be treated as "black box" which will never be touched. This section will introduce the current status and workflow of FieldOpt. The aim of it is to get a brief understanding of how FieldOpt works as a whole.

A.1 Driver Files

FieldOpt takes the JSON (JavaScript Object Notation) file as the driver file, which is the only input file of the FieldOpt. JSON is a lightweight data-interchange format. On the one hand, it is easy for humans to read and write. On the other hand, it is easy for machines to parse and generate. It contains all the configurations needed by FieldOpt in the form of sub-objects. The sub-objects are Global, Optimizer, Simulator, Model. Note that all the sub-objects must be present. An example JSON file can be found at appendix B.

A.2 Optimizer

Currently, two optimization methods have been implemented in FieldOpt, which are Compass Search method (CS) and Asynchronous Parallel Pattern Search method (APPS). And the integration of other optimization algorithms is currently in progress. The different algorithm has its different settings and parameters. The needed parameters have to be provided in the driver file corresponding to the selected optimization algorithm.

A.3 Simulator

FieldOpt supports three type of simulators which are ECLIPSE, ADGPRS, and FLOW. In this work, only ECLIPSE (Schlumberger, 2014) is used.

A.4 Model

When the simulator is ECLIPSE, FieldOpt will parse the reservoir grid through the EGRID file. The path of the EGRID file should be provided so that FieldOpt can read and then generate wells in that reservoir grid model. Currently, all the wells can either be defined by *WellSpline* or by a set of *WellBlocks*. *WellSpline* means that a well is defined by its position of heel and toe, which is an array of float coordinates. The well trajectory is then the connection between the two points. In this way, for a three-dimensional model, one well is defined by six variables for the simulator to handle. *WellBlocks*, in another way, defines one well directly by the perforation of the reservoir blocks. In this work, only the way of *WellSpline* is used and discussed because it is more realistic and practical.

A.5 Objective Function

Currently, FieldOpt only supports one type of objective function to be used by the optimization algorithm, which is *WeightedSum*. Usually, it is set to be the linear combination of the different results of eclipse running. In this work, the Objective Function is always set to be:

$$FOPT - 0.2 \times FWPT$$

where FOPT denotes cumulative oil production, and FWPT refers to cumulative water production.

A.6 Constraints

FieldOpt also supports some different kinds of constraint handling, which includes *Reservoir Boundary, Well Spline Length, Well Spline Interwell Distance*. If such constraints are listed in the driver file, after the new cases are generated by the optimizer, the constraint handling module will take over all the new cases and snap these new cases to constraints. The constraint handling module was implemented by Magnusson (2016). *Reservoir Boundary* just defines the feasible region of a reservoir to be a cubic model. Once the new case is outside the feasible region, it will be projected onto the boundary of the feasible region. *Well Spline Length* defines the upper and lower limit of the well length, in case of too long and too short well length which makes no sense to drilling engineering. *Well Spline Interwell Distance* defines the limit of interwell-distance, since it also makes no sense if the two different wells intersect or they are too close to each other. The detailed derivation and implementation of all these constraints can be found at Magnusson (2016). Note that all these constraint handlers are different with the linear constraint handler implemented in this paper, since the linear constraint handler in our work only applies to our

GSS method. In other words, it is packed together with our GSS algorithm. However, the global constraint handler discussed above can still take over the new cases generated by our algorithm to make changes.

A.7 Bookkeeper

The BookkeeperTolerance sets the tolerance for the case bookkeeper. That is, when the new point generated by the optimizer is close enough to one of the evaluated points, FieldOpt will directly mark that point as evaluated without running the simulator and set the objective function value to be the same as the point which meets the tolerance.

A.8 The main Loop of Serial Runner

FieldOpt supports more than one kind of runner, which include serial runner, mpi runner, synchronous mpi runner and so on. In this work, we use only serial runner since it is easy to use and understand. The main loop in the serial runner is shown on algorithm 5, coded by Baumann (2015), which indicates the working flow of FieldOpt.

```
initialization;
while Optimizer is not finished do
  | Get new case for evaluation;
  | if Bookkeepered then
  |   | Mark the case as bookkeeped;
  | else
  |   | Apply the case to model;
  |   | Run simulator;
  |   | if Simulation succeeds then
  |   |   | Set objective function value to the case;
  |   | else
  |   |   | Set a sentinel value to the case;
  |   | end
  | end
  | Update tentative best case;
  | Log the case;
end
```

Algorithm 5: Serial runner

Before FieldOpt enters this loop, it will firstly parse the driver file to initialize Settings, Model, Simulator, optimizer and other objects. This loop means the start of optimization. FieldOpt will never jump out of the loop until the termination condition is reached. As shown in the algorithm, first, FieldOpt will fetch a new case from the optimizer. How the new case is generated depends on the optimizer. After that, the Bookkeeper will take over this case to see if this case is evaluated or not. If not, FieldOt will apply this new case to the reservoir model to generate the runnable DATA input file for ECLIPSE and then run the simulator. FieldOpt will wait until ECLIPSE finishes the simulation. Then it will read

the summary file of ECLIPSE and calculate the objective function value. If the simulation cannot run because the new point is outside the reservoir, the objective function value of this case will be set to a sentinel value which is a pretty small value. The logger in the loop is responsible for recording the running information and outputs all such information to the output directory. Until now the first round of the loop is over. Then FieldOpt will run into the next round of the loop and fetch another case to evaluate. This is the workflow of FieldOpt in this work.

Appendix B

An Example of the Driver File

```
{
    "Global": {
        "Name": "5spot",
        "BookkeeperTolerance": 5.0
    },
    "Optimizer": {
        "Type": "GSS_Linear_Constraints",
        "Mode": "Maximize",
        "Parameters": {
            "MaxEvaluations": 3200,
            "InitialStepLength": 300.0,
            "MinimumStepLength": 10.0,
            "ContractionFactor": 0.5,
            "ExpansionFactor": 2,
            "ActualDimensionsForEachPoint": 2,
            "WellType": "Vertical",
            "ConstraintFlag": true,
            "LinearConstraints": "0.4831 1 0 0 3.0455 -1 0 0 0 -1 0 0 -3.4444 -1 0 0 0 0 0.4831 1 0 0 3.0455 -1 0 0 0 -1 0 0 -3.4444 -1",
            "Bounding": "1366 3645 -10 -2077 1366 3645 -10 -2077",
            "SearchPatternWhenNoConstraintNearby": "compass",
            "ScalingFactorD": "1 1 1 1",
            "ScalingFactorC": "0 0 0 0"
        },
        "Objective": {
            "Type": "WeightedSum",
            "WeightedSumComponents": [
                {
                    "Coefficient": 1.0, "Property": "CumulativeOilProduction", "TimeStep": -1,
                    "IsWellProp": false
                },
                {
                    "Coefficient": -0.2, "Property": "CumulativeWaterProduction", "TimeStep": -1,
                    "IsWellProp": false
                }
```

```
35                  ]
            },
37          "Constraints": [
            ]
39      },
        "Simulator": {
41          "Type": "ECLIPSE",
            "FluidModel": "DeadOil",
43          "ExecutionScript": "csh_eclrun"
        },
45      "Model": {
            "ControlTimes": [0, 300, 600, 900, 1200, 1500],
47          "Reservoir": {
                "Type": "ECLIPSE"
49          },
            "Wells": [
51              {
                    "Name": "PROD1",
53                  "Group": "G2",
                    "Type": "Producer",
55                  "DefinitionType": "WellSpline",
                    "PreferredPhase": "Oil",
57                  "WellboreRadius": 0.1905,
                    "SplinePoints": {
59                      "Heel": {
                            "x": 400.0,
61                          "y": 1000.0,
                            "z": 1712.0,
63                          "IsVariable": true
                        },
65                      "Toe": {
                            "x": 400.0,
67                          "y": 1000.0,
                            "z": 1712.0,
69                          "IsVariable": true
                        }
71                  },
                    "Controls": [
73                      {
                            "TimeStep": 0,
75                          "State": "Open",
                            "Mode": "BHP",
77                          "BHP": 80.0
                        }
79                  ]
                },
81              {
                    "Name": "PROD2",
83                  "Group": "G2",
                    "Type": "Producer",
85                  "DefinitionType": "WellSpline",
                    "PreferredPhase": "Oil",
87                  "WellboreRadius": 0.1905,
                    "SplinePoints": {
89                      "Heel": {
                            "x": 500.0,
91                          "y": 900.0,
```

```
                        "z": 1712.0,
                        "IsVariable": true
                    },
                    "Toe": {
                        "x": 500.0,
                        "y": 900.0,
                        "z": 1712.0,
                        "IsVariable": true
                    }
                },
                "Controls": [
                    {
                        "TimeStep": 0,
                        "State": "Open",
                        "Mode": "BHP",
                        "BHP": 100.0
                    }
                ]
            },
            {
                "Name": "INJ1",
                "Group": "G1",
                "Type": "Injector",
                "DefinitionType": "WellSpline",
                "PreferredPhase": "Water",
                "WellboreRadius": 0.1905,
                "SplinePoints": {
                    "Heel": {
                        "x": 108.0,
                        "y": 108.0,
                        "z": 1712.0,
                        "IsVariable": false
                    },
                    "Toe": {
                        "x": 108.0,
                        "y": 108.0,
                        "z": 1712.0,
                        "IsVariable": false
                    }
                },
                "Controls": [
                    {
                        "TimeStep": 0,
                        "Type": "Water",
                        "State": "Open",
                        "Mode": "BHP",
                        "BHP": 500.0,
                        "IsVariable": false
                    }
                ]
            },
            {
                "Name": "INJ2",
                "Group": "G1",
                "Type": "Injector",
                "DefinitionType": "WellSpline",
                "PreferredPhase": "Water",
```

```
                    "WellboreRadius": 0.1905,
                    "SplinePoints": {
                        "Heel": {
                            "x": 1332.0,
                            "y": 1332.0,
                            "z": 1712.0,
                            "IsVariable": false
                        },
                        "Toe": {
                            "x": 1332.0,
                            "y": 1332.0,
                            "z": 1712.0,
                            "IsVariable": false
                        }
                    },
                    "Controls": [
                        {
                            "TimeStep": 0,
                            "Type": "Water",
                            "State": "Open",
                            "Mode": "BHP",
                            "BHP": 250.0,
                            "IsVariable": false
                        }
                    ]
                }
            ]
    }
}
```

Example_driver_file.json

Appendix C

GSS_Linear_Constraints.h

```
//
// Created by bo on 2/19/17.
//

#ifndef FIELDOPT_GSS_LINEAR_CONSTRAINTS_H
#define FIELDOPT_GSS_LINEAR_CONSTRAINTS_H

#include "GSS.h"
#include "Optimization/optimizer.h"
#include <Eigen/Core>
#include <vector>

using namespace Eigen;
using namespace std;

namespace Optimization {
    namespace Optimizers {

        class GSS_Linear_Constraints : public GSS
        {

        public:
            QString GetStatusStringHeader() const;
            QString GetStatusString() const;

            GSS_Linear_Constraints(Settings::Optimizer *settings,
                                   Case *base_case,
                                   Model::Properties::
    VariablePropertyContainer *variables,
                                   Reservoir::Grid::Grid *grid,
                                   Logger *logger);

        protected:
            int actual_dimensions_for_each_point;
            MatrixXd linear_constraints_;
            VectorXd bounding_;
            vector<int> unsatisfied_constraints_index_;
```

```
        vector<VectorXd> pattern_;
        vector<VectorXd> initial_pattern_;
        Settings::Optimizer::WellType welltype_; //flag for vert or horz
        double step_length_of_pattern_;
        vector<QUuid> spline_point_id_map_;
        //contain the QUuid of all spline point variables. Each well has 6 variables which are
        // heel.x, heel.y, heel.z, toe.x, toe.y, toe.z
        vector<QUuid> working_points_;
        //contain the QUuid of the actual working spline point variables from mathmetical point of view.
        //for example
        //if json file define a vertical well with two dimensions for the point,
        //spline_point_id_map_ will contain the same 6 QUuid of the 6 variables
        //working_points_ will contain only 2 QUuid of 2 variables, which are heel.x and heel.y
        int num_of_well_;
        bool constraint_flag_; // true means we have constraints, false means we don't have constraints.
        vector<double> scaling_factor_D;
        vector<double> scaling_factor_C;
        bool satisfy_constraints();
        void change_pattern_conform_to_constraints();
        bool check_linear_independence();
        void handleEvaluatedCase(Case *c) override;
        QList<Case *> generate_trial_points();
        void contract();
        void expand();
        bool is_converged();
        TerminationCondition IsFinished();
        void select_pattern(Settings::Optimizer *);
        void transfer_value_to_linear_constraints(QString);
        /*
         * When fieldopt read the linear constraints part from json file, it will save that part
         * to QString. This function is to convert the QString to a Matrixxd.
         */
        void transfer_value_to_bounding(QString);
        /*
         * When fieldopt read the bounding part from json file, it will save that part
         * to QString. This function is to convert the QString to a Vectorxd.
         */
        void transfer_value_to_scaling_factor_D(QString);

        void transfer_value_to_scaling_factor_C(QString);

        void set_spline_point_id_map_(Model::Properties::VariablePropertyContainer *);
        /*
         * This function is used to set the value for spline_point_id_map_.
```

```
81           * spline_point_id_map_ contains the ID of spline point type
    variables in order.
             * that is:
83           * the spline_point_id_map_ contains the ID in the order of
             *
85           * well1.heel.x  well1.heel.y  well1.heel.z
             * well1.toe.x  well1.toe.y  well1.toe.z
87           * well2.heel.x  well2.heel.y  well2.heel.z
             * well2.toe.x  well2.toe.y  well2.toe.z
89           * well3......
             * etc......
91           *
             */
93
            void get_working_points(Model::Properties::
    VariablePropertyContainer *);
95          //set the value for working_points_.
            double scale_variable(double, double, double);
97          double revert_scaled_variable(double, double, double);
        private:
99          void iterate(); //!< Step or contract, perturb, and clear list
     of recently evaluated cases.
            bool is_successful_iteration(); //!< Check if this iteration
    was successful (i.e. if the current tent. best case was found in this
    iteration).
101     };
    }
103 }

105 #endif //FIELDOPT_GSS_LINEAR_CONSTRAINTS_H
```

GSS_Linear_Constraints.h

Appendix D

GSS_Linear_Constraints.cpp

```
//
// Created by bo on 2/19/17.
//

#include "GSS_Linear_Constraints.h"
#include <iostream>
#include "gss_patterns.hpp"


namespace Optimization {
    namespace Optimizers {


        GSS_Linear_Constraints::GSS_Linear_Constraints(Settings::Optimizer
         *settings, Case *base_case,
                                                               Model::Properties::
     VariablePropertyContainer *variables,
                                                               Reservoir::Grid::
     Grid *grid, Logger *logger) : GSS(settings, base_case,

                       variables, grid, logger) {
            constraint_flag_ = settings->parameters().constraint_flag;
            actual_dimensions_for_each_point = settings->parameters().
     actual_dimensions_for_each_point;
            welltype_ = settings->parameters().welltype;
            set_spline_point_id_map_(variables);
            get_working_points(variables);
            select_pattern(settings);
            step_length_of_pattern_= settings->parameters().
     initial_step_length;
            constraint_flag_ = settings->parameters().constraint_flag;
            transfer_value_to_scaling_factor_D(settings->parameters().
     scaling_factor_D);
            transfer_value_to_scaling_factor_C(settings->parameters().
     scaling_factor_C);
            if (constraint_flag_){
```

```
                transfer_value_to_linear_constraints(settings->parameters
().linear_constraints);
                transfer_value_to_bounding(settings->parameters().bounding
);
        }
    }



    QList<Case *> GSS_Linear_Constraints::generate_trial_points() {

        auto trial_points = QList<Case *>();
        pattern_ = initial_pattern_;
        if (!satisfy_constraints()) {

            change_pattern_conform_to_constraints();
        }

        auto rea_base = GetTentativeBestCase()->real_variables();

        for (int i = 0; i < (int)pattern_.size(); i++) {
            auto trial_point = new Case(GetTentativeBestCase());
            if (rea_base.size() > 0) {
                for (int j = 0; j < working_points_.size(); ++j) {

                    double perturbed_variable = scale_variable(
rea_base.value(working_points_[j]), scaling_factor_D[j],
scaling_factor_C[j]) + pattern_[i][j] * step_length_of_pattern_;
                    double reverted_variable = revert_scaled_variable(
perturbed_variable, scaling_factor_D[j], scaling_factor_C[j]);
                    trial_point->set_real_variable_value(
working_points_[j], (int)(reverted_variable/0.1)*0.1);
                    ////round to int
                    if (welltype_ == Settings::Optimizer::WellType::
Vertical){
                        std::vector<QUuid>::iterator position = std::
find(spline_point_id_map_.begin(), spline_point_id_map_.end(),

    working_points_[j]);
                        trial_point->set_real_variable_value(*(
position+3), (int)(reverted_variable/0.1)*0.1);
                        ////round to int
                        //cout << (position+3)->toString().toStdString
() << endl;
                    }
                }

                cout << "the number " << i << " new point is" <<endl;
                for (int k = 0; k < working_points_.size(); ++k) {
                    cout << trial_point->real_variables().value(
working_points_[k]) << "-----" ;
                }
                cout << endl;
            }
            trial_point->set_origin_data(GetTentativeBestCase(), i,
step_length_of_pattern_);
            trial_points.append(trial_point);
```

```
        }
        cout << "in this iteration , " << (int)pattern_.size() << " new
 points has been generated." << endl;

        for (Case *c : trial_points)
            constraint_handler_->SnapCaseToConstraints(c);
        return trial_points;
    }



    bool GSS_Linear_Constraints::satisfy_constraints() {

        if (!constraint_flag_)
            return true;

        unsatisfied_constraints_index_.clear();
        auto rea_b = GetTentativeBestCase()->real_variables();
        VectorXd x;
        x.setZero(working_points_.size());
        int index = 0;
        double distance;
        for (auto id : working_points_) {
            x[index] = scale_variable(rea_b.value(id),
scaling_factor_D[index], scaling_factor_C[index]);
            index++;
        }
        //cout << "x is " << endl << x << endl;

        for (int i = 0; i < bounding_.size() ; ++i) {
            distance = std::abs(linear_constraints_.row(i) * x -
bounding_[i])/ linear_constraints_.row(i).norm();
            cout << "distance is " << endl << distance << endl;
            if (distance < step_length_of_pattern_ &&
                std::find(unsatisfied_constraints_index_.begin(),
unsatisfied_constraints_index_.end(), i)
                == unsatisfied_constraints_index_.end()){
                unsatisfied_constraints_index_.push_back(i);
                std::cout << "unsatisfied occur for constraint\n" <<
linear_constraints_.row(i) << endl;
            }
        }


        if (unsatisfied_constraints_index_.size() == 0){
            return true;
        }
        else{
            return false;
        }
    }



    void GSS_Linear_Constraints::change_pattern_conform_to_constraints
() {

```

```
        pattern_.clear();
125
        if(!check_linear_independence()){//  degenerate case. try to
    contract step length to avoid.
127
            cout << "degenerate case occur. try to avoid it by
    contracting step length." << endl;
129
            do{
131             unsatisfied_constraints_index_.clear();
                step_length_of_pattern_ = step_length_of_pattern_ *
    contr_fac_;
133             satisfy_constraints();

135         } while(!check_linear_independence());

137     }

139     MatrixXd p(unsatisfied_constraints_index_.size(),
    linear_constraints_.cols());
        p.setZero();
141     int index = 0;
        for (int j : unsatisfied_constraints_index_)
143     {
            p.row(index) = linear_constraints_.row(j);
145         index++;
        }
147     MatrixXd tran = p.transpose();
        MatrixXd para1 = p*p.transpose();
149     MatrixXd para2 = para1.inverse();
        MatrixXd right_inverse = tran * para2;
151     std::cout << "the right inverse r is \n" << right_inverse <<
    endl;
        for (int i = 0; i < right_inverse.cols() ; ++i) {
153         right_inverse.col(i).normalize();
            std::cout << "the normalized right inverse r" << i << " is
    \n" << right_inverse.col(i) << endl;
155     }

157     FullPivLU<MatrixXd> lu_decomp(p);
        MatrixXd nullspace = lu_decomp.kernel();
159     std::cout << "Here is a matrix whose columns form a basis of
    the null-space of p:\n"
                  << lu_decomp.kernel() << endl;
161

163     for (int i = 0; i < right_inverse.cols(); i++){
            std::cout << "new direction added due to right inverse\n"
    << -right_inverse.col(i).transpose() << endl;
165         pattern_.push_back(-right_inverse.col(i).transpose());
        }
167


169
        if (!nullspace.isZero()){
171
```

```
            MatrixXd positive_spanning_set(nullspace.rows(), nullspace.cols()*2);
            positive_spanning_set.setZero();
            for (int i=0; i<nullspace.cols(); i++){
                positive_spanning_set.col(i) = nullspace.col(i);
            }
            for (int i=(int)nullspace.cols(); i<nullspace.cols()*2; i++){
                positive_spanning_set.col(i) = -nullspace.col(i-nullspace.cols());
            }
            std::cout << "Here is a matrix N whose columns are a positive spanning set for the nullspace of p:\n"
                      << positive_spanning_set << endl;
            for (int i = 0; i < positive_spanning_set.cols() ; ++i) {
                positive_spanning_set.col(i).normalize();
            }
            std::cout << "the normalized matrix N whose columns are a positive spanning set for the nullspace of p is \n" << positive_spanning_set << endl;
            for (int i = 0; i < positive_spanning_set.cols(); i++){
                std::cout << "new direction added due to positive spanning set\n" << positive_spanning_set.col(i).transpose() << endl;
                pattern_.push_back(positive_spanning_set.col(i).transpose());
            }
        }

        unsatisfied_constraints_index_.clear();

    }



    bool GSS_Linear_Constraints::check_linear_independence() {

        MatrixXd unsatisfied_constraints(unsatisfied_constraints_index_.size(), linear_constraints_.cols());
        unsatisfied_constraints.setZero();
        for (int i = 0; i < unsatisfied_constraints_index_.size(); ++i) {
            unsatisfied_constraints.row(i) = linear_constraints_.row(unsatisfied_constraints_index_[i]);
        }
        FullPivLU<MatrixXd> luA(unsatisfied_constraints);
        int rank = (int)luA.rank();

        if (rank == unsatisfied_constraints.rows())
            return true;
        else
            return false;
    }



    void GSS_Linear_Constraints::iterate()
    {
```

```
        if (!is_successful_iteration() && iteration_ != 0)
            contract();
        else
            expand();
        case_handler_->AddNewCases(generate_trial_points());
        case_handler_->ClearRecentlyEvaluatedCases();
        iteration_++;
    }


    QString GSS_Linear_Constraints::GetStatusStringHeader() const
    {
        return QString("%1,%2,%3,%4,%5,%6,%7")
                .arg("Iteration")
                .arg("EvaluatedCases")
                .arg("QueuedCases")
                .arg("RecentlyEvaluatedCases")
                .arg("TentativeBestCaseID")
                .arg("TentativeBestCaseOFValue")
                .arg("StepLength");
    }


    QString GSS_Linear_Constraints::GetStatusString() const
    {
        return QString("%1,%2,%3,%4,%5,%6,%7")
                .arg(iteration_)
                .arg(nr_evaluated_cases())
                .arg(nr_queued_cases())
                .arg(nr_recently_evaluated_cases())
                .arg(GetTentativeBestCase()->id().toString())
                .arg(GetTentativeBestCase()->objective_function_value
())
                .arg(step_length_of_pattern_);
    }


    void GSS_Linear_Constraints::handleEvaluatedCase(Case *c) {
        if (isImprovement(c))
            updateTentativeBestCase(c);
    }


    bool GSS_Linear_Constraints::is_successful_iteration() {
        return case_handler_->RecentlyEvaluatedCases().contains(
GetTentativeBestCase());
    }


    void GSS_Linear_Constraints::contract() {
        step_length_of_pattern_ = step_length_of_pattern_ * contr_fac_
;
```

```
    }

    void GSS_Linear_Constraints::expand() {
        if (iteration_ != 0)
            step_length_of_pattern_ = step_length_of_pattern_ *
expan_fac_;
    }

    bool GSS_Linear_Constraints::is_converged() {

        if (step_length_of_pattern_ >= step_tol_)
            return false;
        else
            return true;
    }

    Optimizer::TerminationCondition GSS_Linear_Constraints::IsFinished
() {
        if (case_handler_->EvaluatedCases().size() >= max_evaluations_
)
            return MAX_EVALS_REACHED;
        else if (is_converged())
            return MINIMUM_STEP_LENGTH_REACHED;
        else return NOT_FINISHED; // The value of not finished is 0,
which evaluates to false.
    }

    void GSS_Linear_Constraints::transfer_value_to_linear_constraints(
QString linear_constraints) {

        char x[1000];
        double a[100];
        std::string stdstring = linear_constraints.toStdString();
        const char * s = stdstring.c_str();
        const char * s1 = stdstring.c_str();
        int i=0;
        while(s-s1<stdstring.length() && sscanf(s, "%s", x)){
            s += strlen(x)+1;
            a[i++] = atof(x);
        }

        linear_constraints_.setZero(i/working_points_.size(),
working_points_.size());
        int l=0;
        //cout << "a[l] is " << endl;
        for (int j = 0; j < i/working_points_.size() ; ++j) {
            for (int k = 0; k < working_points_.size(); ++k) {
                linear_constraints_.row(j)[k] = a[l]/scaling_factor_D[
k];
```

```
                        //cout << "   " << a[l];
323                     l++;
                    }
325                 //cout << endl << "next " << endl;
                }
327             cout << "linear_constraints is " << endl;
                cout << linear_constraints_ << endl;
329         }

331

333         void GSS_Linear_Constraints::transfer_value_to_bounding(QString
        bounding) {

335             char x[1000];
                double a[100];
337             std::string stdstring = bounding.toStdString();

339             const char * s = stdstring.c_str();
                const char * s1 = stdstring.c_str();
341             int i=0;
                while(s-s1<stdstring.length() && sscanf(s, "%s", x)){
343                 s += strlen(x)+1;
                    a[i++] = atof(x);
345             }
                bounding_.setZero(i);
347             for (int j = 0; j < i ; ++j) {
                    bounding_[j] = a[j];
349                 for (int k = 0; k < working_points_.size(); ++k) {
                        bounding_[j] = bounding_[j] + scaling_factor_C[k]*
        linear_constraints_.row(j)[k];
351                 }
                }
353
                std::cout << "bounding is " << endl << bounding_ <<endl;
355         }

357

359         void GSS_Linear_Constraints::select_pattern(Settings::Optimizer *
        settings) {

361             if(settings->parameters().pattern.toStdString() == "fast"){
                    pattern_= GSSPatterns::fast_pattern(
        actual_dimensions_for_each_point, (int)working_points_.size());
363                 initial_pattern_= GSSPatterns::fast_pattern(
        actual_dimensions_for_each_point, (int)working_points_.size());
                }
365             else if(settings->parameters().pattern.toStdString() == "
        compass"){
                    pattern_= GSSPatterns::compass_pattern(
        actual_dimensions_for_each_point, (int)working_points_.size());
367                 initial_pattern_= GSSPatterns::compass_pattern(
        actual_dimensions_for_each_point, (int)working_points_.size());
                }
369         }
```

```
371

373     void GSS_Linear_Constraints::set_spline_point_id_map_(Model::
    Properties::VariablePropertyContainer *variables) {

375         QHash<QUuid, Model::Properties::ContinousProperty *> *
    Continous_Variables_;

377         Continous_Variables_ = variables->GetContinousVariables();
            vector<QString> wellname;
379
            //find well name
381         for (QUuid key : Continous_Variables_->keys()){
                //Continous_Variables_->value(key)->get_parent_well_name()
    ;
383             if (Continous_Variables_->value(key)->isVariable() &&
                    std::find(wellname.begin(), wellname.end(),
    Continous_Variables_->value(key)->propertyInfo().parent_well_name) ==
    wellname.end()){
385                 cout << Continous_Variables_->value(key)->propertyInfo
    ().parent_well_name.toStdString() << endl;
                    wellname.push_back((QString &&) Continous_Variables_->
    value(key)->propertyInfo().parent_well_name);
387             }

389         }
            num_of_well_ = (int)wellname.size();
391         int spline_end[] = { 3001, 3001, 3001, 3002, 3002, 3002 };//
    means {heel, heel, heel, toe, toe, toe}; see property.h
            int coord[] = {4001, 4002, 4003, 4001, 4002, 4003};// means {x
    , y, z, x, y, z}; see property.h
393
            for (int i = 0; i < wellname.size() ; ++i) {
395
                for (int j = 0; j < 6; ++j) {
397
                    for (QUuid key : Continous_Variables_->keys()){
399                     if(Continous_Variables_->value(key)->propertyInfo
    ().parent_well_name == wellname[i] &&
                            Continous_Variables_->value(key)->propertyInfo
    ().spline_end == spline_end[j] &&
401                         Continous_Variables_->value(key)->propertyInfo
    ().coord == coord[j]){

403                          spline_point_id_map_.push_back(key);
                        }
405
                    }
407
                }
409         }
            cout << "spline_point_id_map_ is: " << endl;
411         for (int j = 0; j < spline_point_id_map_.size(); ++j) {
                cout  << spline_point_id_map_[j].toString().toStdString()
   << endl;
413             cout << Continous_Variables_->value(spline_point_id_map_[j
    ])->propertyInfo().parent_well_name.toStdString() << endl
```

```
                    << Continous_Variables_->value(spline_point_id_map_[j
])->propertyInfo().spline_end << endl
415                 << Continous_Variables_->value(spline_point_id_map_[j
])->propertyInfo().coord << endl;
            }
417     }

419


421     void GSS_Linear_Constraints::get_working_points(Model::Properties
::VariablePropertyContainer *variables) {

423         working_points_ = spline_point_id_map_;
            QHash<QUuid, Model::Properties::ContinousProperty *> *
Continous_Variables_;
425         Continous_Variables_ = variables->GetContinousVariables();

427         if (actual_dimensions_for_each_point == 2) {
                for (QUuid key : Continous_Variables_->keys()) {
429                 if (Continous_Variables_->value(key)->propertyInfo().
coord == 4003) {
                        std::vector<QUuid>::iterator position = std::find(
working_points_.begin(), working_points_.end(),
431
key);
                        if (position != working_points_.end()) // ==
myVector.end() means the element was not found
433                         working_points_.erase(position);
                    }
435             }
            }
437
            if (welltype_ == Settings::Optimizer::WellType::Vertical) {
439             for (QUuid key : Continous_Variables_->keys()) {
                    if (Continous_Variables_->value(key)->propertyInfo().
spline_end == 3002) {
441                     std::vector<QUuid>::iterator position = std::find(
working_points_.begin(), working_points_.end(),

key);
443                     if (position != working_points_.end()) // ==
myVector.end() means the element was not found
                            working_points_.erase(position);
445                 }
                }
447         }
            cout << "working point is: " << endl;
449         for (int j = 0; j < working_points_.size(); ++j) {
                cout << working_points_[j].toString().toStdString() <<
endl;
451         }
        }

453


455
        double GSS_Linear_Constraints::scale_variable(double variable,
double d, double c) {
```

```
457
            double scaled_variable;
459         scaled_variable = d * variable + c;
            return scaled_variable;
461     }

463


465     double GSS_Linear_Constraints::revert_scaled_variable(double
    scaled_variable, double d, double c) {

467         double variable;
            variable = (scaled_variable - c) / d;
469         return variable;
        }
471


473
        void GSS_Linear_Constraints::transfer_value_to_scaling_factor_D(
    QString d) {
475
            char x[1000];
477         double a[100];
            std::string stdstring = d.toStdString();
479
            const char * s = stdstring.c_str();
481         const char * s1 = stdstring.c_str();
            int i=0;
483         while(s-s1<stdstring.length() && sscanf(s, "%s", x)){
                s += strlen(x)+1;
485             a[i++] = atof(x);
            }
487         std::cout << "scaling_factor_D is " << endl;
            for (int j = 0; j < i ; ++j) {
489             scaling_factor_D.push_back(a[j]);
                std::cout << "  "<< a[j];
491         }
            cout<< endl;
493     }

495


497     void GSS_Linear_Constraints::transfer_value_to_scaling_factor_C(
    QString c) {
            char x[1000];
499         double a[100];
            std::string stdstring = c.toStdString();
501
            const char * s = stdstring.c_str();
503         const char * s1 = stdstring.c_str();
            int i=0;
505         while(s-s1<stdstring.length() && sscanf(s, "%s", x)){
                s += strlen(x)+1;
507             a[i++] = atof(x);
            }
509         std::cout << "scaling_factor_C is " << endl;
            for (int j = 0; j < i ; ++j) {
```

```
511                     scaling_factor_C.push_back(a[j]);
                        std::cout  << "  "<< a[j];
513                 }
                    cout<< endl;
515
                }
517         }
    }
```

GSS_Linear_Constraints.cpp

Appendix E

gss_patterns.hpp

```
/******************************************************************************
   Created by einar on 11/21/16.
   Copyright (C) 2016 Einar J.M. Baumann <einar.baumann@gmail.com>

   This file is part of the FieldOpt project.

   FieldOpt is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   FieldOpt is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with FieldOpt.  If not, see <http://www.gnu.org/licenses/>.
******************************************************************************/
#ifndef FIELDOPT_GSS_PATTERNS_HPP_H
#define FIELDOPT_GSS_PATTERNS_HPP_H

#include <Eigen/Core>
#include <vector>

namespace Optimization { namespace GSSPatterns {

        /*!
         * @brief Get the set of search directions containing all
    coordinate directions, bot positive and negative.
         *
         * This is the set used by CompassSearch.
         *
         * For 2D, the set is:
         * [1, 0], [0, 1], [-1, 0], [0, -1]
         *
         * Which looks like +.
```

```
         *
38       * @param num_vars The number of variables in the problem.
         * @return The set of search directions in all coordinate
    directions, positive and negative.
40       */
        inline std::vector<Eigen::VectorXi> Compass(int num_vars) {
42          auto directions = std::vector<Eigen::VectorXi>(2*num_vars);
            for (int i = 0; i < num_vars; ++i) {
44              Eigen::VectorXi dir = Eigen::VectorXi::Zero(num_vars);
                dir(i) = 1;
46              directions[i] = dir;
                directions[i+num_vars] = (-1) * dir;
48          }
            return directions;
50      }

52
        inline std::vector<Eigen::VectorXd> fast_pattern(int
    actual_dimensions_for_each_well , int num_vars ) {
54

56          if (actual_dimensions_for_each_well == 2){

58
                auto pattern = std::vector<Eigen::VectorXd>(num_vars/2*3);
60              for (int i = 0; i < pattern.size(); ++i) {
                    pattern[i] = Eigen::VectorXd::Zero(num_vars);
62              }

64              for (int j = 0; j < num_vars/2 ; ++j) {
                    pattern[3*j][2*j] = 1;
66                  pattern[3*j+1][2*j+1] = 1;
                    pattern[3*j+2][2*j] = -1;
68                  pattern[3*j+2][2*j+1] = -1;
                }
70
                cout << "the fast pattern is " << endl;
72              for (int k = 0; k < pattern.size(); ++k) {
                    cout <<"Next" << endl << pattern[k] << endl;
74              }

76              return pattern;
            }
78          else if(actual_dimensions_for_each_well == 3 ){

80

82              auto pattern = std::vector<Eigen::VectorXd>(num_vars/3*4);
                for (int i = 0; i < pattern.size(); ++i) {
84                  pattern[i] = Eigen::VectorXd::Zero(num_vars);
                }
86
                for (int j = 0; j < num_vars/3 ; ++j) {
88                  pattern[4*j][3*j+2] = 1;
                    pattern[4*j+1][3*j+1] = 1;
90                  pattern[4*j+1][3*j+2] = -1;
                    pattern[4*j+2][3*j] = 1;
```

```
                    pattern[4*j+2][3*j+2] = -1;
                    pattern[4*j+3][3*j] = -1;
                    pattern[4*j+3][3*j+1] = -1;
                    pattern[4*j+3][3*j+2] = -1;
                }

                cout << "the fast pattern is " << endl;
                for (int k = 0; k < pattern.size(); ++k) {
                    cout <<"Next" << endl << pattern[k] << endl;
                }

                return pattern;
            }
        }

        inline std::vector<Eigen::VectorXd> compass_pattern(int
    actual_dimensions_for_each_well, int num_vars) {


            if (actual_dimensions_for_each_well == 2){

                auto pattern = std::vector<Eigen::VectorXd>(num_vars*2);
                for (int i = 0; i < pattern.size(); ++i) {
                    pattern[i] = Eigen::VectorXd::Zero(num_vars);
                }

                for (int j = 0; j < num_vars/2 ; ++j) {
                    pattern[4*j][2*j] = 1;
                    pattern[4*j+1][2*j+1] = 1;
                    pattern[4*j+2][2*j] = -1;
                    pattern[4*j+3][2*j+1] = -1;
                }

                cout << "the compass pattern is " << endl;
                for (int k = 0; k < pattern.size(); ++k) {
                    cout <<"Next" << endl << pattern[k] << endl;
                }



                return pattern;
            }
            else if(actual_dimensions_for_each_well == 3 ){

                auto pattern = std::vector<Eigen::VectorXd>(num_vars*2);
                for (int i = 0; i < pattern.size(); ++i) {
                    pattern[i] = Eigen::VectorXd::Zero(num_vars);
                }

                for (int j = 0; j < num_vars/3 ; ++j) {
                    pattern[6*j][3*j] = 1;
                    pattern[6*j+1][3*j+1] = 1;
                    pattern[6*j+2][3*j+2] = 1;
                    pattern[6*j+3][3*j] = -1;
                    pattern[6*j+4][3*j+1] = -1;
                    pattern[6*j+5][3*j+2] = -1;
                }
```

```
148
                    cout << "the compass pattern is " << endl;
150                 for (int k = 0; k < pattern.size(); ++k) {
                        cout <<"Next" << endl << pattern[k] << endl;
152                 }

154

                    return pattern;
156             }
            }
158     }
}
160
#endif //FIELDOPT_GSS_PATTERNS_HPP_H
```

gss_patterns.hpp

Appendix F

Initial Well placement

```
COMPDAT
    PROD-10 74 89 1 1 OPEN 1* 7.35869 0.1905 1* 1* 1* Z /
    PROD-10 74 89 2 2 OPEN 1* 2.55659 0.1905 1* 1* 1* Z /
    PROD-10 74 89 3 3 OPEN 1* 0.624908 0.1905 1* 1* 1* Z /
    PROD-10 74 89 4 4 OPEN 1* 0.0011354 0.1905 1* 1* 1* Z /
    PROD-10 74 89 5 5 OPEN 1* 18.4681 0.1905 1* 1* 1* Z /
    PROD-10 74 89 6 6 OPEN 1* 13.0011 0.1905 1* 1* 1* Z /
    PROD-10 74 89 7 7 OPEN 1* 0.216822 0.1905 1* 1* 1* Z /
/
```

Appendix G

Optimized Well placement

```
COMPDAT
    PROD-10 63 87 1 1 OPEN 1* 516.131 0.1905 1* 1* 1* X /
    PROD-10 64 87 1 1 OPEN 1* 368.414 0.1905 1* 1* 1* X /
    PROD-10 65 87 2 2 OPEN 1* 431.684 0.1905 1* 1* 1* X /
    PROD-10 66 87 3 3 OPEN 1* 314.849 0.1905 1* 1* 1* X /
    PROD-10 67 87 3 3 OPEN 1* 150.664 0.1905 1* 1* 1* Z /
    PROD-10 67 87 4 4 OPEN 1* 0.0044036 0.1905 1* 1* 1* X /
    PROD-10 68 87 4 4 OPEN 1* 0.00346145 0.1905 1* 1* 1* Z /
    PROD-10 68 87 3 3 OPEN 1* 147.863 0.1905 1* 1* 1* X /
    PROD-10 69 87 3 3 OPEN 1* 23.5874 0.1905 1* 1* 1* Z /
    PROD-10 69 87 2 2 OPEN 1* 173.423 0.1905 1* 1* 1* X /
    PROD-10 70 87 2 2 OPEN 1* 253.454 0.1905 1* 1* 1* X /
    PROD-10 71 87 1 1 OPEN 1* 452.146 0.1905 1* 1* 1* X /
    PROD-10 72 87 1 1 OPEN 1* 171.94 0.1905 1* 1* 1* X /
    PROD-10 73 87 1 1 OPEN 1* 78.2168 0.1905 1* 1* 1* Z /
    PROD-10 73 87 2 2 OPEN 1* 189.706 0.1905 1* 1* 1* X /
    PROD-10 74 87 2 2 OPEN 1* 308.939 0.1905 1* 1* 1* X /
    PROD-10 75 87 2 2 OPEN 1* 45.6866 0.1905 1* 1* 1* X /
/
```