Norwegian University of
Science and Technology

Implementation of Particle Swarm Optimization Algorithm within FieldOpt Optimization Framework

Application of the algorithm to well
placement optimization

Chingiz Panahli

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

Norwegian University of Science and Technology
Department of Geoscience and Petroleum

Abstract

In the present work, particle swarm optimization algorithm is applied in well placement optimization problem using FieldOpt, a software framework that aims at being common platform for field development optimization. Different types of particle swarm optimization algorithm have been developed and integrated inside FieldOpt optimization framework in order to increase the capabilities of the framework for field development optimization problems. The algorithm that has been implemented in this work is simple and flexible and it can be used easily as hybrid with other algorithms in the future. We have applied particle swarm optimization algorithm to optimize the locations of vertical and horizontal wells in simple and realistic reservoir models. A dynamic penalty function was coupled inside the algorithm in order to treat the non-linear constraints, such as well length, inter-well distance and reservoir boundary constraints.

We first apply the algorithm on a simple two dimensional five spot model that consists of four vertical injectors and one horizontal producer. We run particle swarm optimization in order to check the performance of the algorithm and determine optimal parameters by performing a sensitivity analysis. Three cases are run on the simple model and the cases vary in terms of the type of constraints implemented. In all the cases, we use fixed control parameters for injectors and producers. In the first case, we optimize the location of one production well that is subject to the well length constraint only. In the second case, we use both well length and reservoir boundary constraints. In the third case, we use all the constraints in order to optimize the location of two horizontal producers. The results suggest that the algorithm converged to an optimum solution with given constraints and assumptions for the simple five spot model.

After getting satisfactory results on the simple reservoir model, we apply particle swarm optimization on a new benchmark known as OLYMPUS. This is a new field development challenge that is launched by TNO (Netherlands Organization for Applied Scientific Research) within the context of the ISAPP (International Scientific Association for Probiotics and Prebiotics) research program. The reservoir model is characterized with faults, horizontal barriers across the different zones and high permeability channels, which makes the optimization problem more complex than the simple five spot model. The reference operating strategy of the field consists of six injectors and eight producers that are controlled by a pressure constraint. We optimize the locations of first five production wells while fixing the locations and operating parameters of the other wells as same as in the reference case. Because of high computational demand and time limitations, we could not use enough simulations to reach an optimum solution in this case. However, we compare the optimization runs and give suggestions for future work for the well placement part of the OLYMPUS field development plan.

Preface

This thesis is written as part of the Master's degree in Petroleum Engineering, at the Department of Geoscience and Petroleum, NTNU. It was written during the spring semester of 2017 under the supervision of Prof. Jon Kleppe and Postdoc Mathias Bellout.
This work is based on the application of several tools and concepts, such as C++ programming language, numerical methods in petroleum, reservoir simulation and other relevant terminology. Therefore, it is assumed that the reader is familiar with those concepts.
The algorithm developed during this thesis enhances the capabilities of the FieldOpt framework. The main improvements and new functionalities have been made in the optimizer interface of the FieldOpt. The most remarkable result of these improvements is that the optimizer interface is now better organized and more robust. This will make the addition of new algorithms easier in the future.

Acknowledgments

I would like to thank my supervisor, Professor Jon Kleppe, for his continuous support and guidance, and my co-supervisor, Postdoc Mathias Bellout, for his help and advice on almost every aspect of this thesis - without his expertise in optimization theory I would not be able to complete this thesis.
I would also like to express my gratitude to Mansoureh Jesmani for the valuable insight that she gave me. Lastly, thanks to Einar Baumann and Hilmar Magnusson for their continuous help and advice on programming during this thesis.

Table of Contents

Abstract i

Preface iii

Acknowledgments v

Table of Contents viii

List of Figures x

Abbreviations xi

1 Introduction 1
- 1.1 Scope of the work 2
- 1.2 Objectives 2
- 1.3 Thesis outline 3

2 Literature review 5
- 2.1 Well placement optimization 5
- 2.2 PSO in well placement optimization 8

3 Optimization 13
- 3.1 Problem description 13
- 3.2 Objective function 15
- 3.3 Constraints 16
  - 3.3.1 Bound constraints 16
  - 3.3.2 Practical constraints 17
- 3.4 Optimization algorithm 19
  - 3.4.1 Global Best PSO 20
  - 3.4.2 Local Best PSO 23
  - 3.4.3 Neighborhood topologies 23
  - 3.4.4 Parameters 25

3.5 Constraint handling methods . . . 26
3.5.1 Penalty method . . . 27
3.6 Optimization Framework . . . 27

4 Implementation **29**
4.1 FieldOpt . . . 29
4.1.1 Driver file . . . 30
4.1.2 Runner . . . 31
4.1.3 Optimizer interface . . . 31
4.1.4 Simulator interface . . . 31
4.1.5 Case class . . . 31
4.2 PSO Integration . . . 32
4.3 Particle class . . . 33
4.4 PSO class . . . 34
4.4.1 Iterate method . . . 36
4.4.2 Initialize cases method . . . 36
4.4.3 Apply penalty method . . . 36
4.4.4 Update Global Best Case method . . . 37
4.4.5 Update Personal Best Cases method . . . 38
4.4.6 Update Particles method . . . 38

5 Case Study **39**
5.1 Model descriptions . . . 39
5.1.1 5 spot model . . . 39
5.1.2 Olympus model . . . 41
5.2 Optimization results and discussion . . . 43
5.2.1 5 spot results . . . 43
5.2.2 Olympus results . . . 58

6 Conclusion and Recommendations for Further Work **63**

Bibliography **64**

Appendix **67**

List of Figures

3.1 Well trajectory. 14
3.2 Movement of a particle. 20
3.3 Global best PSO (maximization problem) 22
3.4 Star or gbest neighborhood topology 24
3.5 Ring or lbest neighborhood topology 25

4.1 FieldOpt working principle. 30
4.2 Class diagram for Case class 32
4.3 Main classes and class relationship. 33
4.4 Particle class. 34
4.5 PSO class. 35

5.1 5 Spot Model. 39
5.2 Permeability and Porosity distribution 40
5.3 Relative Permeabilities. 40
5.4 Olympus model 41
5.5 Permeability distribution for the first realization. 42
5.6 Porosity distribution for the first realization 42
5.7 Sensitivity analysis for different swarm sizes and maximum number of iterations. 43
5.8 Average NPV over five runs vs number of simulations for different tunings 44
5.9 Average NPV and NPV of five runs vs number of simulations for Tune 3. 45
5.10 Field total oil productions for the first case (scaled). 46
5.11 Field total water productions for the first case (scaled). 47
5.12 Oil saturations at 2920 day for different runs in Tune 3. 49
5.13 Average NPV and NPV of five runs vs number of simulations (Case 2). . 49
5.14 Field total oil productions for the second case (scaled). 50
5.15 Field total water productions for the second case (scaled). 51
5.16 Oil saturations at 2920 day for different runs. 53
5.17 Optimization results for the third case. 54

5.18 Field total oil productions for the third case. 54
5.19 Field total water productions for the third case. 55
5.20 Recovery factors for the third case. 55
5.21 Oil saturations at 2920 day for different runs. 57
5.22 Initial well locations (Top view). 58
5.23 Regions. 58
5.24 Optimization results. 59
5.25 Comparison of field total oil and water productions. 60
5.26 Comparison of field water cut for different runs 60
5.27 Comparison of well locations for different runs. 62

6.1 A simple class diagram 67

Abbreviations

bGA	=	Binary Genetic algorithm
CAPEX	=	Capital expenditure
CF	=	Cash flow
cGA	=	Continuous Genetic algorithm
CP-PSO	=	Centered-Progressive Particle Swarm Optimization
DE	=	Differential evaluation
GA	=	Genetic algorithm
gbest PSO	=	Global best Particle Swarm Optimization
GenOpt	=	Genetic Optimization Program
GPS	=	Generalized Pattern Search
HJDS	=	Hooke-Jeeves direct search
HPSDE	=	Hybrid Particle Swarm Differential Evaluation
lbest PSO	=	Local best Particle Swarm Optimization
MADS	=	Mesh adaptive direct search
MPI	=	Message passing interface
MPSO	=	Modified Particle Swarm Optimization
NPV	=	Net present value
PSO	=	Particle Swarm Optimization
PUNQ-S3	=	Production Forecasting with Uncertainty Quantification
QM	=	Quality map
SPSA	=	Simultaneous perturbation stochastic approximation
SPSO	=	Standart Particle Swarm Optimization

Chapter 1

Introduction

Development of efficient and robust optimization algorithms for generalized oil field development problems has been an area of active research in recent years. The optimization techniques have been applied to different areas of general oilfield problems.

Field development optimization is usually studied as two separate problems, such as well placement optimization and well control optimization. The main goal of the optimization is to maximize an economical indicator or some performance measure by finding the optimal well configuration or operating parameters. Well placement optimization is considered to be more complex than well control optimization because reservoir heterogeneities produce highly non-smooth objective functions containing multiple local optimums. The computational demand is very high for these type of problems, since many different simulations need to be run in reservoir simulators. In a real field development project, simulation time and number of optimization variables may be substantial and inclusion of geological uncertainty further increases the complexity of the optimization problem. Therefore, the optimization algorithms applied to these type of problems need to be efficient and robust in order to handle the complexity of the problem. Many optimization algorithms have been applied to well placement optimization problem in recent years. Most of these algorithms are categorized as either derivative free or gradient based methods.

Gradient based methods have the disadvantage of getting stuck in local minimums and they need extra calculations for the derivatives of the objective function. As a result, gradient based methods have not found much applicability in well placement optimization. However, derivative free methods do not need the calculation of derivatives and they are less susceptible to get stuck in local minimums. Among derivative free algorithms, evolutionary algorithms and swarm intelligence algorithms (particle swarm optimization) proved to perform better than others. Genetic algorithms are one of the most used evolutionary algorithms in well placement optimization [1, 2, 3]. However, some of the swarm intelligence algorithms such as particle swarm optimization showed better results compared with genetic algorithms in recent studies [4, 5].

Particle Swarm Optimization is a novel population based stochastic algorithm and it was

first introduced by Kennedy and Eberhart in 1995 [6]. Since then, it has been used as a robust method to solve optimization problems in a wide variety of applications. Algorithm also became very popular in well placement optimization due to its simplicity, effectiveness and the ability to converge to a good solution in a reasonable amount of time.
In this thesis, we will develop and apply particle swarm optimization for well placement optimization using FieldOpt, a software framework that aims at being common platform for field development optimization (For more information see: `https://github.com/PetroleumCyberneticsGroup/FieldOpt`).

1.1 Scope of the work

FieldOpt optimization framework is a general reservoir-simulation based framework that is developed by Petroleum Cybernetics Group at NTNU. FieldOpt is developed to serve as a common platform for those who want to apply different optimization techniques in field development problems and also conduct research in these areas. The framework includes a variety of derivative free optimization methods that can be applied to general petroleum field development problems. Our overall goal entails the development and integration of particle swarm optimization algorithm inside FieldOpt optimization framework for well placement optimization.

1.2 Objectives

The PSO algorithm is developed based on the simulation of the social behavior of animals, such as bird flocking, fish schooling in order to solve optimization problems. Each member of the population is called a particle and the population is called a swarm. Initially, all the particles are distributed randomly inside the search space. After initialization, each particle updates their velocity and move through the search space by remembering the best positions of themselves (personal best position) and the best position of the whole swarm (global best position). In each iteration, particles find better positions by moving in the stochastic average direction of the personal and global best positions. The procedure repeats until a stopping criteria such as maximum number of iterations is reached.
Different types of particle swarm optimization algorithm have been developed based on the complexity and type of the optimization problem recently. We will focus on the implementation of basic particle swarm optimization in this thesis. In addition, we will also implement the two different types of basic particle swarm optimization that are known as "local best" particle swarm optimization and "global best" particle swarm optimization. These two versions use different kind of neighborhood topologies (grouping of particles into small sub-groups), which affect the convergence speed and the performance of the algorithm. Global best particle swarm optimization uses a "star" neighborhood topology where all the particles communicate with each other. Local best particle swarm optimization can be used with different neighborhood topologies with the "ring" topology being the most common. We will implement "random" neighborhood topology in addition to the "ring" neighborhood topology for local best particle swarm optimization.
In addition to the algorithm and neighborhood topologies, an efficient constraint handling

method is also needed to treat the constraints, since particle swarm optimization algorithm lacks this mechanism in its original design.
After mentioning the most important topics, the main objectives of this thesis can be summarized as below:

- Developing different versions of particle swarm optimization algorithm
- Integrating the algorithm inside FieldOpt optimization framework
- Implementing an efficient constraint handling method to treat constraints
- Applying particle swarm optimization technique for well placement optimization

1.3 Thesis outline

This thesis is organized as follows:

In next section, we describe the main optimization algorithms applied in well placement optimization. We then review the literature related to the application of particle swarm optimization in well placement optimization. Different forms of particle swarm optimization that are applied in well placement optimization and joint optimization of well placement and well controls are presented in this chapter.

In chapter 3, we describe the components of the optimization system, such as problem description, particle swarm optimization algorithm and different types of the algorithm that are implemented in this thesis, constraints and constraint handling methods that are used in our implementation. We also give brief information about the optimization framework in this chapter, since it is one of the most important component of the optimization system.

In chapter 4, we give detailed information about optimization framework and describe the implementation and integration of particle swarm optimization inside the framework.

In chapter 5, we describe the simple five spot model and OLYMPUS model. After describing the models, we present the optimization results for both models and discuss the results.

In chapter 6, we conclude this thesis and give recommendations for future work.

Chapter 2

Literature review

In this section, we will first briefly present the literature related to the optimization techniques that are applied in field development optimization in general. After then, a more in depth review of the literature related to the application of particle swarm optimization in well placement optimization will be presented.
Particle swarm optimization is successfully applied in both well placement and joint optimization problems and it showed better results compared with other algorithms such as genetic algorithms. In addition, many authors applied different modifications for improving the performance of the algorithm in well placement optimization. These modifications include hybridization of the particle swarm optimization with other algorithms, application of the meta-optimization routines to tune the parameters during optimization, incorporation of well pattern operators for large scale field development optimization, etc. We will give more information about these techniques in the following sections.

2.1 Well placement optimization

Well placement optimization is a high dimensional, complex and constrained optimization problem. In large scale applications, number of optimization variables may be substantial and search space can contain multiple optimums depending on the reservoir heterogeneities. Therefore, a large number of function evaluations may be required for each run. In addition, many runs may be required in order to get satisfactory results that represent the optimal solution of the problem. Therefore, the optimization algorithms used for this type of problems need to be efficient, robust and simple.
We can distinguish the optimization algorithms applied in well placement optimization problem within two main categories known as derivative-free and gradient based algorithms. Derivative-free methods are less prone to get stuck in local optimums compared with gradient based methods. However, derivative-free methods may require more function evaluations to get a good solution. On the other hand, gradient based algorithms are computationally fast because the gradient information at a particular point can be used to move rapidly towards an optimum point, but they are more susceptible to get stuck in lo-

cal optimums, especially in heterogeneous reservoirs where the porosity and permeability values vary throughout the reservoir. In addition, gradient based methods require precise knowledge and access to the computations that are used to evaluate the objective function values. Usually, it is difficult to obtain more than just the objective function value from commercial reservoir simulators.

Derivative-free methods. Derivative-free methods are widely used in well placement optimization, since they do not require gradient information as in gradient-based methods and they are not invasive with respect to the reservoir simulators. We can distinguish the main approaches to derivative-free optimization methods as:

- Stochastic methods
- Deterministic methods

Stochastic methods are very popular because they are effective in avoiding the local optimums. These methods use a randomized search to avoid local optimums. Note that, randomized search does not mean unstructured search or search in random directions. These methods include some random parameters in their formulation and these help them to avoid getting stuck in local optimums.
The second approach refers to the pattern-search techniques. Pattern-search techniques use stencils to explore the search space. For example, Generalized Pattern Search (GPS), Mesh Adaptive Direct Search (MADS) and etc.

Stochastic methods. Stochastic or global search methods rely on the random components in order to explore a larger amount of search space by reducing the chance of getting stuck in local optimums. Although these algorithms are effective, it is difficult to analyze them mathematically. Therefore, they are not supported by a solid convergence theory compared with deterministic approaches. Many of the stochastic methods are population based methods. This means that multiple function evaluations are carried out in each iteration. The structure of these algorithms are very flexible, which allows easy modification and coupling with other methods. However, these algorithms still need tuning of the parameters, which can influence their performance significantly. For example, there is still a lack of theoretical knowledge about the optimal population size for these algorithms. It is usually problem dependent and it can take a considerable amount of time and computation to find the optimal population size for each problem. Usually, it can be expected that the larger the population size, the more globally the search is done because the more area of the search space is covered. Note that, although these methods are global in nature, they may serve as a local search technique in cases where the population size is small compared with the number of optimization variables.
Many stochastic methods have been developed in the last decades. We will only focus on the genetic algorithms in this section, which are the most widely used algorithms in well placement optimization [1, 2, 3].

Genetic algorithms. Genetic algorithms [7] are population-based stochastic search methods that work on the basis of the biological evolution principle. In genetic algorithms, solutions are encoded in a special structure called "chromosome" and objective function values are referred to as "fitness". In each iteration, "offspring" chromosomes are generated by applying special genetic operators (e.g., selection, crossover and mutation) to the "parent" chromosomes, which are selected based on the fitness values. This procedure repeats in each iteration until a convergence criteria is reached [7].
Genetic algorithms are categorized in two main groups known as binary genetic algorithms (bGA) and continuous genetic algorithms (cGA). These two algorithms use different methods to encode and manipulate the solutions. In binary genetic algorithms, solutions are encoded as a sequence of zeros and ones, while in continuous genetic algorithms, encoding and manipulating of the solutions are carried out in real space. Both of these algorithms have been successfully applied in well placement optimization problem. For example, in [8], both of them were applied in well placement optimization and their performance were compared. Continuous genetic algorithms showed better results compared with binary genetic algorithms in the paper.
Flexibility and simplicity of these algorithms allow easy coupling of them with other algorithms. As a consequence, many of the studies are focused on the hybrid applications of genetic algorithms in well placement optimization. For example, in the paper presented by Bittencourt and Horne [9], genetic algorithm was hybridized with Polytope and Tabu search algorithms and applied in a real field development project in order to optimize the locations of wells. Another hybrid genetic algorithm was implemented in [10] to optimize the locations of wells by considering uncertainties in well placement optimization.

Deterministic methods. Many of the deterministic methods are pattern search methods. Unlike stochastic algorithms, these algorithms are supported by a mathematical solid convergence theory [11]. Pattern search algorithms use stencils to explore the search space and they work as follows. Firstly, objective function values are calculated for all points and compared with the stencil center. If one points improves the objective function value compared with the stencil center, the stencil is moved to that point and that point becomes the new center. When all the points do not improve the objective function value, the stencil size is decreased and the procedure repeats again until minimum stencil size is reached. These algorithms naturally parallelize because function evaluations for all the points can easily be calculated in parallel. Therefore, these algorithms can perform quite fast when running in parallel. However, they can easily get trapped in local optimums which is one of the biggest disadvantage of these algorithms. As a consequence, these algorithms are usually used with a large stencil size at the beginning of the optimization and the stencil size is decreased over time, in order to increase the global search abilities of these algorithms. Popular pattern-search algorithms in well placement optimization, just to name a few, are Generalized Pattern Search (GPS), Mesh Adaptive Direct Search (MADS) and Hooke-Jeeves Direct Search (HJDS). Since these algorithms easily parallelize, they become very popular in complex optimization problems where many function evaluations are required such as joint optimization problems. For instance, in [12], some of these popular pattern search methods, such as Generalized Pattern Search (GPS) and Hooke-Jeeves direct search are applied for joint optimization of well control and well location variables.

Pattern search methods are also hybridized with stochastic algorithms in order to take advantage of both local search abilities of the pattern search methods and global search abilities of stochastic methods. For example, in [13], Mesh Adaptive Direct Search was hybridized with Particle Swarm Optimization (PSO) and applied in joint optimization of well control and well location variables.

2.2 PSO in well placement optimization

Initial applications. Particle Swarm Optimization was first applied in well placement optimization in the paper presented by Onwunalu and Durlofsky [4]. They found out that on average, particle swarm optimization outperforms one of the most widely used algorithm in well placement optimization known as genetic algorithms. They have used particle swarm optimization with random neighborhood topology, which is referred to as "local best particle swarm optimization". The algorithm was used to optimize the locations of vertical, deviated and multi-lateral wells by including the geological uncertainty in reservoir models considered in the paper. A penalty method was applied to treat the infeasible cases during optimization, by assigning large negative values to their objective function values. Net present value was considered as the objective function and objective function evaluations were carried out in parallel using a cluster of 50 processors. They have used "synchronous" method to update the personal best cases. In synchronous method, personal best positions of all particles are updated after evaluating objective function values of all particles in each iteration. In their paper, they have applied particle swarm optimization on standard benchmark problems and suggested improvements on particle swarm optimization as a further work in order to apply the algorithm for large scale field development problems or real reservoirs.

PSO in large scale field development. Large scale field development includes a large number of wells ranging from a few to several hundreds of wells. These wells can be vertical, horizontal or non-conventional wells in a real life scenario and number of optimization variables may be very large. This can decrease the performance of the particle swarm optimization algorithm significantly. In addition, inclusion of realistic constraints may decrease the performance of the algorithm even more. To overcome this issue many methods have been considered in order to decrease the size of search space and optimization variables. For example, many different feasible reservoir regions may be considered instead of the whole reservoir in order to decrease the number of possible solutions or search space. Another technique is considering the wells in patterns during optimization. Incorporation of these techniques may increase the performance of particle swarm optimization significantly in large scale field development problems.
Onwunalu et.al incorporated well pattern optimization procedure inside particle swarm optimization algorithm in a further work and applied the new algorithm for large scale field development optimization problem [5]. Results showed that the new algorithm gave better results compared with standard particle swarm optimization. They have also implemented a meta-optimization procedure. The procedure optimized the parameters of particle swarm optimization during optimization process. The meta-optimization procedure was implemented in two steps. In first step, the parameters of the algorithm were

optimized and in second step, the algorithm was run with optimized parameters in order to optimize the locations of wells. They concluded that the inclusion of well pattern optimization and meta-optimization procedure inside particle swarm optimization increased the performance of the algorithm in large scale field development problems.

Constraint handling in PSO. Another important factor which can affect the performance of particle swarm optimization in large scale field development problems is constraint handling method. The importance of constraint handling methods was studied by Jesmani et.al in [14]. They incorporated realistic constraints such as well length, inter-well distance, well orientation constraints inside particle swarm optimization and used penalty and decoder constraint handling techniques to treat the constraints. The GenOpt optimization framework was used in order to apply the particle swarm optimization. They have used particle swarm optimization with Von-Neumann neighborhood topology and a linear decreasing inertia weight strategy was also chosen. They run two cases using penalty and decoder techniques. The first case included a 60x60 2D reservoir model with four injectors located at the corner and a horizontal producer in the center of the model. In the second case, they used a synthetic reservoir model with real reservoir properties from a North Sea field. The decoder technique was observed to perform better than penalty method in the paper. They concluded that the decoder technique can be used as a viable alternative to widely used penalty method for particle swarm optimization in well placement optimization.

Hybrid PSO. Nwankwor et al. applied a hybrid form of the particle swarm optimization with an evolutionary algorithm known as the differential evaluation (DE) for well placement optimization [15]. The new algorithm was called hybrid particle swarm differential evolution (HPSDE) algorithm. They considered three cases and uncertainty was taken into the account in two of these cases. For the first case, they used a 2D reservoir model and incorporated the geological uncertainty by generating ten different permeability distributions. The system contained only oil and connate water and they optimized the placement of a single production well. The MATLAB Reservoir Simulation Toolbox was used for running the simulations and evaluating the objective function values. They also performed a sensitivity analysis in order to determine the effects of parameters on the performance of the algorithms. In the second case, they considered the placement of one injector and a producer in water-flooding process. In the third case, they also optimized the location of injectors and producers. But in that case they considered a 3D reservoir model and they neglected the geological uncertainty. Vertical wells were considered in all cases and results from multiple runs were averaged in order to make meaningful comparisons. The results showed that the hybrid particle swarm differential evolution performed better than the particle swarm optimization and the differential evolution algorithms. However, they also noted the importance of parameter tuning for hybrid particle swarm differential evolution algorithm.

PSO in joint optimization. Well placement optimization and well control optimization were traditionally considered as separate optimization problems. However, recent works

[12, 16] demonstrated that optimal locations of wells also depend on the operating parameters such as, well rates and bottomhole pressures. Some of these works focused on the application of the particle swarm optimization for joint optimization of well control and well location variables. In most of these methods, particle swarm optimization was hybridized with other algorithms and applied in joint optimization.

Hybrid PSO in joint optimization. In [17] Particle swarm optimization was coupled with Mesh Adaptive Direct Search (MADS) and applied in joint optimization of well control and well location variables. The new hybrid algorithm took the advantage of stochastic nature of particle swarm optimization and Mesh Adaptive Direct search method, which is a local optimization technique with a proven convergence theory. They used a filter-based approach to treat the non-linear constraints during optimization. Filter based approach is a bi-objective optimization technique, where two separate optimization problems are considered. The first objective is to maximize the net present value while the second objective is to minimize the constraint violations during optimization. Geological uncertainty was not considered in the work. They also introduced a new binary variable (drill/do not drill) to determine the optimum number of wells in addition to the optimum locations. They applied new hybrid algorithm and standalone PSO and MADS for sequential and joint optimization. Results showed that new hybrid algorithm outperformed standalone PSO and MADS, it was also shown that joint optimization gave better results compared with sequential approach. A similar study was also conducted in [18].

Modified PSO applications. Ding et al. [19] used a modified form of particle swarm optimization in combination with the quality map method (MPSO+QM) for well placement optimization and compared the performance of the algorithm with the standard particle swarm optimization (SPSO), centered-progressive particle swarm optimization (CP-PSO) and the modified particle swarm optimization (MPSO). The modification in particle swarm optimization was applied to the inertia weight, the velocity and the position update equations and they introduced a flying time factor in their paper. The net present value (NPV) was taken as the objective function and the ECLIPSE 100 reservoir simulator was used for objective function evaluation. They considered four test cases and optimization was applied to only vertical wells. In the first case, they optimized the location of a vertical production well using all methods described above and they considered the reservoir uncertainty by applying six different realizations of the permeability field. In the second case, the location of six production wells were considered on a 2D reservoir model. In the third case, the location of three producers and two injectors were optimized. In the last case, they used a standard benchmark known as the PUNQ-S3 (Production Forecasting with Uncertainty Quantification) model. The location of ten production wells were optimized. For each case, multiple runs were performed due to the stochastic nature of the particle swarm optimization and the average results were compared. They concluded that the QM+MPSO method outperformed all methods and for the first case where a single well location optimization was considered the quality map was not needed, since the modified particle swarm optimization (MPSO) showed best results in that case.

Modified PSO for non-conventional wells. In a further work, Ding et al. [20] used the same modified particle swarm optimization (MPSO) for optimizing the well location, type and geometry. They established an optimization model of the well trajectory, which could be applied to the vertical, horizontal and deviated wells by changing the model parameters according to the well type. They run three cases on the PUNQ-S3 model using modified particle swarm optimization (MPSO) and standard particle swarm optimization (SPSO). The parameters for the particle swarm optimization were kept same for all cases. In the first case, the location of six vertical wells were optimized using both algorithms. In the second case, the location and trajectory of six deviated wells were considered. And in the third case, the location of one injection well and five production wells were optimized. The results from multiple runs for both algorithms were averaged in each case and the net present values (NPV) were plotted versus the number of iterations. Performance of standard particle swarm optimization (SPSO) and modified particle swarm optimization (MPSO) were compared for all cases. The modified particle swarm optimization showed better results compared with standard particle swarm optimization. They also made comparisons of the highest net present values between cases. Firstly, the comparison between the first and the second case was made. The second case showed lower value of the net present value compared with the first case. Although the net present value was lower for the second case, the oil production was higher compared with the first case. The reason behind this phenomena was explained by the increasing cost of the horizontal wells in the second case. The comparison between the second and the third case was also made. The third case showed lower value of the net present value compared with the second case but two cases had almost same liquid production. This phenomena was explained by the increasing cost of early water injection in the third case and they concluded that the early water injection was not reasonable for the development of the PUNQ-S3 model. However, they suggested that one or two injection wells could be added in the later periods.

Chapter 3

Optimization

In this section, we first describe the well placement problem in general. We then describe the components, such as constraints, objective function, optimization framework and optimization algorithm that are used to solve the well placement problem.

3.1 Problem description

In this work, well trajectory will be handled as a straight line in three-dimensional space (3D). The well trajectory will be defined by a starting point or heel, H, and an end point or toe, T. Each point on a grid can be represented either by its real coordinates x, y, z in real space, R, or its directional indices i, j and k in grid space, G. The parameters to be optimized become the coordinates or the indices that define the heel and toe (H and T, respectively) of the wellbore as shown in Figure 3.1.

However, our implementation of the optimization algorithm is based on the continuous variables, which are the real coordinates of the heel and toe of the wellbore. Meanwhile, most of the reservoir simulators require well locations to be defined in terms of discrete grid blocks.

FieldOpt optimization framework uses a mapping function, m, to transform the grid to real space or vice versa, as shown in Eq. (3.1).

$$m : G \leftrightarrow R \tag{3.1}$$

The heel and toe coordinates will be transformed from grid to real space or vice versa during optimization, via the mapping function, m, as shown in Eq. (3.2).

$$\begin{aligned} H = R : H(x, y, z) \overset{\text{m}}{\leftrightarrow} G : H(i, j, k) \\ T = R : T(x, y, z) \overset{\text{m}}{\leftrightarrow} G : T(i, j, k) \end{aligned} \tag{3.2}$$

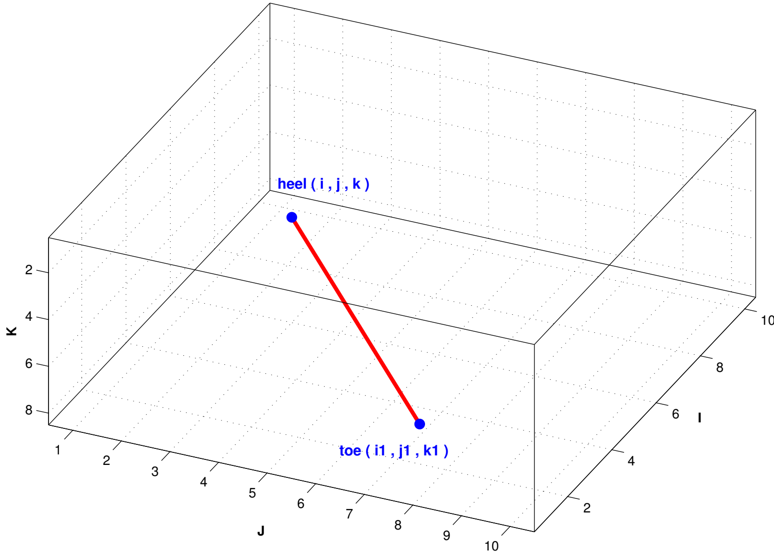


Figure 3.1: Well trajectory.

The optimization problem can now be stated in the general form as in Eq. (3.3).

$$\max_{\mathbf{x}\in X,\mathbf{u}\in U} f(\mathbf{p},\mathbf{x},\mathbf{u}) \quad or \quad \min_{\mathbf{x}\in X,\mathbf{u}\in U} -f(\mathbf{p},\mathbf{x},\mathbf{u})$$
$$\text{subject to :} \quad \begin{cases} g(\mathbf{p},\mathbf{x},\mathbf{u}) = 0, \\ c(\mathbf{p},\mathbf{x},\mathbf{u}) \leq 0, \end{cases} \tag{3.3}$$

where **f** represents objective function to be optimized (e.g., to maximize net present value, $f = NPV$ or cumulative oil production) and **g** represents flow equations for each grid block in a certain time step. This constraint ensures that reservoir flow equations are satisfied while running the simulation and it is used to evaluate objective function value and constraints. Nonlinear constraints, such as well length, inter well distance, constraints on well control variables and other constraints are represented by **c** function.
Dynamic state variables, such as pressure, saturation, compositions and others are represented by **p** vector while well location variables and well control variables are shown with **x** and **u** vector, respectively.

In our implementation, we will not focus on the joint optimization of the well location and well control variables. However, piecewise fixed values of well control variables in time with N_t time intervals will be considered. Therefore, above equation (3.3) needs to be simplified in order to exclude well control variables, **u**, and flow equations, **g**, since it is enforced during simulation.

The optimization problem can now be presented more specifically in a simplified form as

in Eq. (3.4).

$$\max_{\mathbf{x}\in X} f(\mathbf{x}) \quad or \quad \min_{\mathbf{x}\in X} -f(\mathbf{x})$$
$$\text{subject to :} \quad \left\{ \quad c(\mathbf{x}) \leq 0, \right. \tag{3.4}$$

The **x** vector contains both heel and toe vectors as defined in Eq. (3.2). The set $X = \{\mathbf{x} \in X^n;\, x_l \leq \mathbf{x} \leq x_u\}$ defines the allowable values for discrete well location variables or reservoir boundaries. The dimension of **x** or the number of optimization variables, n, depends on the trajectory, type and number of wells.
If we optimize the location of injectors and producers and the wells are assumed to be vertical (i, j), there will be $n = 2(N_I + N_P)$ optimization variables. If we optimize only horizontal production wells with fixed vertical injectors, then we have $n = 6N_P$ optimization variables.

3.2 Objective function

We will consider the net present value (NPV) as the objective function. To calculate the NPV, the cash flow (CF) is first calculated based on the profit. The profit is calculated as the difference between revenue (R) obtained by the sales of the hydrocarbons and the operational expenses (E), such as water production and injection costs. The cash flow at time t is calculated using the following formula given below:

$$CF_t = R_t - E_t \tag{3.5}$$

Since the revenue and expenses at time t depend on the fluid volumes produced at time t, a reservoir simulation has to be run in order to determine the fluid production. The resulting fluid production profiles generated from the simulation run are used to calculate the revenues and expenses as follows:

$$R_t = p_o Q_t^o + p_g Q_t^g \tag{3.6}$$

where p_o and p_g represent the oil price (\$/m^3) and gas price (\$/m^3); Q_t^o and Q_t^g represent the total volume of the oil (m^3) and gas (m^3) produced at time t. The operating expenses at time t is calculated as below:

$$E_t = p_w^p Q_t^{w,p} + p_w^i Q_t^{w,i} \tag{3.7}$$

where p_w^p represents the water production costs (\$/m^3); p_w^i represents the water injection costs (\$/m^3); $Q_t^{w,p}$ and $Q_t^{w,i}$ represent the total volumes of water produced (m^3) and injected (m^3), respectively at time t. Cash flow is then divided by the discount factor $(1+r)^t$ in order to get the discounted cash flow (present value). The cumulative discounted cash flow is obtained as the sum of all discounted cash flows. Finally, the NPV is obtained by subtracting the capital expenditures from cumulative discounted cash flow.

$$NPV = \sum_{t=1}^{T} \frac{CF_t}{(1+r)^t} - C^{capex} \tag{3.8}$$

where T is the total production time in years; r is the annual discount rate; C^{capex} is the total cost to drill and complete all of the wells.
The general formula presented above is modified to an equivalent form inside FieldOpt framework because the framework can only receive the final total productions from reservoir simulators. Therefore, we will assume annual discount rate as zero ($r = 0$). Considering these limitations, FieldOpt uses the following formula to calculate NPV:

$$NPV = (p_o Q_T^o + p_g Q_T^g - p_w^p Q_T^{w,p} - p_w^i Q_T^{w,i}) - C^{capex} \tag{3.9}$$

where Q_T^o, Q_T^g, $Q_T^{w,p}$, $Q_T^{w,i}$ represent the final total productions of oil, gas, produced and injected water. Capital expenditure, C^{capex} is assumed to be constant during optimization.

3.3 Constraints

In well placement optimization, two kinds of constraints that are usually considered are practical and bound constraints. Bound constraints are used to constrain the variable values to a specific range. These variables can be either well location variables or well control variables. For example, all wells must be located in the feasible regions inside the reservoir. Practical constraints are identified during the field development phase and examples include well length constraint, inter-well distance constraint, well orientation constraint and others. All these constraints are imposed on the well placement optimization problem and incorporation of these constraints increases the complexity of the optimization problem.
Different methods have been used for handling constraints. We will give detailed information about these methods in constraint handling section. In the following sections, we will describe the boundary and practical constraints used in our implementation.

3.3.1 Bound constraints

Well placement optimization problem is always subject to the bound constraints. Bound constraints can be applied either to well control variables or well location variables. As mentioned before, in this work, we will only consider well location variables, since the heel and toe points should lie inside reservoir boundaries.
During optimization, particles positions are updated in each iteration and some of the particles tend to leave the search space. To prevent this behavior, particles that violate the bound constraints are projected back to the boundaries of the search space by modifying their positions. In addition to the particle positions, the corresponding particle velocities need to be altered in order to prevent particles moving out of the search space again in next iteration. This technique is the most common way of dealing with bound constraints in particle swarm optimization and it is referred to as "absorb" technique. This technique

is applied as shown in Eq. (3.24).

$$x_j^i(k+1) = \begin{cases} if \quad x_j^i(k+1) > u_j \\ set \quad x_j^i(k+1) = u_j \quad and \quad v_j^i(k+1) = 0 \\ if \quad x_j^i(k+1) < l_j \\ set \quad x_j^i(k+1) = l_j \quad and \quad v_j^i(k+1) = 0 \end{cases} \tag{3.10}$$

In Eq. (3.24), index j shows component, while u_j and l_j represent upper and lower bounds for the jth component of the search space. Absorb technique is useful for synthetic models with rectangular shape. However, real reservoirs have irregular shapes and different regions, which are desirable to place the wells. Therefore, we first need to find a mathematical formula that represents these type of reservoir boundary constraints.
Let us assume that h_i vector contains heel coordinates and t_i vector contains toe coordinates of the ith well. Now, let us define feasible regions for heel and toe points of the ith well as R_h^i and R_t^i, respectively. After defining feasible regions, reservoir boundary constraint can be formulated as in Eq. (3.11).

$$c_R : \begin{cases} h_i \in R_i^h \\ t_i \in R_i^t, \quad \forall i = 1, 2, ..., N_w \end{cases} \tag{3.11}$$

We can approximate the regions R_h^i and R_t^i by the set of polynomial functions, p_i^{h,n_h} and p_i^{t,n_t} where n_h and n_t are the number of polynomials. We can rewrite Eq. (3.11) by assuming $p_i^{h,n_h}(h_i) \leq 0$ or $p_i^{t,n_t}(t_i) \leq 0$ as in Eq. (3.12).

$$c_R : \begin{cases} p_i^{h,n_h}(h_i) \leq 0, \quad n_h = 1, 2, ..., N_{h,i} \\ p_i^{t,n_t}(t_i) \leq 0, \quad n_t = 1, 2, ..., N_{t,i} \\ \forall i = 1, 2, ..., N_w \end{cases} \tag{3.12}$$

After defining Eq. (3.12), we can apply different methods to deal with the constraints. In this work, we have implemented penalty method to handle the constraint violations. We will give more information about the penalty function that we have implemented in the following sections.

3.3.2 Practical constraints

Well length constraint. The well length constraint is one of the most important constraints in well placement optimization. If the well length constraint is not provided, the optimization will result in a too short or too long well that does not represent the real case scenarios. Therefore, well length constraint needs to be incorporated in to the optimizer.
The length of the ith well is calculated by the following formula:

$$l_w^i = \sqrt{(h_x^i - t_x^i)^2 + (h_y^i - t_y^i)^2 + (h_z^i - t_z^i)^2} \tag{3.13}$$

Let us now define the minimum and maximum length of the ith well as l_{min}^i and l_{max}^i, respectively. After defining the well length limits, well length constraint can be written as:

$$c_L : \begin{cases} l_w^i \leq l_{max}^i \\ l_w^i \geq l_{min}^i \end{cases} \tag{3.14}$$

The above expression can be written in an equivalent form as below:

$$c_L : \begin{cases} (l_w^i)^2 \leq (l_{max}^i)^2 \\ -(l_w^i)^2 \leq -(l_{min}^i)^2 \end{cases} \tag{3.15}$$

Let us substitute $(l_w^i)^2$ with L_i for convenience and rewrite the equation in the following form:

$$c_L : \begin{cases} L_i - (l_{max}^i)^2 \leq 0 \\ (l_{min}^i)^2 - L_i \leq 0 \end{cases} \tag{3.16}$$

As shown in the equation above, L_i is a second degree polynomial and it means that c_L is a set of nonlinear constraints.

Inter-well distance constraint. It is important to take the well spacing limitations into account while optimizing the locations of wells in a reservoir. If the wells are too close, a phenomena known as "well interference" may occur in the reservoir which can has negative effects on the performance of the wells. To avoid such scenarios, we have implemented inter-well distance constraint.
Let us now calculate the distance between two wells in three dimensional space. Note that, the same formula can be applied to many wells by grouping them as pairs. Let us assume well paths as two line segments and find the general formula for calculating the distance between two line segments in three dimensional space. Let the endpoints of the first well be ω_1^t and ω_1^h and the endpoints of the second well be ω_2^t and ω_2^h. Then the line segments can be parameterized by the following formula:

$$\begin{aligned} \omega_1(\lambda_1) &= (1-\lambda_1)\omega_1^t + \lambda_1\omega_1^h, \quad \lambda_1 \in [0,1] \\ \omega_2(\lambda_2) &= (1-\lambda_2)\omega_2^t + \lambda_2\omega_2^h, \quad \lambda_2 \in [0,1] \end{aligned} \tag{3.17}$$

The squared distance between two points on the line segments is calculated as below:

$$\begin{aligned} R(\lambda_1, \lambda_2) =& |\omega_1(\lambda_1) - \omega_2(\lambda_2)|^2 \\ &= a\lambda_1^2 - 2b\lambda_1\lambda_2 + c\lambda_2^2 + 2d\lambda_1 - 2e\lambda_2 + f \end{aligned} \tag{3.18}$$

where

$$a = (\omega_1^h - \omega_1^t)\cdot(\omega_1^h - \omega_1^t),\ b = (\omega_1^h - \omega_1^t)\cdot(\omega_2^h - \omega_2^t),\ c = (\omega_2^h - \omega_2^t)\cdot(\omega_2^h - \omega_2^t),$$
$$d = (\omega_1^h - \omega_1^t)\cdot(\omega_1^t - \omega_2^t),\ e = (\omega_2^h - \omega_2^t)\cdot(\omega_1^t - \omega_2^t),\ f = (\omega_1^t - \omega_2^t)\cdot(\omega_1^t - \omega_2^t)$$

From the definitions above, one can easily observe that :

$$|(\omega_1^h - \omega_1^t) \times (\omega_2^h - \omega_2^t)|^2 = ac - b^2 \geq 0 \tag{3.19}$$

Let us now define $\triangle = ac - b^2$. Depending on the value of $\triangle$, two conditions may exist:

$$\triangle = \begin{cases} ac - b^2 > 0, & \text{Segments are non-parallel,} \\ ac - b^2 = 0, & \text{Segments are parallel.} \end{cases} \tag{3.20}$$

If the cross product of the vectors in Eq. (3.19) is zero, then the segments are parallel and the graph of $R(\lambda_1, \lambda_2)$ is a parabolic cylinder. In the second case, where $\triangle > 0$, segments are non-parallel and the graph of $R(\lambda_1, \lambda_2)$ is a paraboloid.
To find the minimum distance between two line segments, we need to minimize the function $R(\lambda_1, \lambda_2)$ over the unit square $[0, 1]^2$. Since $R(\lambda_1, \lambda_2)$ is a continuous differentiable function, the minimum point is located either at a point on the boundary of the square or inside the square (only if $\triangle > 0$). In the latter case, the gradient defined by Eq. (3.21)

$$\nabla R(\lambda_1, \lambda_2) = 2(a\lambda_1 - b\lambda_2 + d, -b\lambda_1 + c\lambda_2 - e) \tag{3.21}$$

becomes zero only when $\lambda_1^* = (be - cd)/\triangle$ and $\lambda_2^* = (ae - bd)/\triangle$. If $(\lambda_1^*, \lambda_2^*) \in [0, 1]^2$, then the minimum of $R(\lambda_1, \lambda_2)$ is located at point $(\lambda_1^*, \lambda_2^*)$. Otherwise, the minimum is located at the boundary of the square. The boundary of the square is defined by four corners $(0, 0)$, $(1, 0)$, $(0, 1)$, $(1, 1)$ and four edge points $(\lambda_1^0, 0)$, $(\lambda_1^1, 0)$, $(0, \lambda_2^0)$, $(0, \lambda_2^1)$. If we solve $\nabla R(\lambda_1, \lambda_2) = 0$ for λ_1 and λ_2, we can find that $\lambda_1^0 = -d/a$, $\lambda_1^1 = (b - d)/a$, $\lambda_2^0 = e/c$ and $\lambda_2^1 = (b + e)/c$.
FieldOpt framework uses a simple algorithm to calculate all 9 critical points, evaluate R at that points and select the point that gives the minimum squared distance. The algorithm also includes parallel segments. After finding the minimum distance between wells, inter-well distance constraint can be formulated for any number of wells as below:

$$c_D : \{ \quad R(\lambda_{1,i}^*, \lambda_{2,j}^*) \geq d_{min}^2, \ \forall i, j = 1, 2, ..., N_w. \ (i \neq j) \tag{3.22}$$

or in another form:

$$c_D : \{ \quad d_{min}^2 - R(\lambda_{1,i}^*, \lambda_{2,j}^*) \leq 0, \ \forall i, j = 1, 2, ..., N_w. \ (i \neq j) \tag{3.23}$$

where
d_{min} represents the minimum distance provided as a constraint and $R(\lambda_{1,i}^*, \lambda_{2,j}^*)$ represents the minimum squared distance between i^{th} and j^{th} wells.

3.4 Optimization algorithm

Particle Swarm Optimization is introduced by Kennedy and Eberhart in 1995 [6]. The algorithm uses a group of particles to search the solution space of an objective function. Each particle represents a solution vector in this space and the algorithm updates this solution vector in each iteration in order to find the optimal solution. In the beginning, each particle is assigned a random velocity and position vector in the search space. Each particle has a memory of the previous best value of the objective function and corresponding previous best position vector. In addition, every particle in the swarm knows the global best value of the objective function among all the particles and corresponding global best position vector. During optimization process, each particle moves stochastically towards the previous and global best position of the particle (Figure 3.2). Particles tend to move towards better positions in each iteration and the process repeats until all the particles converge to an optimal solution.

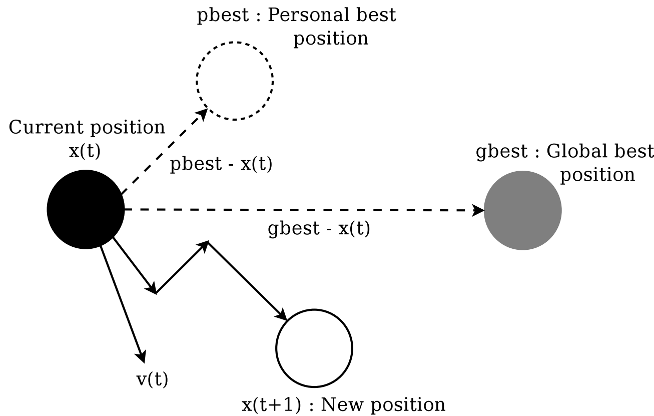


Figure 3.2: Movement of a particle.

In recent years, different kinds of particle swarm optimization have been developed based on the type of optimization problem. For example, Binary PSO for solving discrete problems, Multi-objective PSO for solving several optimization problems simultaneously and other methods. We will only focus on basic particle swarm optimization in this thesis. Basic particle swarm optimization has two main variants, known as global best particle swarm optimization (gbest) and local best particle swarm optimization (lbest).
These methods use different neighborhood topologies, which have an effect on the performance of the algorithm. Neighborhood topology is the grouping of particles into small sub-groups. Particles only exchange information with the particles that are located in their neighborhood. In global best particle swarm optimization, particles exchange information with all the other particles in the swarm and this neighborhood topology is called "star" topology. However, in local best particle swarm optimization, particles only know the global best position inside their neighborhood instead of the whole swarm. Local best particle swarm optimization can be used with different neighborhood topologies. We will only focus on the "random" and "ring" neighborhood topologies that are used in our implementation.

3.4.1 Global Best PSO

Global best version of particle swarm optimization has faster information transfer through the search space due to the effective communication and large inter-connectivity between all the particles in the swarm. Therefore, global best particle swarm optimization converges faster compared with local best particle swarm optimization. However, global best particle swarm optimization has the possibility to converge to a local optimum. Therefore, global best version can sometimes give inaccurate solutions compared with local best particle swarm optimization.
In the global best particle swarm optimization, each particle $i \in [1, 2, ..., n]$ is identified by a position, $x_i(k)$ and a velocity vector, $v_i(k)$ at iteration k. Also at iteration k, each particle i also remembers the best individual position, $p_i^{best}(k)$ that it has visited since the

first iteration and the global best position among all particles, $g^{best}(k)$.
In each iteration, the personal best position of each particle i is updated based on the following formula (considering maximization problem):

$$p_i^{best}(k+1) = \begin{cases} p_i^{best}(k) & if \quad f(x_i(k+1)) < f(p_i^{best}(k)) \\ x_i(k+1) & if \quad f(x_i(k+1)) > f(p_i^{best}(k)) \end{cases} \tag{3.24}$$

where
i shows the particle index and $f(x_i(k))$ shows the objective function value of particle i at position x and iteration k.
The global best position can be updated either on the basis of the individual best positions or current positions of all particles in each iteration.

$$g^{best}(k) = arg\,max\ f(p_i^{best}(k)) \quad or \quad g^{best}(k) = arg\,max\ f(x_i(k)) \tag{3.25}$$

During search process, each particle updates its velocity based on the individual and global best positions. The velocity update equation for each particle is shown below:

$$v_i(k+1) = v_i(k) + c_1 r_1(k)(p_i^{best}(k) - x_i(k)) + c_2 r_2(k)(g^{best}(k) - x_i(k)) \tag{3.26}$$

The equation presented above is used in the standard version of particle swarm optimization. This form lacks the mechanism to control the velocity, which determines the exploration and exploitation abilities of the algorithm.
Shi and Eberhart [21] introduced a new concept called inertia weight(w) to overcome this issue. The velocity update equation was modified as below in order to include the inertia weight:

$$v_i(k+1) = \omega v_i(k) + c_1 r_1(k)(p_i^{best}(k) - x_i(k)) + c_2 r_2(k)(g^{best}(k) - x_i(k)) \tag{3.27}$$

where
c_1 and c_2 are acceleration coefficients or trust parameters, $r_1(k)$ and $r_2(k)$ are random numbers taken from uniform distribution between [0,1] at iteration k.
All the particles fly through the search space by adjusting their positions, which depend on their velocities. The position of each particle at iteration $k+1$ is calculated based on the following formula:

$$x_i(k+1) = x_i(k) + v_i(k+1) \tag{3.28}$$

The formulas (3.27) and (3.28) are used to update the particle velocities and positions during optimization process. However, one needs to use different formulas for initializing the particle positions and corresponding velocities at first iteration. This is usually done by setting all particle velocities to zero and by using a uniform distribution for particle positions. Initial particle positions can affect the performance of the algorithm significantly, since they denote how much of the search space is covered. If the initial positions are not distributed properly, the algorithm will have difficulties to find the optimal solution if the solution is far away from the covered area. Therefore, the initial positions of particles are distributed uniformly between the maximum and minimum ranges of the search space. The initialization of each particle is given by the following formula:

$$x_i(0) = x_i^{min} + r_i(x_i^{max} - x_i^{min}) \tag{3.29}$$

where
x_i^{min} and x_i^{max} represent minimum and maximum ranges of the search space and r_i is the random number taken from uniform distribution between [0,1].
Note that, initial personal best positions are also set equal to the initial positions as shown in the formula below:

$$p_i^{best}(0) = x_i(0) \tag{3.30}$$

The flowchart in Figure 3.3 shows the algorithmic steps of global best particle swarm optimization. As shown in the flowchart, algorithm starts by setting the parameters, such as inertia weight, acceleration coefficients and maximum number of iterations.

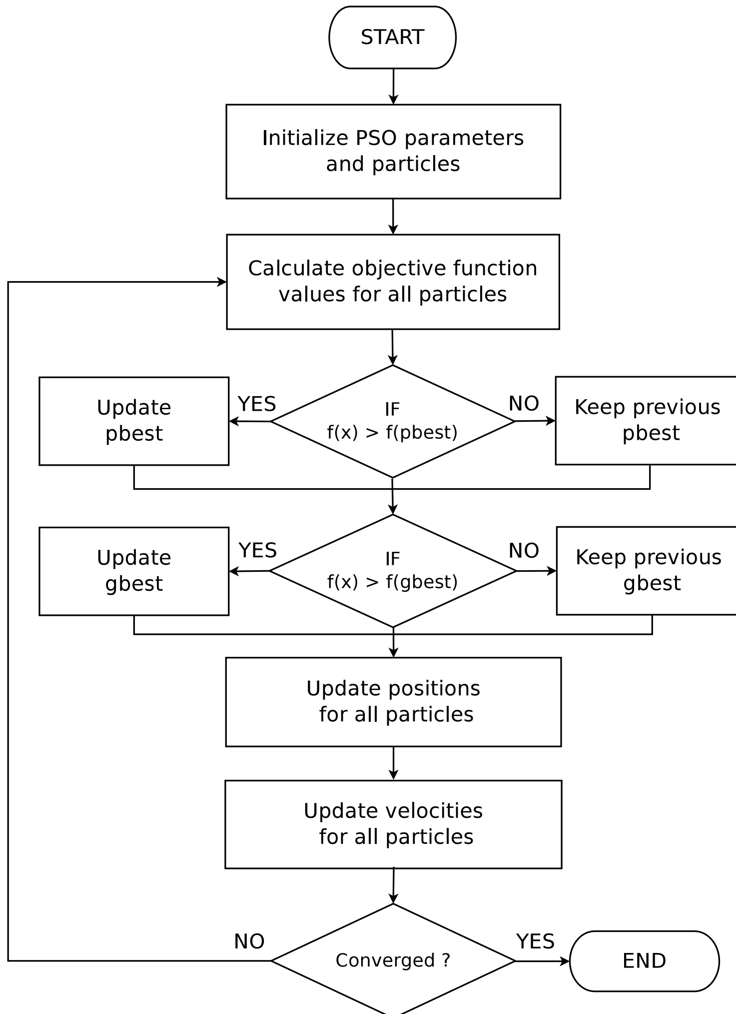


Figure 3.3: Global best PSO (maximization problem)

After setting the algorithm parameters, all the particles are distributed randomly inside the

search space and the velocity of each particle is set to zero. Objective function values for all particles are calculated based on the initial positions. Initial personal best positions are set as same as initial positions and global best position is updated based on the objective function values. After calculating personal and global best positions for all particles, velocity of each particle is calculated and particle positions are updated based on the velocities. In next iterations, personal and global best cases are compared with the previous values and updated accordingly. After updating personal and global best positions, the process continues again with updating velocities and positions. This process repeats until a convergence criteria such as maximum number of iterations is reached.

3.4.2 Local Best PSO

Local best particle swarm optimization uses the same procedure as global best particle swarm optimization. The only difference is in the velocity update equation (Eq.3.27) with different social components for each algorithm. The social component in global best particle swarm optimization represents the whole swarm and there is only one unique global best position for each particle in the swarm. However, in local best particle swarm optimization, local best positions are used for each sub-group instead of the global best position. Therefore, each sub-group has their own global best position. The velocity update equation for local best particle swarm optimization is modified as below:

$$v_i(k+1) = \omega v_i(k) + c_1 r_1(k)(p_i^{best}(k) - x_i(k)) + c_2 r_2(k)(l_i^{best}(k) - x_i(k)) \quad (3.31)$$

where
$l_i^{best}(k)$ represents the local best position for the ith particle at iteration k.
In local best particle swarm optimization, neighborhoods overlap and the speed of information exchange reduces. Therefore, local best particle swarm optimization converges slowly. However, it is more diverse and less prone to get stuck in local optimums compared with global best particle swarm optimization.
In next section, we describe the different neighborhood topologies that are used in our implementation of local and global best particle swarm optimization.

3.4.3 Neighborhood topologies

In particle swarm optimization, each particle is assigned to some neighborhood, which is a group of particles in the entire swarm. However, in global best version of the particle swarm optimization, the whole swarm is considered as a group or neighborhood because all the particles communicate with all the rest. The performance of the algorithm depends on the topology used and the neighborhood size. In general, for large neighborhoods, the algorithm converges faster but it does not guarantee an optimal solution. However, small neighborhoods can take a while to converge to an optimal solution, but the quality of solution is better because it is less prone to get stuck in local optimums.
Note that, neighborhood topologies are usually defined by the particle index, and not by the particle location. For global best particle swarm optimization, the neighborhood of a particle with index $i \in \{1, ..., n_p\}$ consists of all the particles in the swarm:

$$N_i = \{1, ..., n_p\} \quad (3.32)$$

For local best topology, considering a neighborhood size of l, with $l > 1$, the neighborhood of a particle with index $i \in \{1, ..., n_p\}$ consists of all the particles whose index are in the set:

$$N_i = \{i - l, ..., i, ..., i + l\} \tag{3.33}$$

We assume that particle indices wrap around. For example, we replace -1 with $n_p - 1$ or -3 with $n_p - 3$, etc.
We will now describe the most common topologies that are used in global and local versions of the particle swarm optimization. Global best particle swarm optimization uses so called "star" topology, while local best version of the algorithm uses "ring" neighborhood topology.

Star topology. In star neighborhood topology, each particle communicates with all the other particles. Therefore, star neighborhood topology has greater connectivity and interaction between particles compared with other topologies. Star neighborhood topology enables fast convergence of the algorithm. However, the solution may not always show the optimal solution because it can get stuck in the local optimums. The star neighborhood topology for a group of six particles is illustrated in Figure 3.4.

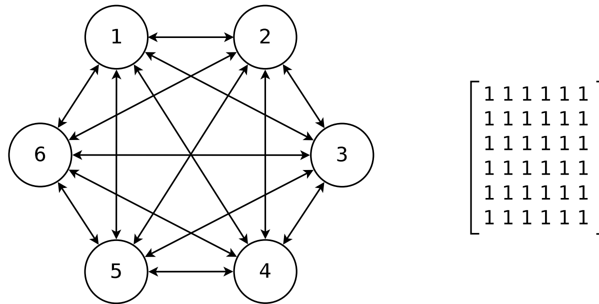


Figure 3.4: Star or gbest neighborhood topology

The matrix on the right of the figure represents communication matrix. Communication matrix represents the relationship between particles. If there is communication between particle i and particle j, the relevant value is set to one in the communication matrix. Otherwise, it is set to zero.

Ring topology. In ring topology, particles only communicate with their immediate neighbors as shown in Figure 3.5.
For example, the particle with index $i = 1$ only communicates particles $i = 2$ and $i = 6$. This is also illustrated in the communication matrix on the right of the figure where the 1st, 2nd and 6th columns of 1st row are set to one and others are set to zero.
In general, for ring topology, the neighborhood of the particle with index i can be found based on the formula given below:

$$N_i = \{i - 1mod(N_p), i, i + 1mod(N_p)\} \tag{3.34}$$

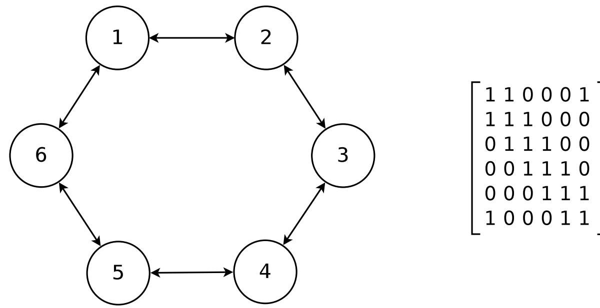


Figure 3.5: Ring or lbest neighborhood topology

where
N_i represents the neighbors for particle i and N_p shows the number of particles.

3.4.4 Parameters

We now describe the parameters that are used in our implementation, which can have a strong influence on the performance of particle swarm optimization. These parameters are swarm size, number of iterations, acceleration coefficients and inertia weight. Note that, depending on the problem, these parameters can have strong or no influence at all.

Swarm size. Swarm size S is the number of particles in the swarm. A big swarm size covers more area inside search space and increases the exploration abilities of the algorithm. Therefore, a good solution may be obtained in a reasonable amount of time with less iterations compared with small swarm size. However, a large swarm size may increase the computational time per iteration. There is still a lack of theoretical analysis about how a good swarm size should be defined. In some versions of the algorithm, swarm size is automatically calculated depending on the dimension D:

$$S = 10 + \lfloor 2\sqrt{D} \rfloor \tag{3.35}$$

However, the formula presented above give results that is far from the optimal swarm size. Therefore, the swarm size is usually set manually depending on the problem.

Number of iterations. The maximum number of iterations also depends on the complexity of the optimization problem. A large number of iterations may guarantee an optimal solution, but it may also lead to an additional computation if the algorithm has already converged to an optimal solution. On the other hand, too few iterations may result in a premature convergence.
In our implementation, we use the maximum number of iterations as stopping criteria, since we do not know the optimal value. We set the maximum number of iterations high in order to avoid the premature convergence and to ensure that a good solution is obtained.

Acceleration coefficients. Acceleration coefficients, c_1 and c_2 adjusts the stochastic influence of cognitive and social components, respectively. The cognitive component helps particles to move towards their personal best positions achieved in previous iterations while social component helps particles to move towards global best position for all particles. If c_1 is set to high compared with c_2, the particles will be strongly affected by their personal best positions. On the other hand, if c_2 is set to high compared with c_1, all the particles will tend to move towards global best which may result in a premature convergence. Accelaration coefficients are normally set manually such that :

$$c_1 + c_2 = 4 \tag{3.36}$$

condition is satisfied. Setting $c_1 = c_2 = 2$ is normally considered as a good choice because particles are attracted towards the average of their personal best positions and global best position.

Inertia weight. Inertia weight was first applied by Shi and Eberhart [21] in order to adjust the influence of previous velocities in the velocity update equation. If the particle velocities are too high during optimization, particles will not be able to move back towards an optimum point and swarm will diverge. On the other hand, if the velocities are too low during optimization, the particles will only focus on some local regions (exploitation) and they will not explore the search space efficiently. Inertia weight controls the exploration and exploitation abilities of the algorithm by adjusting velocities. Inertia weight can be implemented either as a fixed or dynamically changing parameter. Dynamic inertia weight strategy was found to perform better than fixed inertia weight strategy. We have applied both these strategies in our implementation. We have implemented a linear decreasing inertia for dynamic inertia weight strategy. Linear decreasing inertia weight is calculated based on the following formula:

$$\omega(k) = \omega_0 - \frac{(\omega_0 - \omega_1)k}{K} \tag{3.37}$$

where
k and K shows the current and maximum number of iterations, respectively, ω_0 is the initial inertia weight and ω_1 is the final inertia weight, with $0 \leq \omega_1 \leq \omega_0$.
Linear decreasing inertia weight increases the performance of the algorithm by maintaining the balance between exploration and exploitation. High values of inertia weight helps the swarm to explore the search space in the beginning and it is decreased over time which changes the swarm behavior from exploration to exploitation.

3.5 Constraint handling methods

Particular Swarm Optimization algorithm is originally designed to solve unconstrained optimization problems. Therefore, the original design lacks the mechanism to handle constraints which is similar to the evolutionary algorithms. Most of the constraint handling techniques that are used in particle swarm optimization are adopted from evolutionary algorithms.

3.5.1 Penalty method

Penalty method is one of the most popular approaches to deal with the constraints in particle swarm optimization. In general, a penalty function is formulated as in Eq. (3.38).

$$f_p(x) = f(x) + P(x), \quad x \in R^n \tag{3.38}$$

where $f(x)$ is the objective function and $P(x)$ is the penalty factor. Note that, penalty factor is only applied when the constraints are violated. If all the constraints are satisfied, no penalty will be applied (Eq. (3.39)).

$$P(x) = \begin{cases} 0, & \text{if x is feasible} \\ p > 0, & \text{if x is infeasible} \end{cases} \tag{3.39}$$

In our implementation, we will use the penalty function that is shown in Eq. (3.40). This function was found to exhibit promising results with particle swarm optimization [22]. Note that, we have changed the plus sign in original function to minus, in order to apply it to the maximization problem.

$$f_p(x) = f(x) - c(k)H(x), \quad x \in R^n \tag{3.40}$$

where $c(k)$ is dynamically modified penalty coefficient and k is the iteration number. $H(x)$ is penalty factor that is defined as in Eq. (3.41).

$$H(x) = \sum_{i=1}^{n} \theta(q_i(x)) q_i(x)^{\gamma(q_i(x))} \tag{3.41}$$

where $q_i(x) = max\{0, c_i(x)\}$, $i = 1, 2, ..., n$, $\theta(q_i(x))$ is a multi-stage assignment function and $\gamma(q_i(x))$ is the power of the penalty function. $\theta(q_i(x))$, $\gamma(q_i(x))$ and $c(k)$ depend on the problem. We used the same functions as suggested in [22]. These functions apply different penalties on the objective function based on the extent of the constraint violation (Eq. (3.42)).

$$\begin{gathered} \gamma(q_i(x)) = \begin{cases} 1, & \text{if} \quad q_i(x) < 1 \\ 2, & \text{otherwise} \end{cases} \\ \theta(x) = \begin{cases} 10, & \text{if} \quad q_i(x) < 0.001 \\ 20, & \text{if} \quad q_i(x) < 0.1 \\ 100, & \text{if} \quad q_i(x) < 1 \\ 300, & \text{otherwise} \end{cases} \end{gathered} \tag{3.42}$$

We have selected $c(k) = \sqrt{k}$ for the penalty coefficient. Note that, penalty factor increases in each iteration.

3.6 Optimization Framework

The optimization algorithm has been implemented using FieldOpt optimization framework. The optimization framework includes a variety of optimization algorithms, such as

asynchronous parallel pattern search (APPS), generating set search (GSS) and compass search algorithm. The framework also provides flexibility by allowing users to implement their custom algorithms. Therefore, we have written an implementation of the PSO algorithm that can be used as hybrid with other algorithms, and also run as a straightforward PSO algorithm. ECLIPSE, the Automatic Differentiation General Purpose Research Simulator (AD-GPRS) and Flow can be used as reservoir simulator. The coupling is handled by a simulator interface inside FieldOpt. FieldOpt's MPI-based implementation also enables to run simulations in parallel, by running the software on a computational cluster. In next chapter, we will give more information about the structure of the optimization framework.

Chapter 4

Implementation

In this section, we will first give detailed information about FieldOpt optimization framework and then describe how we have implemented and integrated particle swarm optimization algorithm inside FieldOpt optimization framework. We will use UML class diagrams to explain important classes and methods. Note that, we will only show simplified diagrams with most important attributes, methods and inheritances. Also note that, in our class diagrams when a class inherits from an abstract class, it implicitly implements all of its virtual methods.

4.1 FieldOpt

In order to add new functionality to FieldOpt, it is important to understand the structure and working principle of the program. The program runs on linux terminal and requires full paths to the simulation data file, grid file, output folder, driver file and execution scripts as input. Runner type is also specified by user. For example, if we want to use serial runner with ECLIPSE reservoir simulator, we should run FieldOpt from terminal as:

```
./FieldOpt /path/to/the/driver file/ /path/to/the/output folder/ -r serial -g /path/to/the/grid file/ -e execution_scripts/csh_eclrun.sh -s /path/to/the/data file/
```

The program uses boost libraries to parse these inputs from terminal as strings and uses them to run optimization and to read and write the files in specified paths. Driver file is the most important input to the program, since the optimizer, optimizer parameters, simulator type, optimization variables are all specified in this file. Driver file contents are read in the settings section of the program, and model, optimization and simulation objects are created according to the relevant sections specified in the driver file. These objects are required by different classes inside the program. After reading driver file, the control is passed to the user specified runner which implements abstract runner class. Therefore, abstract runner class is called first inside runner. Abstract runner class includes methods to initialize the settings, model, simulator, objective function, base case, optimizer, bookkeeper and logger

before starting optimization. The order in which abstract class initializes is important. For example, base case should be initialized before optimizer because optimizer requires base case for initialization. After initializing all the required sections, the control is passed to the optimizer interface. Optimizer interface passes the control to the required optimizer. Optimizer gets the required variables from model and creates new cases based on the specific algorithm used by that optimizer. These cases are then passed to the runner, and runner passes the program control to the simulator interface in order to evaluate the cases. The simulator interface calls the specified simulator class. The simulator class stores the original simulation data file contents and overwrites new cases. After modifying data file, simulator is called to evaluate the new model. Results from simulation is read and passed to the runner. Runner updates the objective function values of the cases and passes the evaluated cases to the optimizer interface. This process repeats until the end of the optimization process. The optimization loop is illustrated in Figure 4.1.

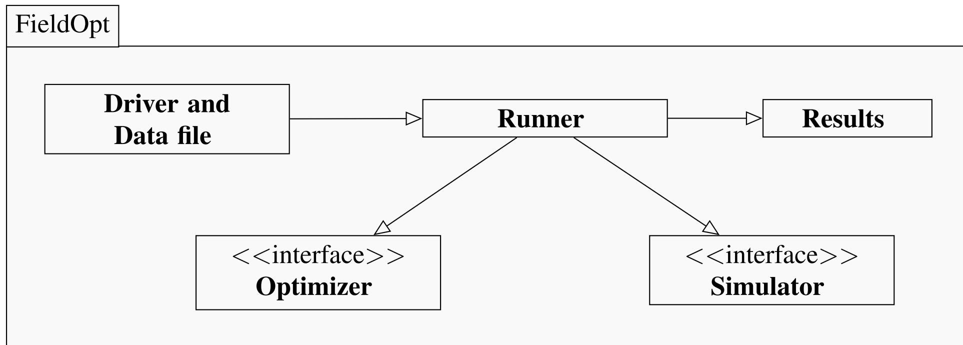


Figure 4.1: FieldOpt working principle.

FieldOpt uses bookkeeper class to avoid evaluating the same cases. This is done by specifying a bookkeeper tolerance parameter inside driver file. When the case is received by the runner, runner calls this class in order to check the status of the case. The bookkeeper class calculates the difference between current and previous cases. If the difference is less than the specified tolerance, the case is said to be evaluated before, otherwise the case is a new case and it will be passed to the simulator for evaluation. Finally, all the results are logged by logger class during optimization and results are stored inside output folder.

4.1.1 Driver file

FieldOpt uses driver files in json format which is easy for users to edit and also easy for machines to parse. Driver files contain four main parts, which are explained below:

- Global section: includes global definitions, such as the name of output file and bookkeeper tolerance
- Optimizer section: Algorithm specific parameters such as maximum number of iterations, number of particles, etc. are defined in this section. Components for objective function evaluation and constraints are also defined in this section.

- Simulator section: Reservoir simulator and the name of the script to run this simulator is declared in this section.
- Model section: All the wells, well controls, well type, control times are specified here. Optimization variables are declared by setting the IsVariable property to true.

All these sections are read in the settings section inside FieldOpt and relevant objects are created.

4.1.2 Runner

Runner is the main driving engine in optimization process. FieldOpt uses different runners such as serial runner, mpi runner, synchronous mpi runner and abstract runner. These runners are defined by users in program options while running the software. Mpi runners are used to run optimization in parallel such as in linux clusters. All the runners implement abstract runner class, which initializes the most important sections that are needed by all runners before starting optimization process.

4.1.3 Optimizer interface

All the optimizers inside FieldOpt, except APPS, implement optimizer interface. This interface helps the algorithms to access constraints and other important classes and it holds data members and functions that are important for all of the optimizers during optimization process. The case handler class is one of the most important class for the interface, since all the cases are handled by this class. The case handler class stores the list of recently evaluated cases and all evaluated cases. Evaluated cases are counted in each iteration and it can be used as a stopping criteria by specifying the maximum number of evaluations in driver file. The case handler class also adds the new cases to the queue list. This cases are dequeued one by one in order to be passed to the runner. The runner then calls simulator interface to run simulations and evaluate objective function value.

4.1.4 Simulator interface

Simulator interface is responsible for running simulations and reading the results. The inteface includes three classes for Flow, ADGPRS and ECLIPSE reservoir simulators. These classes implement the abstract `Simulator` class. Different sections of simulation data file contents are stored inside this interface. Modification to these sections during optimization process is handled by driver file writers. After writing new simulation data file, reservoir simulations are run by invoking a system call with the new data file as input parameter to the simulators installed in the system. FieldOpt uses `ert` libraries to read the results from finished simulations. After reading the results, the objective function value for a particular case is calculated.

4.1.5 Case class

All the variables from perturbed model are hold inside Case class. Case class includes methods to set the value of objective function and variable values for a particular case.

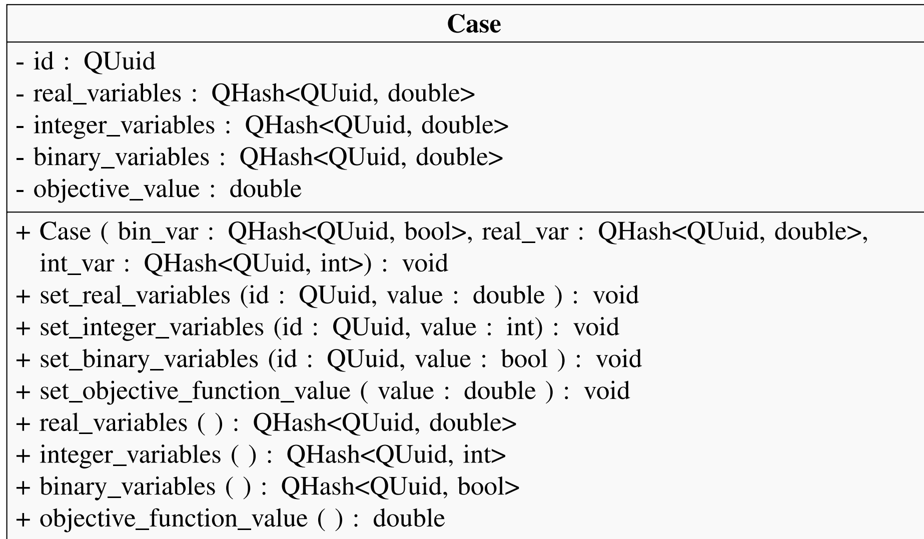


Figure 4.2: Class diagram for Case class

All the case objects and variables are assigned a unique id. The simple class diagram of the case class is illustrated in Figure 4.2.
This class is assumed as a simplified form of the model class. It is simpler than model class, since it only stores variable values in simple data structures and occupies less space in memory. This enhancement makes the case class useful to work with the optimizers. This class is one of the most important classes in FieldOpt, since all the optimizers work with case objects. Although the case class is very simple and efficient, simulator interface only uses model objects to run the simulations. Therefore, the model is updated by applying new cases before running simulation.

4.2 PSO Integration

The main algorithm was implemented inside `PSO` class. `PSO` class is created as a sub-class of the `Optimizer` class. `Optimizer` class includes important attributes and methods that are needed for all optimization algorithms. These methods are defined as virtual methods inside `Optimizer` class. Virtual methods need to have specific implementations for the classes derived from `Optimizer` class. Therefore, we have reimplemented virtual methods `iterate()` and `isFinished()` for `PSO` class.

We have written a separate driver file for particle swarm optimization that specifies the algorithm specific parameters. We have then modified optimizer class in the settings section of the program in order to read these parameters. Another addition was made in abstract runner class since all the optimizers are initialized inside this class. We have added PSO

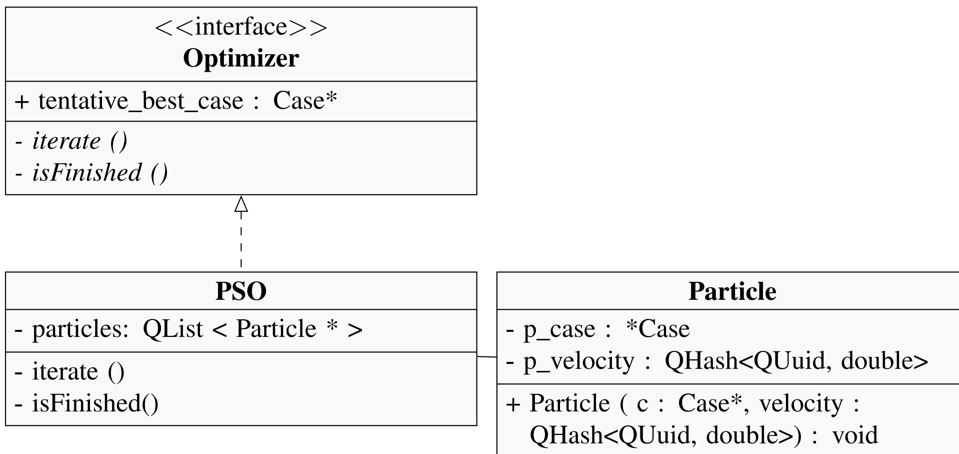


Figure 4.3: Main classes and class relationship.

to the list of optimizers inside initialize optimizer section of the abstract runner class. We also implemented new constraint classes to use with PSO class. These classes are then added to the list of constraints inside the constructor of constraint handler class. Since the PSO class needs direct access to the constraint class, we have added a new method to constraint handler class in order to get the constraints because optimizers can only access to the constraint handler class.
While implementing particle swarm optimization algorithm inside FieldOpt, we also needed to create an additional particle class, since the optimizers work with case objects and case objects only hold the variable values. Therefore, we assigned velocities for each case by creating a particle class that holds both the cases and associated velocities. Case variables and velocities are in the vector format and each component is identified by a unique id. We have assigned the same ids for both the cases and associated velocities, so the value of each component can be updated according to the velocity value of that specific component. The cases are decoupled from particles after updating case variables in order to apply it to the model and run simulations because particle objects can not be applied to the model object. After running simulation and updating objective function values, the cases and corresponding velocities are coupled again in order to update the case variables.

4.3 Particle class

In particle swarm optimization, each particle holds variables and corresponding velocities. As mentioned in previous section, variables are hold inside case class. During optimization, FieldOpt instantiates objects of this class and applies it to the model. The model is then passed to the simulator for objective function evaluation.
The particle class was created in order to keep variables values from case class and corresponding velocities. Note that, the velocities are only needed to update variable values. After updating variables, cases and velocities are decoupled in order to apply the cases to

the model and run simulation.
Class diagram of the `Particle` class is shown in Figure 4.4.

Particle
- particle_case : *Case - particle_velocity : QHash<QUuid, double>
+ Particle (c : Case*, velocity : QHash<QUuid, double>) : void + get_case (): *Case + set_particle_velocity (id : QUuid , value : double) : void + get_particle_velocity () : QHash<QUuid, double>

Figure 4.4: Particle class.

As shown in the diagram above, particle class includes three important methods which are needed by PSO class:

- `get_case()`: Since the particle and assigned case point to the same place in memory, we can get corresponding cases from particle objects using this method
- `set_particle_velocity()`: This method is needed to set the corresponding velocity value of the components to zero in case these components leave the search space.
- `get_particle_velocity()`: This method returns the velocity of a particle object in the vector format.

4.4 PSO class

The main algorithm is implemented inside `PSO` class. `PSO` class contains methods to initialize particles, to find global and personal best cases for all particles, methods to check constraint violations and apply penalties on objective function values and also methods for creating neighborhoods for each particle. `PSO` class needs to instantiate objects of `Particle` class in order to create particles. We will give detailed information about `Particle` class in the next section.
A simplified diagram of `PSO` class showing the most important methods and main attributes is shown in Figure 4.5.

The main methods of `PSO` class are listed below:

- `iterate()`: This method calls the required methods in a stepwise manner during optimization.
- `initialize_cases()`: This method is used to perturb initial cases and velocities based on the number of particles.
- `apply_penalty()`: This method is used to check constraint violations and apply penalties to the objective function if the constraints are violated.

PSO

- id_list : QList<QUuid>
- real_max : QHash<QUuid,double>
- real_min : QHash<QUuid,double>
- max_iter : double

+ PSO (*settings : Settings::Optimizer, *base_case : Case, *variables : Model::Properties::VariablePropertyContainer, *grid : Reservoir::Grid::Grid) : void
- *iterate () : void*
- initialize_cases () : QList < Case *>
- perturb_real_variables (base_variables : QHash < QUuid, double >) : QHash < QUuid, double >
- initialize_velocity (size : int) : QHash < QUuid, double >
- set_particles (particles : QList < Particle * >) : void
- get_particles () : QList < Particle * >
- find_case_velocity (*c : Case) : QHash < QUuid, double >
- update_global_best_case (recent_cases : QList < Case * >) : void
- set_personal_best_cases (pbest_cases : QList < Case * >) : void
- get_personal_best_cases () : QList < Case * >
- update_personal_best_cases () : void
- select_neighborhood_topology () : void
- set_local_best_cases (lbest_cases : QList < Case * >) : void
- create_global_best_case_list () : void
- create_ring_communication_matrix (np : int) : vector <vector< int > >
- update_local_best_cases_ring () : void
- create_random_communication_matrix (np : int) : vector <vector< int > >
- update_local_best_cases_random () : void
- update_particles () : QList< Case * >
- get_local_best_cases () : QList < Case *>
- apply_penalty (cases : QList < Case * >) : void

Figure 4.5: PSO class.

• `update_global_best_case()`: This method is used to find the global best case and update it in each iteration.

• `update_personal_best_cases()`: This method is used to find the personal best case for each particle and update it in each iteration.

• `update_particles()`: This method is used to update the velocities and positions of each particle during optimization.

All the other methods are called inside these main methods and brief description of them will be given while describing these main methods. In the following sections, we will give detailed information about these main methods.

4.4.1 Iterate method

The most important method for PSO class is `iterate()` method because all the critical methods that are important for the optimization process are called inside this method. Algorithm 1 shows the simple working principle of `iterate` method.

Algorithm 1 Iterate method

```
1: if ( iteration = 0 ) then
2:     AddNewCases(initialize_cases())
3: else
4:     apply_penalty(RecentlyEvaluatedCases())
5:     update_personal_best_cases()
6:     update_global_best_case(RecentlyEvaluatedCases())
7:     AddNewCases(update_particles())
8: end if
9: ClearRecentlyEvaluatedCases()
10: iteration++
```

At first iteration, `initialize_cases()` method is called and generated cases are passed to the runner for objective function evaluation. This method is only called at first iteration. In next iteration, evaluated cases are passed to the `apply_penalty()` method for checking constraint violations. Inside this method, the objective function values are updated based on the constraint violations. After updating objective function values, personal and global best cases are updated by calling `update_personal_best_cases()` and `update_global_best_case()` methods. Lastly, `update_particles()` method is called in order to update the cases and associated velocities. New cases are passed to the runner for objective function evaluation and the procedure repeats again until maximum number of iterations is reached.

4.4.2 Initialize cases method

This method is only used at the beginning of the optimization. Inside this method, a list of cases and particles are created based on the number of particles. Particle velocities and variables are initialized by calling `initialize_velocities()` and `perturb_real_variables()` methods. We set initial velocities to zero and apply uniform distribution to all the components of real variables. The perturbed real variables values are assigned to the cases and all of these cases are saved as a vector. Particle velocities and cases are used to create particle objects by sending them to the constructor of Particle class. These particles are put in a vector and saved using `set_particles()` method.

4.4.3 Apply penalty method

This method receives the list of recently evaluated cases and checks if the constraints are satisfied. If the constraints are violated, penalty is applied to the objective function value. Note that, by applying penalties, we reduce the chance of the infeasible particles to be selected as best particle among others, since the penalized objective function value is less

compared with other particles (maximization problem). We have implemented well length, inter-well distance and reservoir boundary constraints inside this method.

4.4.4 Update Global Best Case method

This method receives the list of recently evaluated cases and saves the case that has the best objective function value. This value is saved as `tentative_best_case`. In next iterations, `tentative_best_case` is compared with new cases and updated if there is improvement in objective function value compared with previous iteration. This method

Algorithm 2 Update global best case method

```
 1: if ( iteration = 0 ) then
 2:     best_case=base_case
 3: else
 4:     for ( Case c : RecentlyEvaluatedCases ) do
 5:         if ( mode=Maximize )  then
 6:             if ( c→obj.value > best_case→obj.value ) then
 7:                 best_case=c
 8:             end if
 9:         else
10:             if ( mode=Minimize ) then
11:                 if ( c→obj.value < best_case→obj.value ) then
12:                     best_case=c
13:                 end if
14:             end if
15:         end if
16:     end for
17: end if
```

is designed to work both in maximization and minimization problems. This method only works for gbest particle swarm optimization. In the case of lbest particle swarm optimization, `update_local_best_cases_ring()` and `update_local_best_cases_random()` methods are used. These methods and `update_global_best_case()` use the same principle to update the global best case except that first one only compares the objective function values of the particles that are defined as neighbors. In lbest particle swarm optimization, there are more than one global best case for each particle, since the particles only communicate with their neighbors compared with whole swarm in gbest version. Therefore, we have decided to make a vector of global best case that works for both versions. We created a `create_global_best_case_list()` method to save the global best case as a vector for gbest algorithm. This method creates a vector of global best case, based on the number of particles in each iteration and assigns the same value for all of them. Note that, these values will be different for lbest algorithm.

4.4.5 Update Personal Best Cases method

This method is called after evaluating objective function values of all particles in each iteration. This method also works for both minimization and maximization problems. The method compares the objective function values of each particle with previous iterations and saves this as a vector for all particles. This method uses `set_personal_best_cases()` method in order to save the personal best cases. Saved personal best cases are needed inside `update_particles()` method and they also need to be updated in each iteration. Therefore, we have implemented `get_personal_best_cases()` method to access this personal best cases.

4.4.6 Update Particles method

This method is used to update the velocities and positions of particles that are saved in previous iterations. Inside this method, `select_neighborhood_topology()` method is called first. After then, based on the topology defined in driver file, communication matrix is created using `create_ring_communication_matrix()` or `create_random_communication_matrix()` and local best cases are updated on the basis of the communication matrix by using `update_local_best_cases_ring()` or `update_local_best_cases_random()`.

After updating local best cases, the velocity of each case is found by calling `find_case_velocity()` method. Inside this method, `get_particles()` method is first called in order to get the list of saved particles. Then, `get_case()` method is called for each particle and ids are compared with the input case id. If the id is same, the velocity of that particle is returned by calling `get_particle_velocity()`. After finding current velocities of each evaluated case, velocity and position update equations are applied for all the components.

Chapter 5

Case Study

5.1 Model descriptions

5.1.1 5 spot model

We consider a simple 2D, two phase (oil and water) model. The model does not include any faults or dipping as shown in Figure 5.1. The grid contains 3600 ($60\times60\times1$) grid blocks with uniform size for each of them. The dimensions of the model are 1440 m long, 1440 m wide and 24 m thick. The top of the model is at 1700 m with 170 bar initial

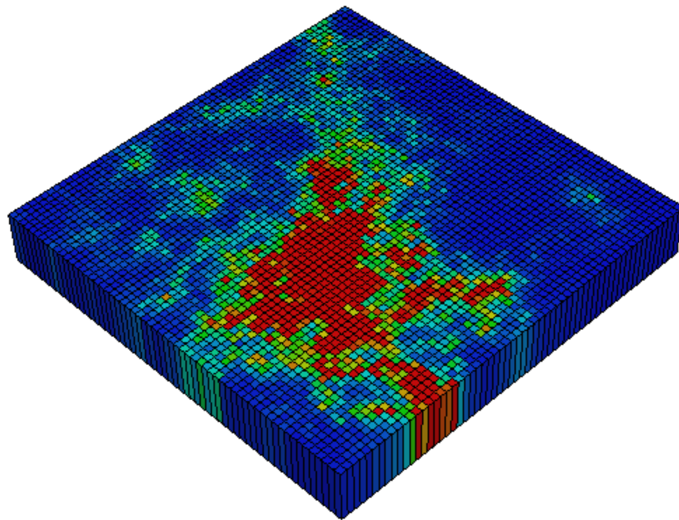
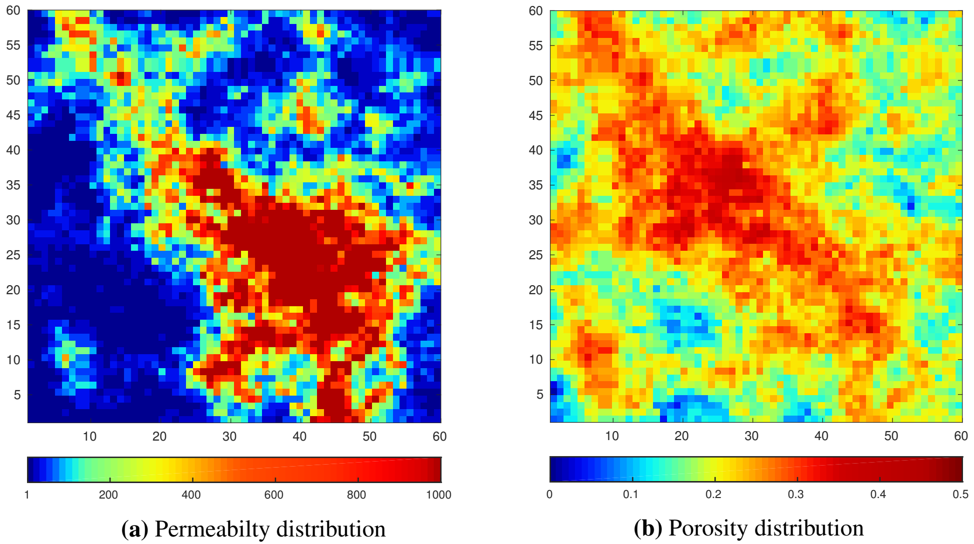


Figure 5.1: 5 Spot Model.



(a) Permeabilty distribution

(b) Porosity distribution

Figure 5.2: Permeability and Porosity distribution

pressure at this point. Initially, the model is fully saturated with oil (no connate water). The permeability and porosity distribution is a cut off layer 21 of the SPE 10 model which are both shown in Figure 5.2. Residual water saturation was 0.15 and capillary pressure was neglected. Relative permeabilities are shown in Figure 5.3.

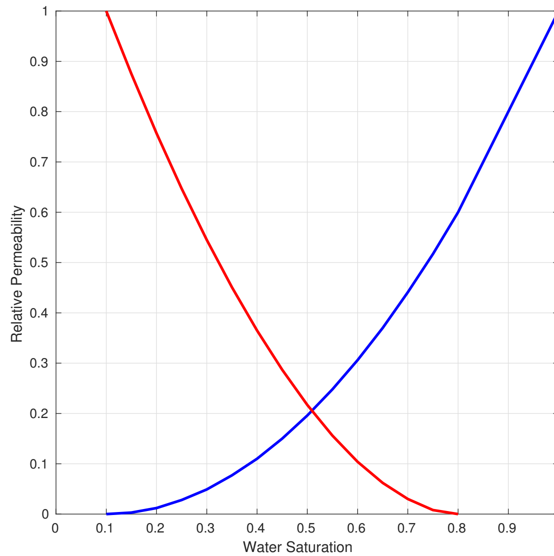


Figure 5.3: Relative Permeabilities.

5.1.2 Olympus model

Olympus synthetic reservoir model was inspired by a virgin oilfield in the North sea and was developed for the purpose of a benchmark study for field development optimization. The reservoir thickness is 50 m and the entire thickness is modeled by 16 layers. The reservoir consists of top and bottom parts which are separated by an impermeable shale layer. The top part of the reservoir contains fluvial channel sands while the bottom part contains alternating layers of coarse, medium and fine sands. Table 5.1 shows the summary of facies properties for different parts of the reservoir.

Table 5.1: Facies properties.

Facies type	Zone	Porosity	Permeability	Net-to-Gross
Channel sand	Top	0.2 - 0.35	400 - 1000 mD	0.8 - 1
Shale	Top and barrier	0.03	1 mD	0
Coarse sand	Bottom	0.2 - 0.3	150 - 400 mD	0.7 - 0.9
Sand	Bottom	0.1 - 0.2	75 - 150 mD	0.75 - 0.95
Fine sand	Bottom	0.05 - 0.1	10 - 50 mD	0.9 - 1

The field is 9 km by 3 km and it is modeled by 341,728 grid cells of which 192,750 are active. The impermeable shale layer that separates the top and bottom parts of the reservoir contains mostly inactive cells. The reservoir is also separated by 6 minor faults into different regions in horizontal direction and it is bounded on one side by a boundary fault (Figure 5.4).

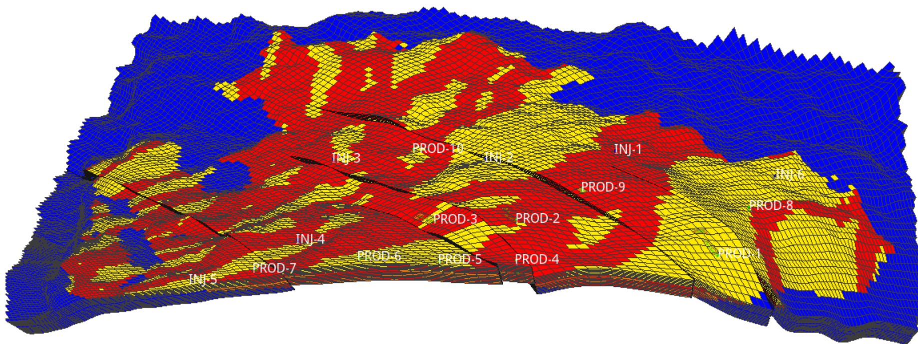


Figure 5.4: Olympus model

Uncertainty was taken into account by performing 50 realizations in which different porosity, permeability, net-to-gross and initial water saturations are generated. The oil-water contact (OWC) was determined to be 2090 m with a hydrostatic pressure of 206 bar and it was considered the same for all realizations. Grid and faults are also kept same for all realizations.

The relative permeability curves are different for each facies. Therefore, initial water saturations are also different for each realization, since facies distribution varies in each

realization. We will only focus on the first realization in this thesis, since we do not consider uncertainty because of high computational demand and time.

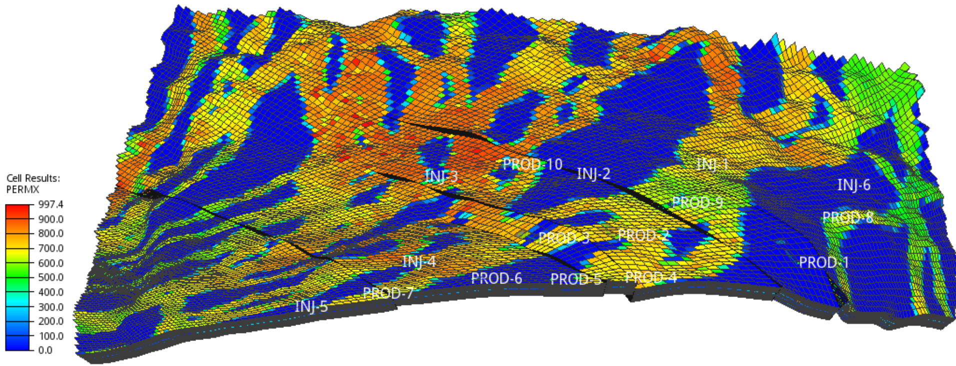


Figure 5.5: Permeability distribution for the first realization.

Figure 5.5 and 5.6 show the permeability and porosity distribution for first realization, respectively.

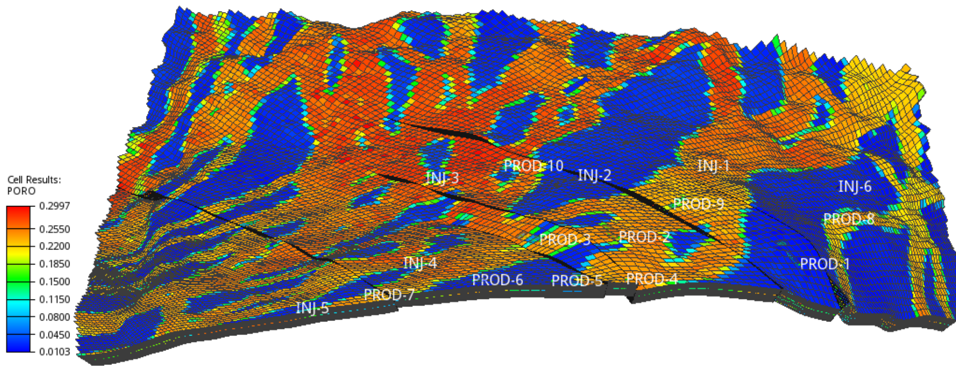


Figure 5.6: Porosity distribution for the first realization

The reference operating strategy for the first realization consists of 10 producers and 6 injectors which are controlled by a pressure constraint.

5.2 Optimization results and discussion

5.2.1 5 spot results

Case 1. In this case, we place four injection wells at the corner of the model and optimize the location of one horizontal well. All of the injectors are controlled with a constant rate of 500 Sm^3/day. The producer is controlled with a target liquid rate of 5000 Sm^3/day and a minimum BHP limit of 120 $bara$. We use only well length and reservoir boundary constraints in this case, since we have only one well. The minimum and maximum well lengths are set to be 200 m and 600 m, respectively. The production time is set for 8 years (2920 days).

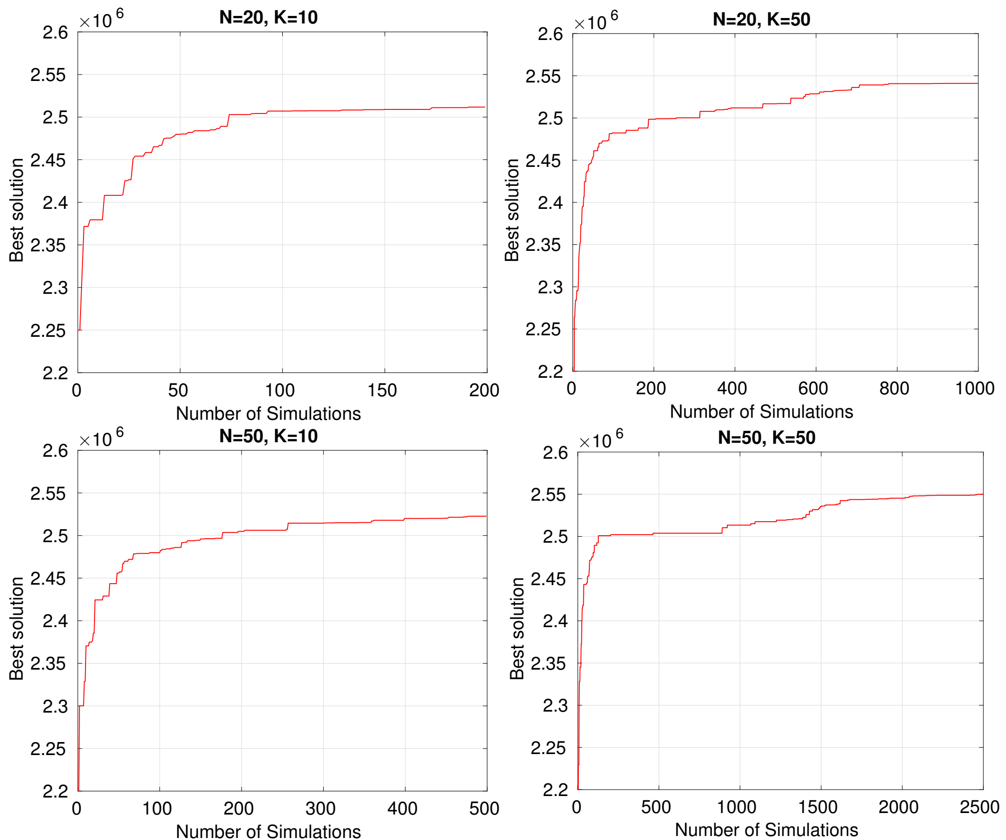


Figure 5.7: Sensitivity analysis for different swarm sizes and maximum number of iterations.

We first perform sensitivity analysis to study the performance of PSO using different swarm sizes and maximum number of iterations. We consider a combination of small and big numbers for both swarm size (N) and maximum number of iterations (K). For this purpose, we use 20 and 50 particles for swarm size, 10 and 50 iterations for maximum

number of iterations. The results show the average of 5 runs for each combination of N and K.
Figure 5.7 shows the comparison of NPV as a function of total number of simulations for each combination. It is evident that increasing the number of iterations from $K = 10$ to $K = 50$ for the same swarm sizes ($N = 20$ or $N = 50$) increases the performance of PSO. More iterations increases the global exploration abilities of the algorithm by helping the particles to update their locations more, therefore increases the chance of obtaining better results. Increasing swarm size from $N = 20$ to $N = 50$ for the same number of iterations ($K = 10$ or $K = 50$) also increases the performance of the algorithm. This may be related to the better coverage area provided by big swarm sizes. We obtain the best results for a swarm size of $N = 50$ and maximum iterations of $K = 50$.
After determining the optimal swarm size and maximum number of iterations, we perform sensitivity analysis on cognitive and social components and also inertia weight strategy. We use three different tunings for this purpose as shown in Table 5.2.

Table 5.2: PSO parameters for different tunings

Parameters	ω_1	ω_0	c_1	c_2	N_p	K
Tune 1	1.2	0	2.8	1.2	50	50
Tune 2	0.721	0.721	1.193	1.193	50	50
Tune 3	1.2	0	1.193	1.193	50	50

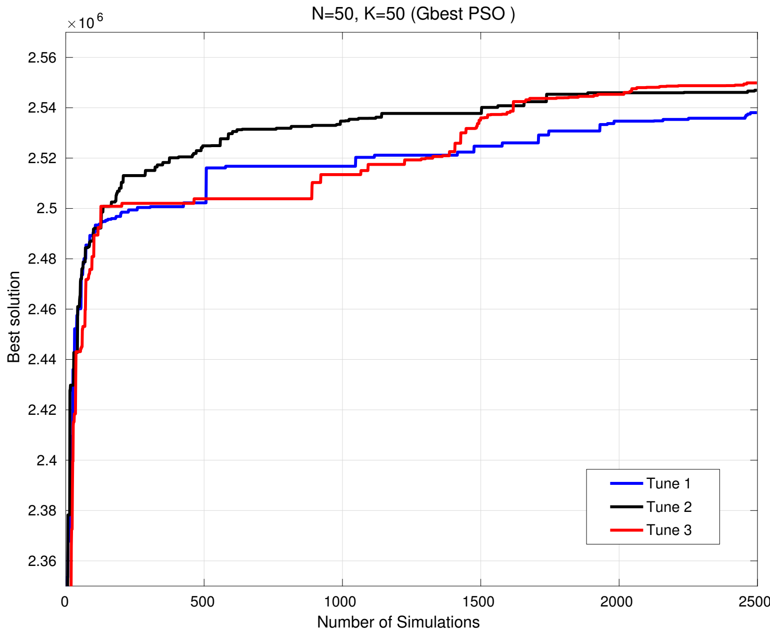


Figure 5.8: Average NPV over five runs vs number of simulations for different tunings

In the first and third tuning, we apply a dynamically changing inertia weight strategy that starts from $\omega = 1.2$ and decreases linearly to $\omega = 0$, while in the second tuning we

apply a static inertia weight strategy with $\omega = 0.721$. We also apply different acceleration coefficients (social and cognitive coefficients) in Tune 1 and 3.
We first compare Tune 1 and 3 in order to determine the effect of social and cognitive components. The results show only the effect of acceleration coefficients since the inertia weight strategy, swarm size and maximum number of iterations are the same. In Tune 1, we set a higher cognitive coefficient $c_1 = 2.8$ than social coefficient $c_2 = 1.2$ and in Tune 3, we set the same number for social and cognitive coefficients $c_1 = c_2 = 1.193$. Results show that setting the social and cognitive components in Tune 3 as equal performs better than Tune 1 (Figure 5.8) in which the social and cognitive components are different. This may be related to particle trajectories in Tune 3. Because when the social and cognitive components are the same, particles move stochastically towards the average of particles' personal and global best positions. On the other hand, setting a higher coefficient for cognitive component helps the particles to move stochastically towards their personal best positions. Although this is true for this specific model, we are not sure for other reservoirs since they have different properties and distribution that yields completely different objective function surfaces.

After comparing the effect of social and cognitive coefficients, we also compare the

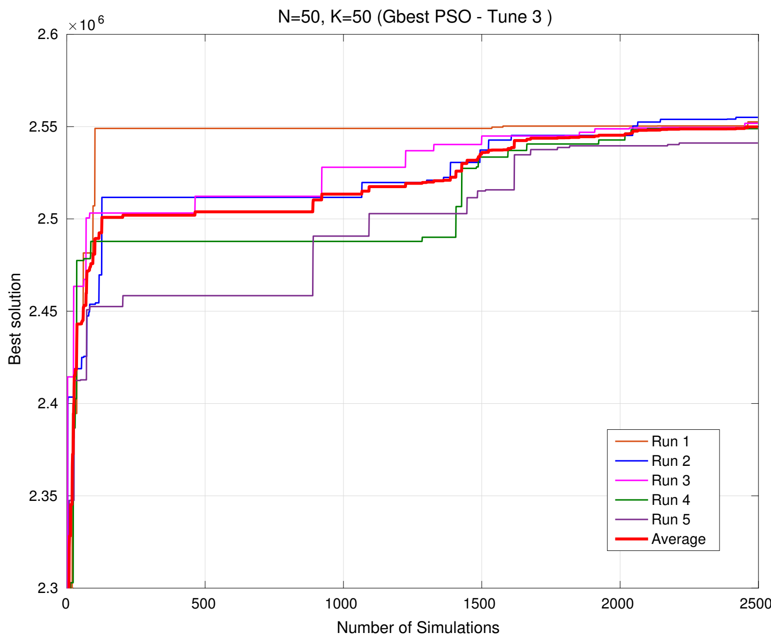


Figure 5.9: Average NPV and NPV of five runs vs number of simulations for Tune 3.

inertia weight strategies. We have implemented a linear decreasing strategy in our implementation, but it can also be used as a static inertia weight strategy by setting the start and end values as equal. We take the best results from the comparison of Tune 1 and 3, and compare it with Tune 2. This shows that, linear decreasing inertia weight strategy in Tune 3 achieves better results compared with static inertia weight strategy in Tune 2 as shown in

Figure 5.8. This is explained by better global exploration abilities of the algorithm in Tune 3. Because, in Tune 3, higher initial values of inertia weight help the particles to update their positions more compared with static inertia weight strategy. Therefore, it explores the search space more efficiently at initial iterations and it is decreased gradually for each iteration in order to increase the exploitation abilities of the algorithm.
In our comparisons discussed above, Tune 3 achieved the best results. Note that, all the tunings show the average results over 5 optimization runs. We present 5 optimization runs and their average for the third tuning in Figure 5.9.
As observed in Figure 5.9, all the runs give similar results with small differences suggest-

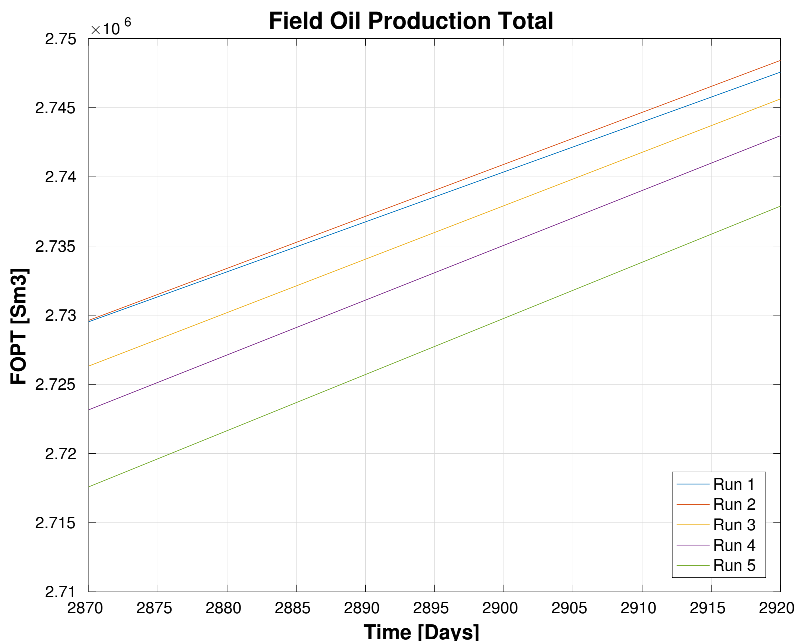


Figure 5.10: Field total oil productions for the first case (scaled).

ing that the algorithm converged to a optimum solution. Among the 5 optimization runs, we have obtained the best results in Run 2 with third tuning. This is also confirmed by total oil and water productions for each run. Total oil and water productions are almost the same for all the runs with small differences. Therefore, we present the scaled results of total oil and water productions to confirm the results.
Figure 5.10 and 5.11 show the scaled results for total oil and water production for 5 optimization runs. These results confirm the results achieved by pso because Run 2 achieves the highest total oil production and the smallest water production, suggesting that oil recovery factor is the highest for Run 2.
Figure 5.12 shows the final well locations and resulting final oil saturations for all 5 optimization runs. The injectors are shown with red circles while the producer is shown with a solid black line. Labels on the wells show their heel coordinates. We optimize the heel and toe coordinates of one horizontal producer. Therefore, we have 6 optimization variables in

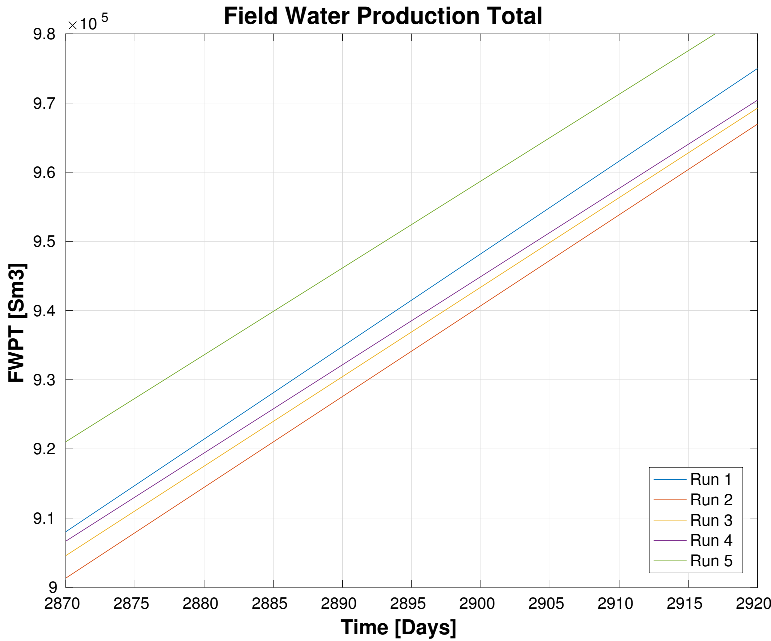
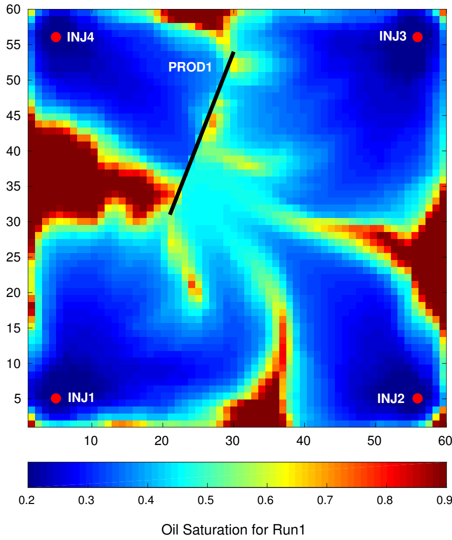


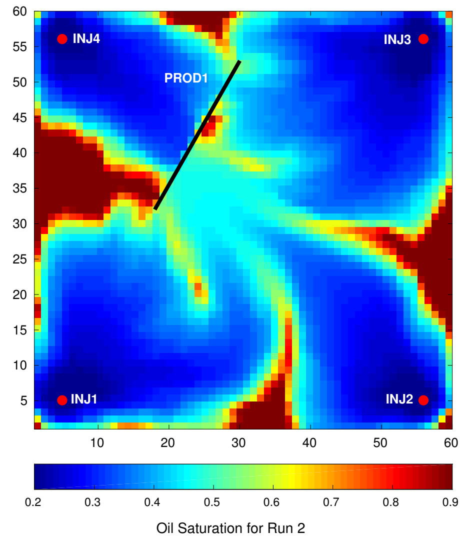
Figure 5.11: Field total water productions for the first case (scaled).

this case.
All the runs converged to the north-east part of the model in this case. This region is a reasonable location, because it is located in low permeability region of the reservoir in order to avoid early breakthrough of water coming from the injectors. Therefore, in low permeability region, we can obtain the highest oil production for a given production time (8 years).
All the final well configurations provide an efficient sweep by providing an equal displacement of oil. Although the final well configurations provide an equal displacement of oil, some of the oil is left in the corners of the reservoir. This may be related either to production scenario (production time or well controls such as BHP or rate) used in this case or reservoir properties. Especially the north, east and west regions where much of the oil is left correspond to very low permeability regions. For the production scenario, we used an upper liquid target limit and minimum bottomhole pressure limit for the producer in order to avoid the early drainage of the oil within the given production time. However, this can also be optimized along with the well location to provide more optimal results but well control optimization is out of the scope of this thesis. Therefore, we have only focused on the well placement optimization part.
In Figure 5.12, we can visually check the results. As can be seen from the figure, Run 2 shows better sweep efficiency both in heel and toe points. Run 1 and 4 show less sweep efficiency near the toe points compared with Run 2, while Run 3 and 5 show less sweep efficiency near the heel points.
Note that, we have implemented "absorb" technique to deal with reservoir boundary con-

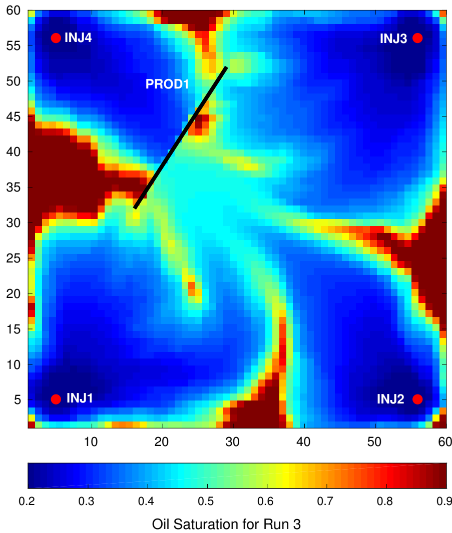
straints and a dynamic penalty method to treat the well length constraints in this case. In the absorb technique, when one of the heel or toe components (x, y, z) of the well leaves the search space, they are set back to the boundaries and corresponding velocity components are set to zero and particles move into the search space in next iterations.
In penalty method, depending on the extent of the constraint violations, penalties are applied to the objective function values in order to avoid that cases being selected as personal and global best positions, which affect the performance of the algorithm.



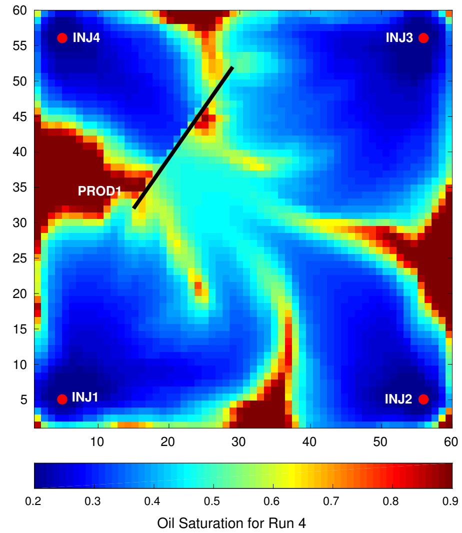
(a) Run 1



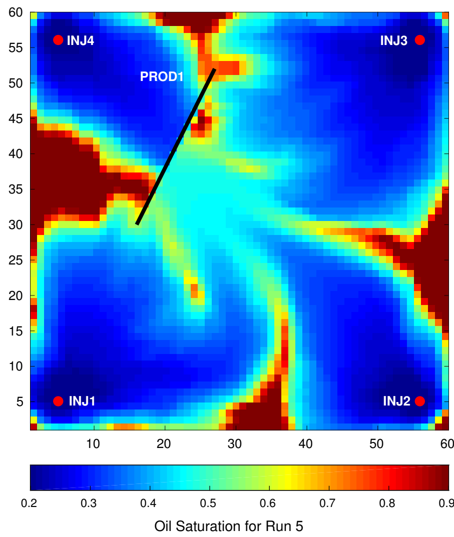
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

Figure 5.12: Oil saturations at 2920 day for different runs in Tune 3.

Case 2. In this case, we optimize the location of one horizontal producer inside a specific region within the reservoir. This region may represent the constraints that are decided

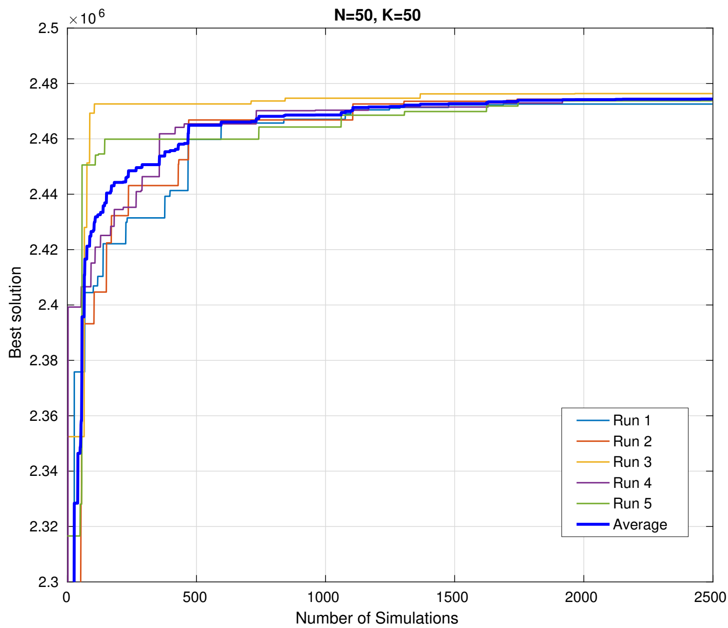


Figure 5.13: Average NPV and NPV of five runs vs number of simulations (Case 2).

during field development phase and imposed on well placement optimization problem. We formulate this region as a constraint by using polynomials. This constraint can be used to incorporate and apply reservoir engineering knowledge to the optimization problem. For example, this region may represent the regions that are bounded by faults or regions with good reservoir properties, which are desirable to place the wells.
The optimization problem in this case is to maximize the NPV by changing the heel and toe coordinates of one horizontal production well, which is subject to reservoir boundary and well length constraints. Although this is similar to the first case, the reservoir boundary constraints in this case are different than box constraints (reservoir boundary) used in the first case. We have used "absorb" technique for box constraints in the first case. However, in this case, we formulate the reservoir boundary constraint using polynomials and penalty method is more relevant for this kind of constraints. We also apply penalty method to treat the well length constraints. The minimum and maximum well lengths are set the same as in the first case. Production time and well controls for both injectors and producer are also set the same as in the first case. For particle swarm optimization, we took the tuning of

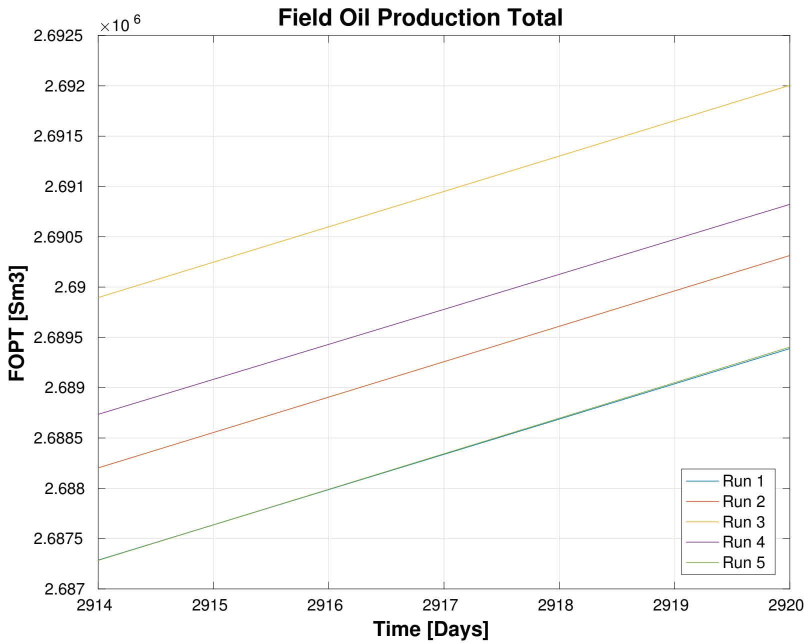


Figure 5.14: Field total oil productions for the second case (scaled).

parameters that gave the best results in the first case, which is Tune 3. Since the particle swarm optimization is a stochastic method, it makes sense to run the algorithm multiple times and take the average results in order to make meaningful comparisons. Therefore, we also run the optimization 5 times and take the average of the runs.
Figure 5.13 shows the improvement of objective function versus number of simulations for 5 optimization runs and their average. All the runs give similar results as we expect. Among the runs, Run 3 gives the best results compared with the others. The average results

are shown with a blue solid line and the average results are very close to those obtained by five runs. We compare the total oil and water productions for all 5 optimization runs in Figure 5.14 and 5.15. It is evident that Run 3 gives the highest total oil production and lowest total water production among other runs. Run 1 and 5 give almost the same total oil production. However, the total water productions are different for two runs (Run 1 gives more total water production than Run 5).

Final well configurations and reservoir boundary constraints are illustrated in Figure 5.16.

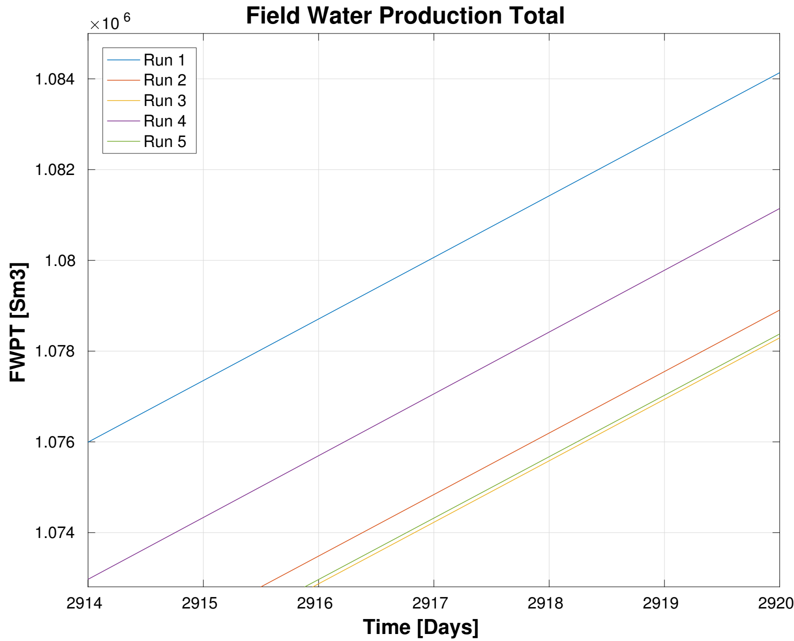
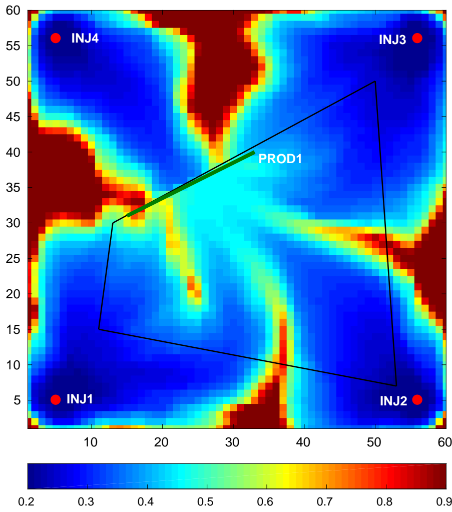


Figure 5.15: Field total water productions for the second case (scaled).

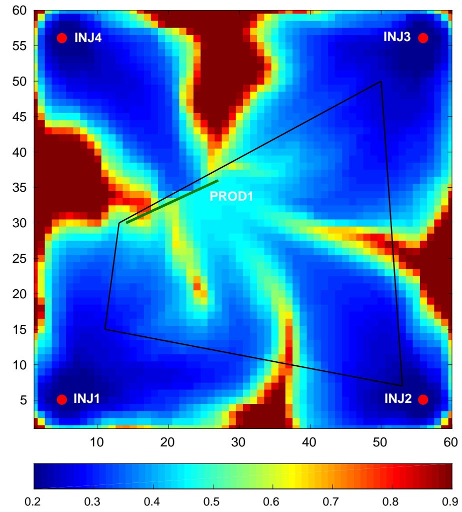
Production well is shown by a solid green line while reservoir boundaries are illustrated by solid black lines. As shown in the Figure 5.16, all the runs converged to similar well locations, which are in the north east part of the model. These results correspond with the results in the first case. Because the optimal location for the model was found to be in the north east part of the model in the first case. Since we use the same production scenario and the same parameters for particle swarm optimization, the expected result should be close to the north east part.

We observe that optimal locations are found to be close to the boundaries in north-east direction. This region is low permeability region and provides the most efficient drainage of reservoir oil such that the water breakthrough time is delayed. We also observe that in this case more oil left at the end of production compared with the first case, especially in the north part. This is related to the boundary constraints implemented in this case, since we limit the search in a specific region, which may be far away from optimal case that provides the highest recovery of the oil. However, the results still represent the optimal case with given constraints and for the specific control parameters. Also note that, in this case,

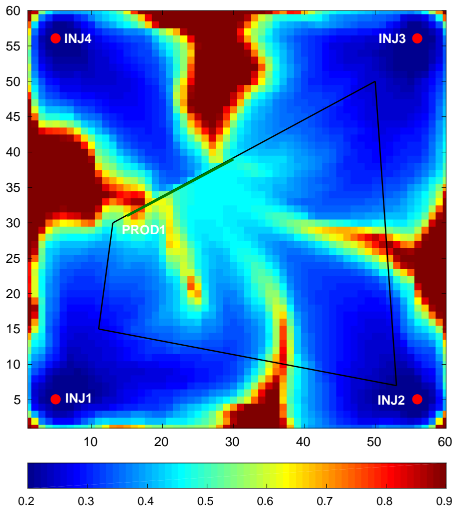
penalty function can also effect the results, since some of simulations are consumed for infeasible cases, which in turn reduces the chance of the algorithm to converge to a better solution within the given maximum number of iterations for the specific swarm size.



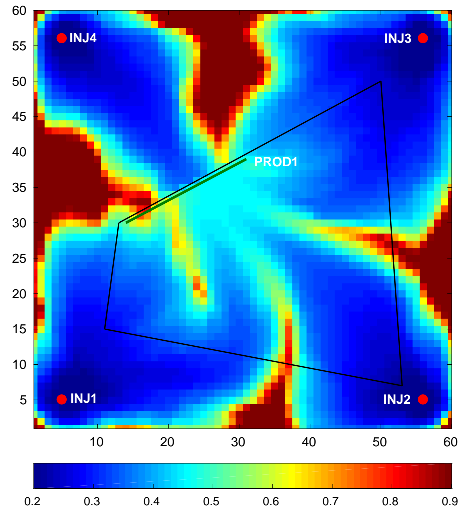
(a) Run 1



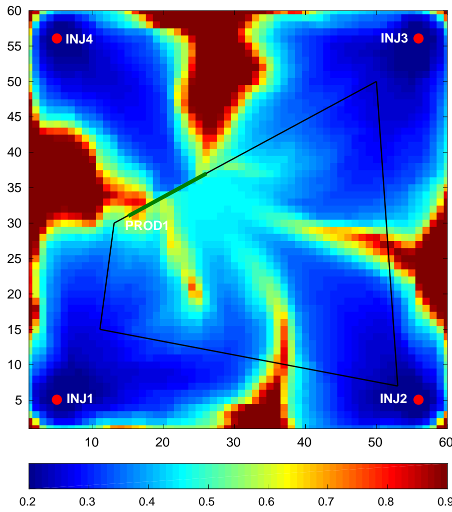
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

Figure 5.16: Oil saturations at 2920 day for different runs.

Case 3. In this case, we apply reservoir boundary, inter-well distance and well length constraints to optimize the heel and toe coordinates of two horizontal production wells. This case is computationally more demanding than the first and second cases because there are twelve optimization variables. We applied inter-well distance constraint to keep a certain distance between wells in order to avoid interference between wells, which may affect the performance negatively. We use a minimum distance of $d_{min} = 200m$ for inter-well distance constraint. Reservoir boundary constraints are similar to those used in the second case. However, in this case, we have changed the boundaries in order to include two wells inside the boundaries. The minimum and maximum well lengths are set as same as in the first and second case. The penalty method is applied to treat all the constraints. We apply the same well controls for producers with a liquid target limit rate of 5000 Sm^3/day and a minimum BHP limit of 120 $bara$.
Figure 5.17 shows the evolution of the NPV of the best solution versus number of the simulations. Each thin curve corresponds to a different optimization run, and the thick blue line depicts the average of best solutions from five runs. In this case, there is a big deviation between the runs and their average. This may be related to the number of constraints used in this case. We observe that more constraints degrade the performance of the algorithm. This might be related the ratio of infeasible and feasible cases because when more constraints are used, this ratio increases and it affects the performance of the algorithm. Note that, we use a constant capital expenditure in the objective function evaluation and it is possible that the algorithm finds two optimal cases with different well lengths, but very similar objective function values. However, in a real case scenario, a longer well costs more than a shorter well with the same production. For example, we have obtained very similar results for Run 1 and 3. These runs have quite similar total oil (Figure 5.18) and water (Figure 5.19) productions, as well as, same recovery factors (Figure 5.20).

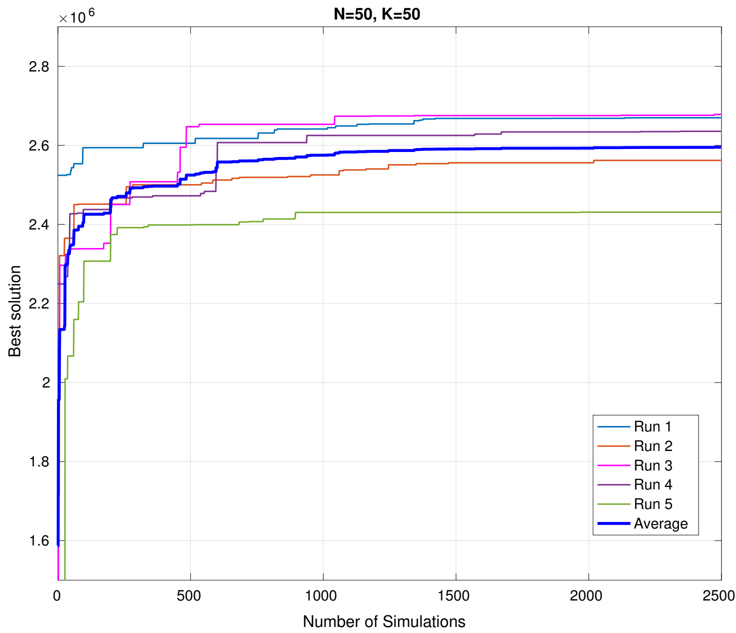


Figure 5.17: Optimization results for the third case.

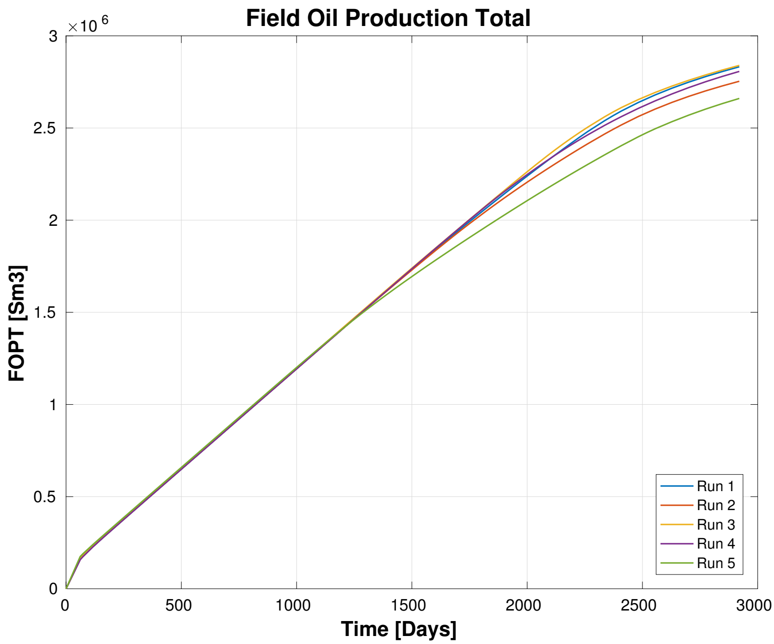


Figure 5.18: Field total oil productions for the third case.

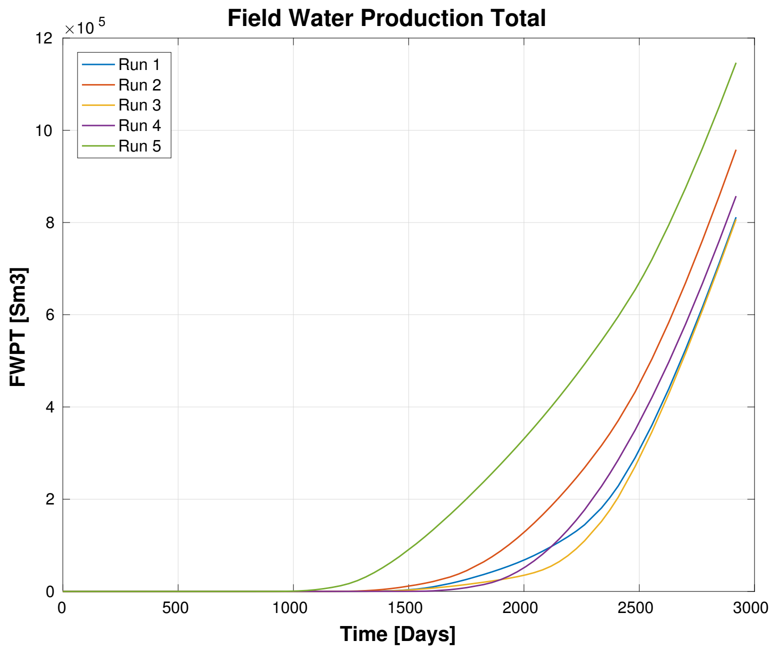


Figure 5.19: Field total water productions for the third case.

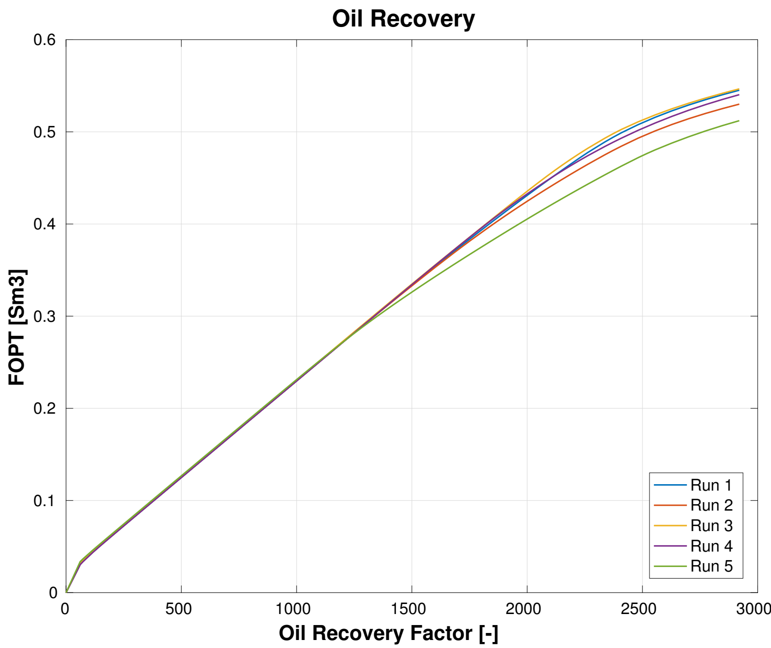
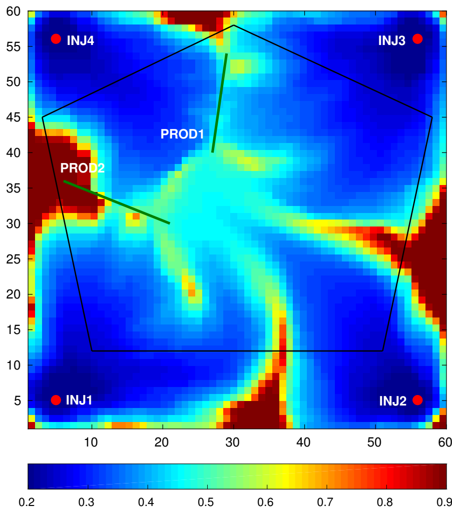
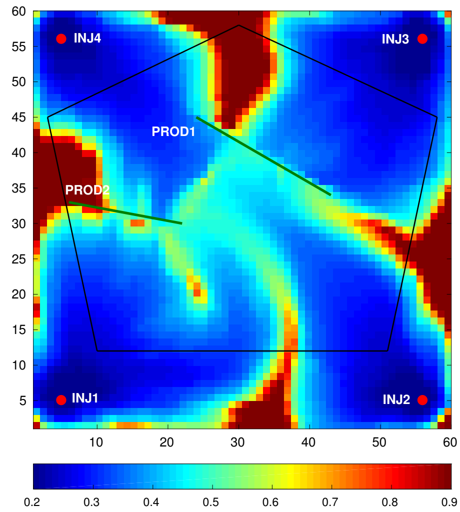


Figure 5.20: Recovery factors for the third case.

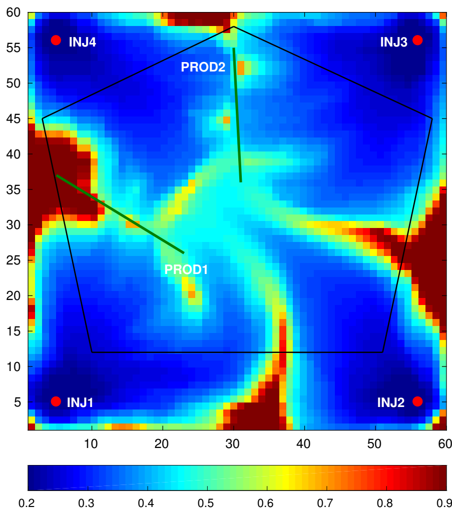
However, the final well configurations corresponding to Run 1 and 2 are different. Figure 5.21 shows the final well configuration for different optimization runs. The resulting oil saturations are shown in the background. The producers are shown with solid green lines. It can be seen from the figure that, in Run 1, the wells are relatively shorter than Run 3 as we have discussed above. Because of the shorter well lengths in Run 1, the final locations are also different than Run 3. Therefore, considering the cost of wells, Run 1 should be taken as optimal case. As in the first and second cases, the algorithm again finds the optimal location in low permeability region in order to avoid early water breakthrough.



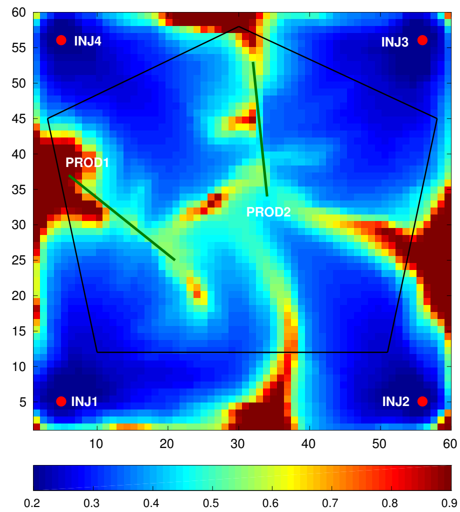
(a) Run 1



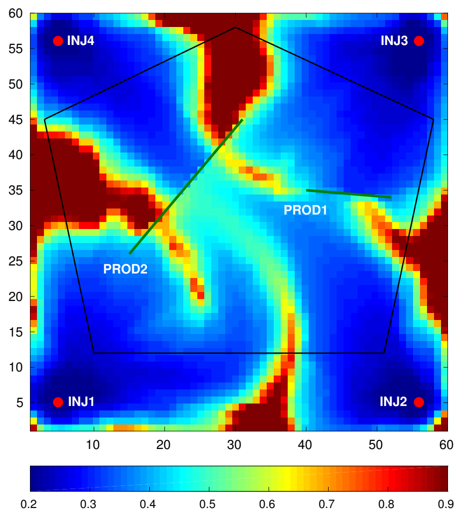
(b) Run 2



(c) Run 3



(d) Run 4



(e) Run 5

Figure 5.21: Oil saturations at 2920 day for different runs.

It can be visually (also from Figure 5.17) observed that relatively less oil is left at the end of the production compared with the first and second cases. This is related to the number of wells used in this case. Although very little oil is left in the reservoir at the end of production, the effect of well controls need to be investigated further, since we are not sure how the specific well control parameters affect the drainage of the wells.

5.2.2 Olympus results

Olympus synthetic model contains small faults which makes the regular well patterns sub-optimal. Therefore, the location of individual wells need to be optimized. Figure 5.22 shows the initial locations of the wells. There are 10 producers and 6 injectors, initially. All the producers and injectors are controlled by a BHP constraint of 175 and 235 bar, respectively.

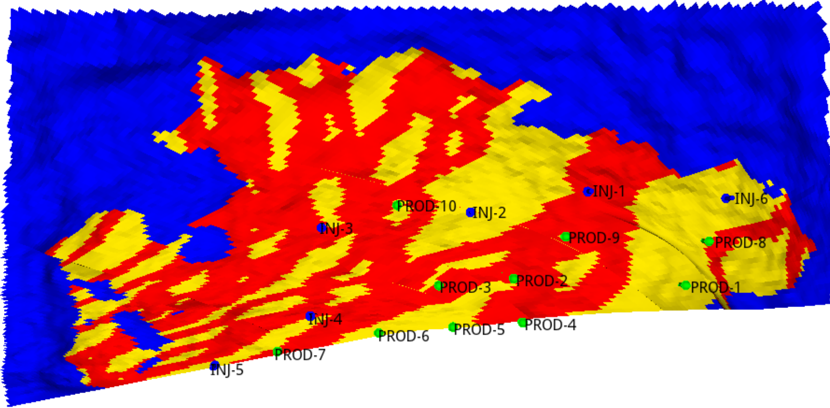


Figure 5.22: Initial well locations (Top view).

We optimize the locations of 5 production wells (PROD1, PROD2, PROD3, PROD4, PROD5) for 5 years, while fixing the well controls and other well locations variables constant during the optimization.

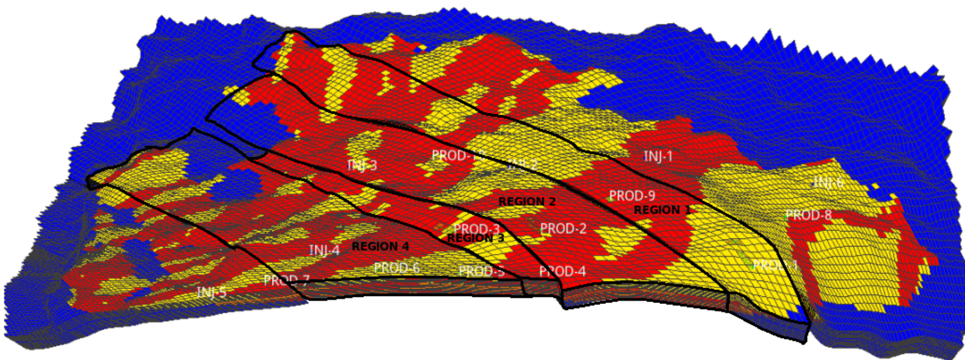


Figure 5.23: Regions.

We divide the reservoir into 4 regions in order to initialize and optimize the well locations inside those regions as shown in Figure 5.23. The regions include both top and bottom

zones of the reservoir and cover most of the area that contains oil. We have used the i, j, k cell indexes to define the regions. We use special functions inside FieldOpt optimization framework to get the real x, y, z coordinates of cells for initializing particle swarm optimization, since the algorithm we have implemented only works with continuous variables. We apply absorb technique when the well location variables are outside the defined regions. For this purpose, we create a list of cells for each region and check it against new well locations in each iteration. If the corresponding cell for new well location is not in the list, we project that location to the nearest bound and set the corresponding velocity component to zero.

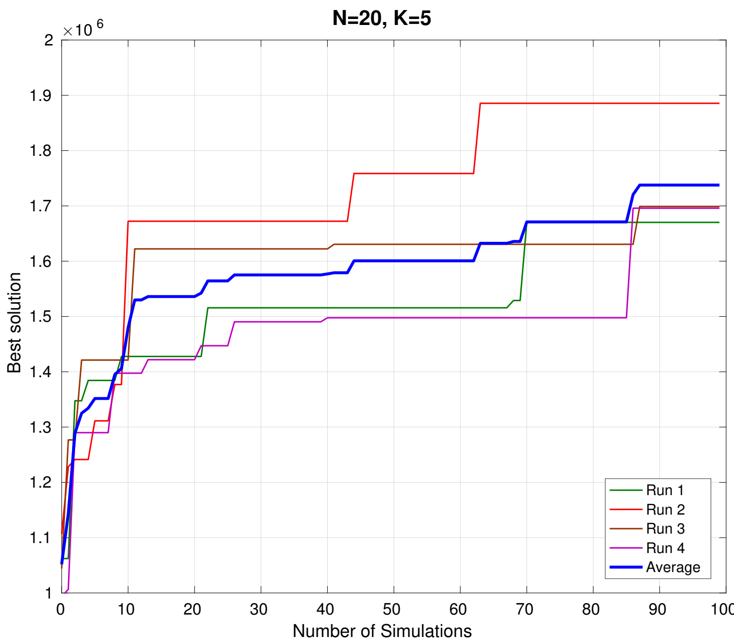


Figure 5.24: Optimization results.

We have defined one region for each well, except the third one. The third region includes wells, PROD3 and PROD 4, while the first, second and fourth regions include PROD1, PROD2 and PROD5, respectively. Therefore, we only apply inter-well distance constraint for the third region between the wells, PROD3 and PROD4. A minimum inter-well distance of 200 m is used in order to avoid the interference problems in the reservoir. Inactive cells are treated such that heel or toe point of a well can not be located inside inactive cells, but the well trajectory can cross the inactive cells. We have defined regions in active cells. However, there is an impermeable shale layer (layer 8) associated with inactive cells, which separates the top and bottom zones of the reservoir. We exclude these cells when we define the regions to solve the inactive cell issues.
In this example, we use a swarm size of 20 and set the maximum number of simulations to 100. Acceleration coefficients are set the same, $c_1 = c_2 = 1.193$, since these numbers

proved to perform better in previous examples. We perform 4 optimization runs and compare them.

Figure 5.24 shows the evolution of the NPV of best solution versus number of simula-

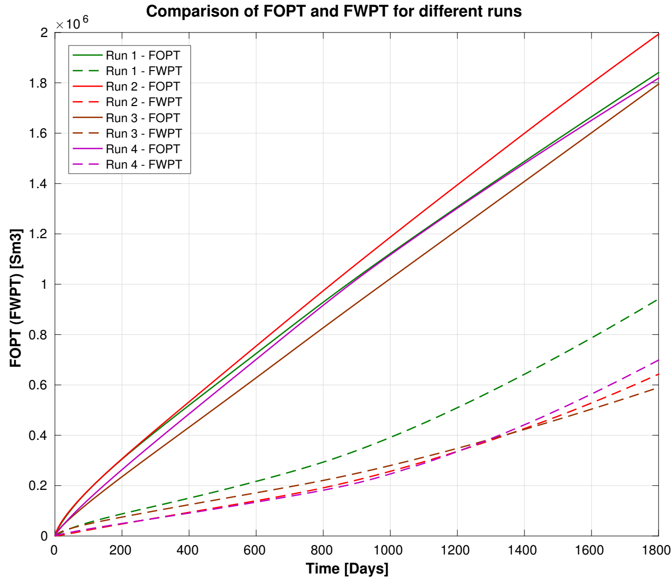


Figure 5.25: Comparison of field total oil and water productions.

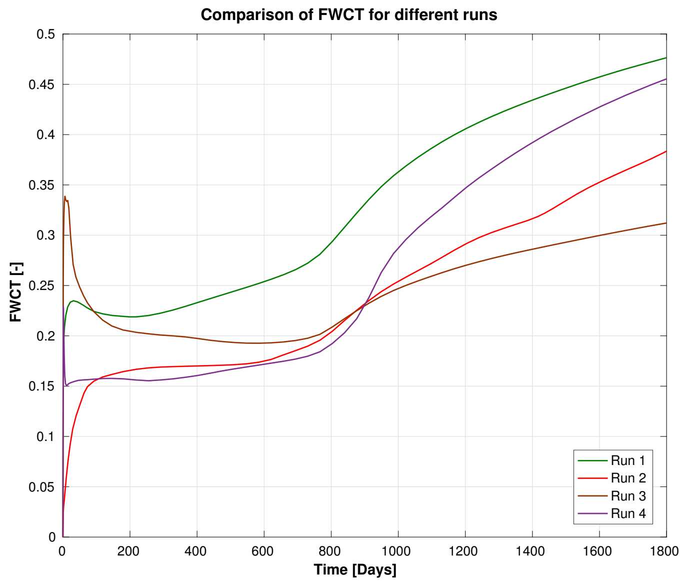
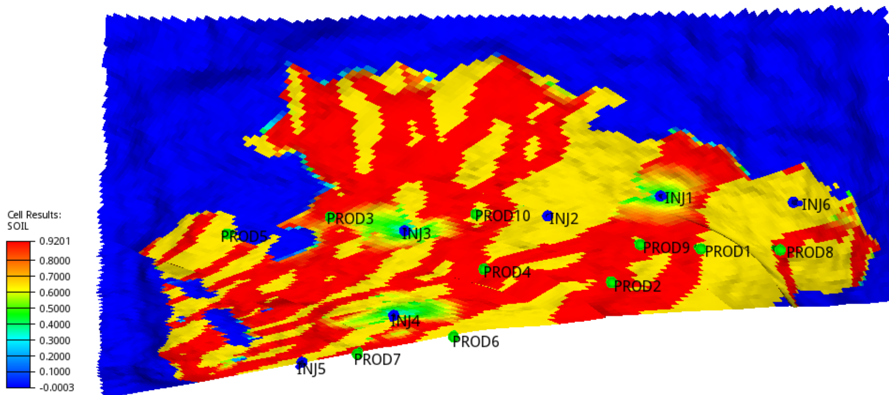
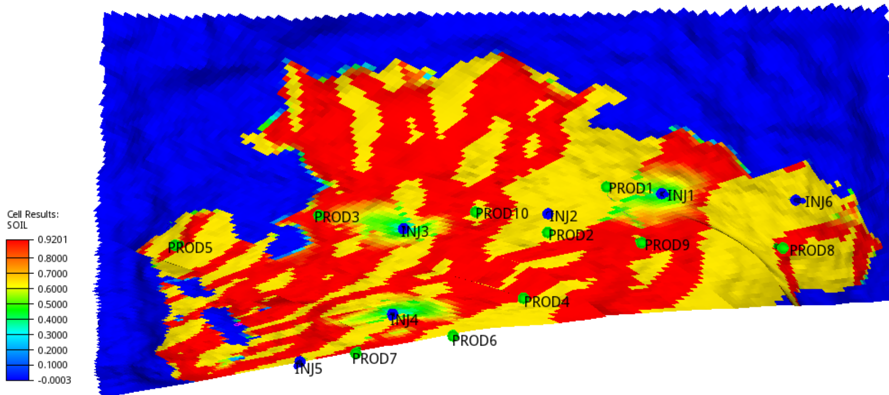


Figure 5.26: Comparison of field water cut for different runs .

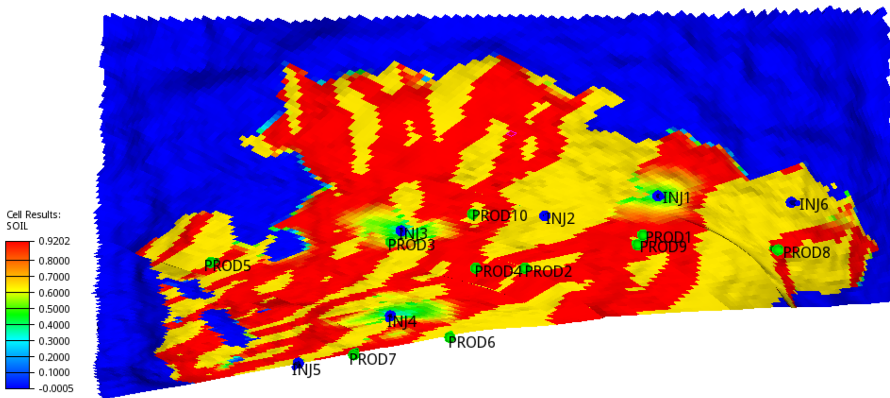
tions. Thin curves represent different optimization runs, while the thick blue line shows the average of 4 optimization runs. We observe that Run 2 performs better than the others and there is a high deviation between runs and their average. This means the algorithm did not converge to an optimal solution with the given parameters. This is related to the swarm size and maximum number of simulations used in this case.
We could not use a bigger number for swarm size and maximum number of iterations, because of the limited time and some issues related to current version of the optimization framework. However, in this case, we show that our implementation of the particle swarm optimization algorithm can be applied to optimize the well locations in real reservoir models. This is also confirmed with optimization results. The optimization results seem to be reasonable in this case.
To analyze the optimization results, we compare field total oil and water productions in Figure 5.25. Although the total water production in Run 3 is less than Run 2, the higher total oil production in Run 2 compensates this and yields the highest NPV. It can also be observed that all the runs have an earlier water-breakthrough time compared with Run 2.



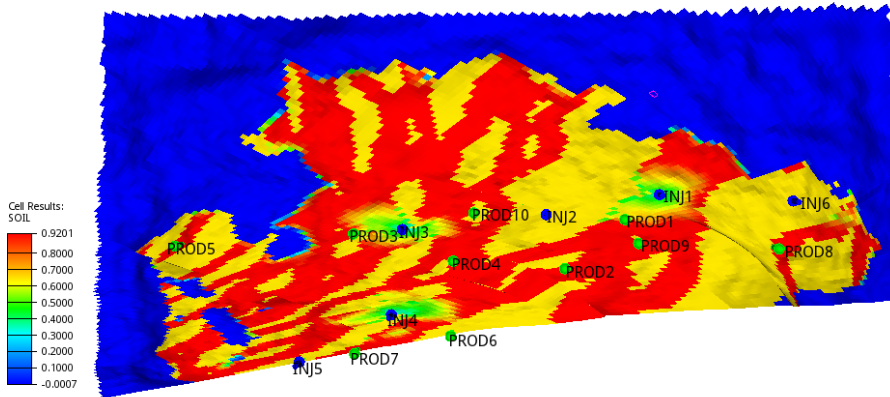
(a) Run 1



(b) Run 2



(c) Run 3



(d) Run 4

Figure 5.27: Comparison of well locations for different runs.

Figure 5.27 shows the optimized well locations for five production wells (PROD1, PROD2, PROD3, PROD4, PROD5). The background shows the oil saturation at the end of five years. It can be observed from the figure that some of the oil near the zones around the injectors, INJ1 and INJ3, move towards aquifer. This is related to the high permeability channels in the top zone of the reservoir. The presence of injectors in these zones enables early water breakthrough in the producers that are located nearby. Therefore, either locations or well control parameters of these injectors need to be optimized. We could not perform a full optimization because of the current software issues and time limitations.

Chapter 6

Conclusion and Recommendations for Further Work

This thesis has dealt with the development and integration of particle swarm optimization algorithm inside FieldOpt optimization framework. A main goal for this thesis has been to increase the capabilities of FieldOpt optimization framework by adding a new optimization algorithm that can be used to optimize the well locations on simple and realistic reservoir models. Furthermore, a standard description of the well placement problem was formulated including realistic constraints such as well length, inter-well distance and reservoir boundary constraints. A dynamic penalty function was also incorporated into particle swarm optimization algorithm in order to handle these constraints. Optimization runs have been performed using PSO with different parameters. PSO with dynamic inertia weight strategy provided better performance than static inertia weight strategy. It is also observed that setting the acceleration coefficients with the same values improved the performance of the algorithm. Although it performed better in these specific cases, it is uncertain for other models, since it depends on the objective function properties. In the case of olympus model, we could not perform more simulations to find the optimal locations because of higher computational demands and limited time. The framework also needs to be improved in order to handle realistic cases. However, the main goal of this thesis was to show the applicability of our algorithm to realistic cases.
Currently, our algorithm works only with continuous variables and it was applied only for well placement optimization. However, it can be applied easily for well control optimization and joint optimization, as well. The optimization type is not shown in FieldOpt optimization framework and it needs to be defined in driver file and added into the relevant parts inside FieldOpt in a future work. This is related to the different initializing strategies in well control, joint and well placement optimization. In the case of well control optimization, one needs to know only the maximum and minimum values of the well control variables. On the other hand, in well placement optimization, the algorithm will use the grid to take the maximum and minimum values of the well location variables. Therefore, if the optimization type is included in FieldOpt, one can define different initializing methods

depending on the type of optimization. This is also true for other optimization algorithms, such as Genetic algorithms.
One of the main challenges while developing the particle swarm optimization algorithm have been initializing the algorithm (well placement) for real reservoir models, since it is difficult to get the real coordinates of upper and lower bounds compared with rectangular simple models where the upper and lower bounds have the same real values. Therefore, we have changed the initialization of the algorithm to start with the random cells within defined regions and used functions inside FieldOpt to get the real coordinates of these cells, since the algorithm only works with real coordinates. This is a significant achievement, because the user can manually define regions for each well in driver file for initializing the algorithm and searching in a specific region for specific well. This can reduce the number of iterations to achieve an optimal solution by not wasting time for searching through all the regions that can be undesirable to locate the wells. For example, one can incorporate engineering knowledge to the problem by defining regions which have high possibility of being optimal locations. Although this is a significant achievement, we did not include the base case (reference strategy) while initializing the algorithm since we were not sure the effect of this on the performance of the algorithm. However, this can be investigated in a future work. We believe that this can increase the performance of the algorithm when a base case that is close to the optimal location is added. This case will act as a global best case and change the trajectories of the particles towards optimal solution from the beginning of the optimization. Therefore, it can converge to an optimal solution faster. However, this strategy is closer to gradient based methods where a good initial guess can achieve better results in a reasonable amount of time. To our knowledge, the effect of the inclusion of a good initial guess on the performance of PSO has not been investigated in previous works.
Although the algorithm that we have implemented is effective, there is still a high computational demand for realistic cases. This is mainly related to the optimization framework itself and it needs to be improved in the future. Another improvement to the optimization framework can be in the objective function part. The current objective function is simple and it can handle only one realization. Therefore, it needs to be improved in order to include multiple realizations at the same time. Future work will consist of applying the algorithm to more complex cases including geological uncertainty and non-conventional well types with variable production strategies as part of the overall field development problem.

Bibliography

[1] Y. J. Túpac, M. M. B. Vellasco, and M. A. C. Pacheco, "Selection of alternatives for oil field development by genetic algorithms," *Revista de Engenharia Térmica*, vol. 1, no. 2, 2002.

[2] G. Montes, P. Bartolome, A. L. Udias, *et al.*, "The use of genetic algorithms in well placement optimization," in *SPE Latin American and Caribbean Petroleum Engineering Conference*, Society of Petroleum Engineers, 2001.

[3] B. Yeten, L. J. Durlofsky, K. Aziz, *et al.*, "Optimization of nonconventional well type, location, and trajectory," *SPE Journal*, vol. 8, no. 03, pp. 200–210, 2003.

[4] J. E. Onwunalu and L. J. Durlofsky, "Application of a particle swarm optimization algorithm for determining optimum well location and type," *Computational Geosciences*, vol. 14, no. 1, pp. 183–198, 2010.

[5] J. E. Onwunalu, *Optimization of field development using particle swarm optimization and new well pattern descriptions*. PhD thesis, Stanford University, 2010.

[6] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pp. 39–43, IEEE, 1995.

[7] D. Goldberg, "Genetic algotithms in search, optimization anf machine learning," 1989.

[8] A. Y. Bukhamsin, M. M. Farshi, K. Aziz, *et al.*, "Optimization of multilateral well design and location in a real field using a continuous genetic algorithm," in *SPE/DGS Saudi Arabia Section Technical Symposium and Exhibition*, Society of Petroleum Engineers, 2010.

[9] A. C. Bittencourt, R. N. Horne, *et al.*, "Reservoir development and design optimization," in *SPE Annual Technical Conference and Exhibition*, Society of Petroleum Engineers, 1997.

[10] B. Guyaguler, R. N. Horne, *et al.*, "Uncertainty assessment of well placement optimization," in *SPE annual technical conference and exhibition*, Society of Petroleum Engineers, 2001.

[11] V. Torczon, "On the convergence of pattern search algorithms," *SIAM Journal on optimization*, vol. 7, no. 1, pp. 1–25, 1997.

[12] M. C. Bellout, D. Echeverría Ciaurri, L. J. Durlofsky, B. Foss, and J. Kleppe, "Joint optimization of oil well placement and controls," *Computational Geosciences*, pp. 1–19, 2012.

[13] T. D. Humphries and R. D. Haynes, "Joint optimization of well placement and control for nonconventional well types," *Journal of Petroleum Science and Engineering*, vol. 126, pp. 242–253, 2015.

[14] M. Jesmani, M. C. Bellout, R. Hanea, and B. Foss, "Well placement optimization subject to realistic field development constraints," *Computational Geosciences*, vol. 20, no. 6, pp. 1185–1209, 2016.

[15] E. Nwankwor, A. K. Nagar, and D. Reid, "Hybrid differential evolution and particle swarm optimization for optimal well placement," *Computational Geosciences*, pp. 1–20, 2013.

[16] M. Zandvliet, M. Handels, G. van Essen, R. Brouwer, J.-D. Jansen, *et al.*, "Adjoint-based well-placement optimization under production constraints," *SPE Journal*, vol. 13, no. 04, pp. 392–399, 2008.

[17] O. J. Isebor, L. J. Durlofsky, and D. Echeverría Ciaurri, "A derivative-free methodology with local and global search for the constrained joint optimization of well locations and controls," *Computational Geosciences*, vol. 18, no. 3, pp. 463–482, 2014.

[18] T. Humphries and R. Haynes, "Joint optimization of well placement and control for nonconventional well types," *Journal of Petroleum Science and Engineering*, vol. 126, pp. 242 – 253, 2015.

[19] S. Ding, H. Jiang, J. Li, and G. Tang, "Optimization of well placement by combination of a modified particle swarm optimization algorithm and quality map method," *Computational Geosciences*, vol. 18, no. 5, pp. 747–762, 2014.

[20] S. Ding, H. Jiang, J. Li, G. Liu, and L. Mi, "Optimization of well location, type and trajectory by a modified particle swarm optimization algorithm for the punq-s3 model," 2016.

[21] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pp. 69–73, IEEE, 1998.

[22] K. E. Parsopoulos, M. N. Vrahatis, *et al.*, "Particle swarm optimization method for constrained optimization problems," *Intelligent Technologies–Theory and Application: New Trends in Intelligent Technologies*, vol. 76, no. 1, pp. 214–220, 2002.

Appendix

Class Diagrams

This section provides basic information about the class diagrams used in this thesis.

ClassName
- privateAttribute : double + publicAttribute : vector<int>
- privateMethod (): void + publicMethod (param: vector<int>) : vector<int>

Figure 6.1: A simple class diagram

A simple class diagram is shown as in Figure 6.1.

- The name of the class is in bold on the first line
- Methods always have parentheses after the name.
- Private attributes and methods are prefixed with a '-' sign.
- Public attributes and methods are prefixed with a '+' sign.
- The type of an attribute and the return-type of a method is indicated after the last ':' sign.
- The type of a parameter is indicated after ':' sign inside the parentheses.
- When a data structure type such as 'vector' is followed by angle brackets, the word inside the brackets indicates the type of the elements which make up the data structure.

Program files

```
//
// Created by chingiz on 09.02.17.
//

#include <iostream>
#include <iomanip>
#include <Utilities/math.hpp>
#include "PSO.h"

#define ZMAX(x) (((x) > 0) ? (x) : 0.0)
#define GAMMA(x) ((x < 1) ? x : x*x)
#define THETA(x) ((x < 0.001) ? 10 : ((x <= 0.1) ? 20 : ((x <= 1) ? 100 : 300)))

namespace Optimization{
    namespace Optimizers{

        PSO::PSO(::Settings::Optimizer *settings, Case *base_case,
                 ::Model::Properties::VariablePropertyContainer *variables,
                 Reservoir::Grid::Grid *grid, Logger *logger)
                : Optimizer(settings, base_case, variables, grid, logger)
        {
            cons_ = constraint_handler_->get_constraints_from_case_handler();
            settings_=settings;
            max_iter_=(settings->parameters().max_evaluations)/(settings->parameters().number_of_particles);
            id_list_=base_case->real_variables().keys();
            gen_ = get_random_generator();
            grid_=grid;
        }

        void PSO::iterate() {
            if (iteration_==0){
                update_base_case_pso();
                case_handler_->AddNewCases(initialize_cases());}
            else
            {
                update_personal_best_cases();
                case_handler_->AddNewCases(update_particles());
            }
            case_handler_->ClearRecentlyEvaluatedCases();
            iteration_++;
        }

        Optimizer::TerminationCondition PSO::IsFinished() {
            if (case_handler_->EvaluatedCases().size()-1>=max_evaluations_)
                return MAX_EVALS_REACHED;
            else return NOT_FINISHED;
        }

        QList<Case *> PSO::initialize_cases() {
            auto cases = QList<Case *>();
```

```
51            auto particles = QList<Particle *>();
52            srand((unsigned int) time(NULL));
53            for (int i = 0; i < settings_->parameters().
      number_of_particles; ++i) {
54                auto one_case=new Case(GetTentativeBestCase());
55                one_case->set_real_variables(perturb_real_variables(
      one_case->real_variables()));
56                auto particle = new Particle(one_case,
      create_random_velocity((int) one_case->GetRealVarVector().size()));
57                particles.append(particle);
58                cases.append(one_case);
59            }
60            set_particles(particles);
61            absorb_particles(cases);
62            return cases;
63        }
64
65        QHash<QUuid,double > PSO::perturb_real_variables(QHash<QUuid,
      double > real_variables) {
66            QList<QString> names;
67            names.append("PROD1");
68            names.append("PROD2");
69            names.append("PROD3");
70            names.append("PROD4");
71            names.append("PROD5");
72            names.append("PROD6");
73            names.append("PROD7");
74            names.append("PROD8");
75            names.append("PROD9");
76            names.append("PROD10");
77
78          for (QString name: names) {
79              std::vector<int>  uniform_i,uniform_j,uniform_k;
80           if (QString::compare(name, "PROD1")==0 ){
81               uniform_i=random_integers(gen_, 90, 109, 2);
82               uniform_j=random_integers(gen_, 74, 82, 2);
83               uniform_k=random_integers(gen_, 1, 15, 2);
84          }
85          else if (QString::compare(name, "PROD2")==0){
86             uniform_i=random_integers(gen_, 77, 114, 2);
87             uniform_j=random_integers(gen_, 86, 97, 2);
88             uniform_k=random_integers(gen_, 1, 15, 2);
89          }
90          else if (QString::compare(name, "PROD3")==0){
91             uniform_i=random_integers(gen_, 53, 72, 2);
92             uniform_j=random_integers(gen_, 99, 102, 2);
93             uniform_k=random_integers(gen_, 1, 15, 2);
94          }
95          else if (QString::compare(name, "PROD4")==0) {
96              uniform_i = random_integers(gen_, 83, 101, 2);
97              uniform_j = random_integers(gen_, 99, 102, 2);
98              uniform_k = random_integers(gen_, 1, 15, 2);
99           }
100         else if (QString::compare(name, "PROD5")==0 ){
101             uniform_i=random_integers(gen_, 39, 58, 2);
102             uniform_j=random_integers(gen_, 109, 120, 2);
103             uniform_k=random_integers(gen_, 1, 15, 2);
```

```
104             }
105             else if (QString::compare(name, "PROD6")==0 ){
106                 uniform_i=random_integers(gen_, 85, 111, 2);
107                 uniform_j=random_integers(gen_, 108, 120, 2);
108                 uniform_k=random_integers(gen_, 1, 15, 2);
109             }
110              else if (QString::compare(name, "PROD7")==0 ){
111                  uniform_i=random_integers(gen_, 90, 104, 2);
112                  uniform_j=random_integers(gen_, 129, 139, 2);
113                  uniform_k=random_integers(gen_, 1, 15, 2);
114              }
115              else if (QString::compare(name, "PROD8")==0 ){
116                  uniform_i=random_integers(gen_,93,107, 2);
117                  uniform_j=random_integers(gen_,34,52,2);
118                  uniform_k=random_integers(gen_, 1,15, 2);
119              }
120              else if (QString::compare(name, "PROD9")==0 ){
121                  uniform_i=random_integers(gen_, 33, 59, 2);
122                  uniform_j=random_integers(gen_, 74, 83, 2);
123                  uniform_k=random_integers(gen_, 1, 15, 2);
124              }
125              else if (QString::compare(name, "PROD10")==0 ){
126                  uniform_i=random_integers(gen_, 50, 72, 2);
127                  uniform_j=random_integers(gen_, 86, 97, 2);
128                  uniform_k=random_integers(gen_, 1, 15, 2);
129              }
130                   real_variables[cons_->get_heel_x_id(name)]=grid_->GetCell(
        uniform_i[0],uniform_j[0],uniform_k[0]).center().x();
131                   real_variables[cons_->get_heel_y_id(name)]=grid_->GetCell(
        uniform_i[0],uniform_j[0],uniform_k[0]).center().y();
132                   real_variables[cons_->get_heel_z_id(name)]=grid_->GetCell(
        uniform_i[0],uniform_j[0],uniform_k[0]).center().z();
133                   real_variables[cons_->get_toe_x_id(name)]=grid_->GetCell(
        uniform_i[1],uniform_j[1],uniform_k[1]).center().x();
134                   real_variables[cons_->get_toe_y_id(name)]=grid_->GetCell(
        uniform_i[1],uniform_j[1],uniform_k[1]).center().y();
135                   real_variables[cons_->get_toe_z_id(name)]=grid_->GetCell(
        uniform_i[1],uniform_j[1],uniform_k[1]).center().z();
136             }
137             return real_variables;
138         }
139 
140         QHash<QUuid, double> PSO::create_random_velocity(int size) {
141             QHash<QUuid,double > velocities= QHash<QUuid, double >();
142             velocities.reserve(size);
143             for (QUuid id:id_list_) {
144                 velocities.insert(id, 0);
145             }
146             return velocities;
147         }
148 
149         QHash<QUuid, double> PSO::find_case_velocity(Case *c) {
150             auto particles=get_particles();
151             for (auto particle : particles) {
152                 if (particle->get_case()->id() == c->id()) {
153                     auto velocity= particle->get_particle_velocity();
154                     return velocity;
```

```
155                 }}
156         }
157 
158         void PSO::update_global_best_case(Case * c) {
159             if ((isImprovement(c)))
160             {
161                 updateTentativeBestCase(c);
162             }
163         }
164 
165         void PSO::set_personal_best_cases(QList<Case *>
    personal_best_cases) {
166             personal_best_cases_=personal_best_cases;
167         }
168 
169         QList<Case *> PSO::get_personal_best_cases() {
170             return personal_best_cases_;
171         }
172 
173         void PSO::update_personal_best_cases() {
174             if(iteration_==1){
175                 set_personal_best_cases(case_handler_->
    RecentlyEvaluatedCases());
176             }
177             else {
178                 for (int i=0; i<settings_->parameters().
    number_of_particles;++i) {
179                     if (mode_ == Settings::Optimizer::OptimizerMode::
    Maximize) {
180                         if (get_personal_best_cases()[i]->
    objective_function_value() < case_handler_->RecentlyEvaluatedCases()[i
    ]->objective_function_value())
181                             get_personal_best_cases()[i]=case_handler_->
    RecentlyEvaluatedCases()[i];
182                     }
183                     else if (mode_ == Settings::Optimizer::OptimizerMode::
    Minimize) {
184                         if (get_personal_best_cases()[i]->
    objective_function_value() > case_handler_->RecentlyEvaluatedCases()[i
    ]->objective_function_value())
185                             get_personal_best_cases()[i]=case_handler_->
    RecentlyEvaluatedCases()[i];
186                     }}}
187         }
188 
189         void PSO::select_neighborhood_topology() {
190             switch (settings_->neighborhood()){
191                 case Settings::Optimizer::PsoNeighborhoods ::Global:
192                     create_global_best_case_list();
193                     break;
194                 case Settings::Optimizer::PsoNeighborhoods ::Random:
195                     update_local_best_cases_random();
196                     break;
197                 case Settings::Optimizer::PsoNeighborhoods ::Ring:
198                     update_local_best_cases_ring();
199                     break;
200                 default:
```

```
201                 throw std::runtime_error("Unable to initialize
    neighborhood: neighborhood type set in driver file not recognized.");
202             }
203         }
204 
205         void PSO::set_local_best_cases(QList<Case *> local_best_cases) {
206             local_best_cases_=local_best_cases;
207         }
208 
209         void PSO::create_global_best_case_list() {
210             QList<Case *> global_best_cases;
211             global_best_cases.reserve(settings_->parameters().
    number_of_particles);
212             for (int i = 0; i <settings_->parameters().number_of_particles
     ; ++i) {
213                 global_best_cases.append(GetTentativeBestCase());
214             }
215             set_local_best_cases(global_best_cases);
216         }
217 
218         std::vector<std::vector<int >> PSO::
    create_ring_communication_matrix(int size) {
219             std::vector<std::vector<int >> ring_matrix((unsigned long)
    size, std::vector<int>((unsigned long) size));
220             for (int i = 0; i <size; ++i) {
221                 if(i==0) {
222                     ring_matrix[i][i] = 1;
223                     ring_matrix[i][i+1]=1;
224                     ring_matrix[i][size - 1] = 1;
225                 }
226                 else if(i==size-1) {
227                     ring_matrix[i][size - 1 - i] = 1;
228                     ring_matrix[i][i]=1;
229                     ring_matrix[i][i-1]=1;
230                 }
231                 else{
232                     ring_matrix[i][i]=1;
233                     ring_matrix[i][i-1]=1;
234                     ring_matrix[i][i+1]=1;
235                 }}
236             return ring_matrix;
237         }
238 
239         void PSO::update_local_best_cases_ring() {
240             QList<Case *> local_best_cases_ring;
241             local_best_cases_ring=personal_best_cases_;
242             auto ring_matrix=create_ring_communication_matrix(settings_->
    parameters().number_of_particles);
243             for (int i = 0; i < settings_->parameters().
    number_of_particles; ++i) {
244                 int bn=i;
245                 for (int j = 0; j <settings_->parameters().
    number_of_particles ; ++j) {
246                     if (ring_matrix[i][j] && personal_best_cases_[j]->
    objective_function_value()< personal_best_cases_[bn]->
    objective_function_value()){
247                         bn=j;
```

```
248                             local_best_cases_ring[i]=personal_best_cases_[bn];
249                         }}}
250             set_local_best_cases(local_best_cases_ring);
251         }
252 
253         std::vector<std::vector<int >> PSO::
    create_random_communication_matrix(int size) {
254             std::vector<std::vector<int >> random_matrix((unsigned long)
    size, std::vector<int>((unsigned long) size));
255             vector<int> rn(5);
256             for (int i = 0; i <size; ++i) {
257                 random_matrix[i][i]=1;
258                 for (int j = 0; j <size ; ++j) {
259                     for(int k=0;k<5;++k){
260                         rn[k]=rand()%(size -1);
261                         random_matrix[i][rn[k]]=1;
262                     }}
263             }
264             return random_matrix;
265         }
266 
267         void PSO::update_local_best_cases_random() {
268             QList<Case *> local_best_cases_random;
269             local_best_cases_random=personal_best_cases_;
270             auto random_matrix=create_random_communication_matrix(
    settings_->parameters().number_of_particles);
271             for (int i = 0; i < settings_->parameters().
    number_of_particles; ++i) {
272                 int bn=i;
273                 for (int j = 0; j <settings_->parameters().
    number_of_particles ; ++j) {
274                     if (random_matrix[i][j] && personal_best_cases_[j]->
    objective_function_value()< personal_best_cases_[bn]->
    objective_function_value()){
275                         bn=j;
276                         local_best_cases_random[i]=personal_best_cases_[bn
    ];
277                     }}}
278             set_local_best_cases(local_best_cases_random);
279 
280         }
281 
282         QList<Case *> PSO::update_particles() {
283             select_neighborhood_topology();
284             QList<Case *> new_cases = QList<Case *>();
285             QList<Particle *> new_particles=QList<Particle *>();
286             double iw_start=settings_->parameters().inertia_weight1;
287             double iw_end=settings_->parameters().inertia_weight2;
288             double iw=iw_end+(iw_start-iw_end)*((max_iter_-iteration_)/
    max_iter_);
289             for (int i = 0; i < settings_->parameters().
    number_of_particles; ++i) {
290                 auto gbest_pos = get_local_best_cases()[i]->real_variables
    ();
291                 auto new_case = new Case(GetTentativeBestCase());
292                 auto new_vel=find_case_velocity(case_handler_->
    RecentlyEvaluatedCases()[i]);
```

```
293                 auto new_pos = case_handler_->RecentlyEvaluatedCases()[i
        ]->real_variables();
294                 auto pbest_pos = personal_best_cases_[i]->real_variables()
        ;
295                 auto c1=settings_->parameters().coefficient1;
296                 auto c2=settings_->parameters().coefficient2;
297                 auto rn1=((double)rand()/RAND_MAX);
298                 auto rn2=((double)rand()/RAND_MAX);
299                 for (QUuid id:id_list_) {
300                     new_vel[id]=iw*new_vel[id]+c1*rn1*(pbest_pos[id]-
        new_pos[id])+c2*rn2*(gbest_pos[id]-new_pos[id]);
301                     new_pos[id]=new_pos[id]+new_vel[id];
302                 }
303                 new_case->set_real_variables(new_pos);
304                 auto particle=new Particle(new_case,new_vel);
305                 new_particles.append(particle);
306                 new_cases.append(new_case);
307             }
308             set_particles(new_particles);
309             absorb_particles(new_cases);
310             return new_cases;
311         }
312 
313         QList<Case *> PSO::get_local_best_cases() {
314             return local_best_cases_;
315         }
316 
317         void PSO::apply_penalty(Case *c) {
318             QList<QString> names;
319             names.append("PROD1");
320             names.append("PROD2");
321             names.append("PROD3");
322             names.append("PROD4");
323             names.append("PROD5");
324             names.append("PROD6");
325             names.append("PROD7");
326             names.append("PROD8");
327             names.append("PROD9");
328             names.append("PROD10");
329             double min=cons_->get_well_min_length();
330             double max=cons_->get_well_max_length();
331             double f=c->objective_function_value();
332             double distance0=cons_->get_shortest_distance_2_wells(c,"PROD1
        ","PROD9");
333             double distance1=cons_->get_shortest_distance_2_wells(c,"PROD2
        ","PROD10");
334             double distance2=cons_->get_shortest_distance_2_wells(c,"PROD3
        ","PROD4");
335             double distance3=cons_->get_shortest_distance_2_wells(c,"PROD5
        ","PROD6");
336             double length1=cons_->get_well_length(c, "PROD1");
337             double length2=cons_->get_well_length(c, "PROD2");
338             double length3=cons_->get_well_length(c, "PROD3");
339             double length4=cons_->get_well_length(c, "PROD4");
340             double length5=cons_->get_well_length(c, "PROD5");
341             double length6=cons_->get_well_length(c, "PROD6");
342             double length7=cons_->get_well_length(c, "PROD7");
```

```
343            double length8=cons_->get_well_length(c, "PROD8");
344            double length9=cons_->get_well_length(c, "PROD9");
345            double length10=cons_->get_well_length(c, "PROD10");
346            std::vector<double > q(24);
347            double H=0.0;
348            q[0]=ZMAX(200-distance0);
349            q[1]=ZMAX(200-distance1);
350            q[2]=ZMAX(200-distance2);
351            q[3]=ZMAX(200-distance3);
352            q[4]=ZMAX(min-length1);
353            q[5]=ZMAX(length1-max);
354            q[6]=ZMAX(min-length2);
355            q[7]=ZMAX(length2-max);
356            q[8]=ZMAX(min-length3);
357            q[9]=ZMAX(length3-max);
358            q[10]=ZMAX(min-length4);
359            q[11]=ZMAX(length4-max);
360            q[12]=ZMAX(min-length5);
361            q[13]=ZMAX(length5-max);
362            q[14]=ZMAX(min-length5);
363            q[15]=ZMAX(length6-max);
364            q[16]=ZMAX(min-length7);
365            q[17]=ZMAX(length7-max);
366            q[18]=ZMAX(min-length8);
367            q[19]=ZMAX(length8-max);
368            q[20]=ZMAX(min-length9);
369            q[21]=ZMAX(length9-max);
370            q[22]=ZMAX(min-length10);
371            q[23]=ZMAX(length10-max);
372
373            for(int i = 0; i < 24; ++i) {
374                if (q[i] > 0) {
375                    H += THETA (q[i]) * GAMMA (q[i]);
376                }
377            }
378            double ck=iteration_*iteration_;
379            double new_objective_value=f-ck*H;
380            case_handler_->UpdateCaseObjectiveFunctionValue(c->id(),
    new_objective_value);
381        }
382
383        void PSO::handleEvaluatedCase(Case *c) {
384            apply_penalty(c);
385            update_global_best_case(c);
386        }
387
388        void PSO::update_base_case_pso() {
389            if (mode_ == Settings::Optimizer::OptimizerMode::Maximize) {
390                case_handler_->UpdateCaseObjectiveFunctionValue(
    GetTentativeBestCase()->id(),std::numeric_limits<double>::min());
391            }
392            else if (mode_ == Settings::Optimizer::OptimizerMode::Minimize
    ) {
393                case_handler_->UpdateCaseObjectiveFunctionValue(
    GetTentativeBestCase()->id(),std::numeric_limits<double>::max());
394            }
395
```

```
396         }
397
398         void PSO::absorb_particles(QList<Case *> cases) {
399             QList<QString> names;
400             names.append("PROD1");
401             names.append("PROD2");
402             names.append("PROD3");
403             names.append("PROD4");
404             names.append("PROD5");
405             names.append("PROD6");
406             names.append("PROD7");
407             names.append("PROD8");
408             names.append("PROD9");
409             names.append("PROD10");
410             for (Case *c: cases) {
411                 for (QString name:names) {
412                     bool heel_feasible = false;
413                     bool toe_feasible = false;
414                     auto heelx=c->real_variables()[cons_->get_heel_x_id(
    name)];
415                     auto heely=c->real_variables()[cons_->get_heel_y_id(
    name)];
416                     auto heelz=c->real_variables()[cons_->get_heel_z_id(
    name)];
417                     auto toex=c->real_variables()[cons_->get_toe_x_id(name
    )];
418                     auto toey=c->real_variables()[cons_->get_toe_y_id(name
    )];
419                     auto toez=c->real_variables()[cons_->get_toe_z_id(name
    )];
420                     if (QString::compare(name, "PROD1")==0 ){
421                         index_list_=get_cell_indices_of_regions
    (69,117,73,84,1,15);
422                     }
423                     else if (QString::compare(name, "PROD2")==0){
424                         index_list_=get_cell_indices_of_regions
    (73,117,85,98,1,15);
425                     }
426                     else if (QString::compare(name, "PROD3")==0 ){
427                         index_list_=get_cell_indices_of_regions
    (53,82,99,102,1,15);
428                     }
429                     else if (QString::compare(name, "PROD4")==0){
430                         index_list_=get_cell_indices_of_regions
    (83,117,99,102,1,15);
431                     }
432                     else if (QString::compare(name, "PROD5")==0 ){
433                         index_list_=get_cell_indices_of_regions
    (37,68,103,122,1,15);
434                     }
435                     else if (QString::compare(name, "PROD6")==0 ){
436                         index_list_=get_cell_indices_of_regions
    (69,117,103,122,1,15);
437                     }
438                     else if (QString::compare(name, "PROD7")==0 ){
439                         index_list_=get_cell_indices_of_regions
    (57,117,123,153,1,15);
```

```
440                     }
441                     else if (QString::compare(name, "PROD8")==0 ){
442                         index_list_=get_cell_indices_of_regions
    (77,117,24,55,1,15);
443                     }
444                     else if (QString::compare(name, "PROD9")==0 ){
445                         index_list_=get_cell_indices_of_regions
    (29,68,73,84,1,15);
446                     }
447                     else if (QString::compare(name, "PROD10")==0 ){
448                         index_list_=get_cell_indices_of_regions
    (34,72,85,98,1,15);
449                     }
450 
451                     for (int ii=0; ii<index_list_.length(); ii++){
452                         if (grid_->GetCell(index_list_[ii]).EnvelopsPoint(
453                                 Eigen::Vector3d(heelx, heely, heelz))) {
454                             heel_feasible = true;
455                         }
456                         if (grid_->GetCell(index_list_[ii]).EnvelopsPoint(
457                                 Eigen::Vector3d(toex, toey, toez))) {
458                             toe_feasible = true;
459                         }
460                     }
461                     if (!heel_feasible){
462                         Eigen::Vector3d projected_heel =
    WellConstraintProjections::well_domain_constraint_indices(Eigen::
    Vector3d(heelx, heely, heelz), grid_, index_list_);
463                         if ((c->real_variables()[cons_->get_heel_x_id(name
    )])!=projected_heel(0)){
464                             c->set_real_variable_value(cons_->
    get_heel_x_id(name), projected_heel(0));
465                             change_velocity(c, cons_->get_heel_x_id(name))
    ;
466                         }
467                         if ((c->real_variables()[cons_->get_heel_y_id(name
    )])!=projected_heel(1)){
468                             c->set_real_variable_value(cons_->
    get_heel_y_id(name), projected_heel(1));
469                             change_velocity(c,cons_->get_heel_y_id(name));
470                         }
471                         if ((c->real_variables()[cons_->get_heel_z_id(name
    )])!=projected_heel(2)) {
472                             c->set_real_variable_value(cons_->
    get_heel_z_id(name), projected_heel(2));
473                             change_velocity(c, cons_->get_heel_z_id(name))
    ;
474                         }
475                     }
476                     if (!toe_feasible){
477                         Eigen::Vector3d projected_toe =
    WellConstraintProjections::well_domain_constraint_indices(Eigen::
    Vector3d(toex, toey, toez), grid_, index_list_);
478                         if ((c->real_variables()[cons_->get_toe_x_id(name)
    ])!=projected_toe(0)){
479                             c->set_real_variable_value(cons_->get_toe_x_id
    (name), projected_toe(0));
```

```
480                             change_velocity(c, cons_->get_toe_x_id(name));
481                         }
482                         if ((c->real_variables()[cons_->get_toe_y_id(name)
    ])!=projected_toe(1)){
483                             c->set_real_variable_value(cons_->get_toe_y_id
    (name), projected_toe(1));
484                             change_velocity(c,cons_->get_toe_y_id(name));
485                         }
486                         if ((c->real_variables()[cons_->get_toe_z_id(name)
    ])!=projected_toe(2)) {
487                             c->set_real_variable_value(cons_->get_toe_z_id
    (name), projected_toe(2));
488                             change_velocity(c, cons_->get_toe_z_id(name));
489                         }
490                     }
491                 }
492 
493             }
494         }
495 
496         void PSO::change_velocity(Case *c, QUuid id) {
497             auto particles=get_particles();
498             for (auto particle : particles) {
499                 if (particle->get_case()->id() == c->id()) {
500                     particle->set_particle_velocity(id,0);
501                 }
502             }
503         }
504 
505         QList<int> PSO::get_cell_indices_of_regions(int imin, int imax,
    int jmin, int jmax, int kmin, int kmax) {
506             QList<int> index_list;
507             for (int i = imin; i <= imax; i++){
508                 for (int j = jmin; j <= jmax; j++){
509                     for (int k = kmin; k <= kmax; k++){
510                         index_list.append(grid_->GetCell(i, j, k).
    global_index());
511                     }
512                 }
513             }
514             return index_list;
515         }
516     }}
```

Listing 6.1: PSO.cpp

```
//
// Created by chingiz on 09.02.17.
//

#ifndef FIELDOPT_PSO_H
#define FIELDOPT_PSO_H

#include "optimizer.h"
#include "Particle.h"
#include <boost/random.hpp>
#include "ConstraintMath/well_constraint_projections/
    well_constraint_projections.h"

namespace Optimization{
    namespace Optimizers{

        /*!
         * @brief The PSO class implements a modified form of the
         * original Particle Swarm Optimization as presented by Shi and
    Eberhart
         * in the 1998 paper A Modified particle swarm optimizer. The
    modifed Particle Swarm
         * Optimizer includes inertia weight in velocity update equation.
         *
         * This implementation also borrows some from the Maurice Clerc's
    book Particle Swarm
         * Optimization.
         * @note  This algorithm can only be applied to continuous
    variables. The algorithm
         * implements two main types of Particle Swarm Optimization known
    as gbest and lbest
         * Particle Swarm Optimization. The ring and random neighborhood
    topologies are
         * implemented for lbest Particle Swarm Optimization.
        */

        class PSO : public Optimizer {
        public:

            /*!
             * @brief General constructor for PSO algorithm that calls the
     primary
             * Optimizer constructor.
            */
            PSO(::Settings::Optimizer *settings, Case *base_case,
                ::Model::Properties::VariablePropertyContainer *variables,
                Reservoir::Grid::Grid *grid, Logger *logger);
        private:
            QList<QUuid> id_list_;
            QHash<QUuid,double > real_max_;
            QHash<QUuid,double > real_min_;
            double max_iter_;
            Optimization::Constraints::
    CombinedSplineLengthInterwellDistanceReservoirBoundary *cons_;
            Settings::Optimizer *settings_;
            QList<int> index_list_;
            Reservoir::Grid::Grid *grid_;
```

```
49          QList<Particle *> particles_; //!< List of particles.
50          QList<Case *> personal_best_cases_; //!< List of personal best
     cases.
51          QList<Case *> local_best_cases_; //!< List of local best cases
     for each particle
52
53          virtual void iterate() override; //!< Create cases, find
    personal and global best cases and  update cases, clear recently
    evaluated cases.
54          /*!
55           * \brief IsFinished Check if the optimization is finished.
56           *
57           * This algorithm has one termination condition: max number of
     objective function evaluations
58           * \return True if the algorithm has finished, otherwise false
    .
59           */
60          virtual TerminationCondition IsFinished() override;
61          QList<Case *> initialize_cases(); //!< Creates a list of
    different cases based on base case.
62          QHash<QUuid,double > perturb_real_variables(QHash<QUuid,double
     >);//!< Perturbs the base case variables randomly.
63          QHash<QUuid,double > create_random_velocity(int); //!< Creates
     a random velocity vector for each particle. Takes same IDs as real
    variables.
64          void set_particles(QList<Particle *>particles){particles_=
    particles;} //!< Saves the list of particles.
65          QList<Particle *> get_particles(){ return particles_;} //!<
    Gets the list of saved particles.
66          QHash<QUuid,double > find_case_velocity(Case *c); //!< Finds
    the velocity of the particle according to the specific case.
67          void update_global_best_case(Case *c); //!< Finds the best
    case among all evaluated cases based on objective function value.
68          void set_personal_best_cases(QList<Case *>); //!< Saves the
    personal best case for each particle.
69          QList<Case *> get_personal_best_cases(); //!< Gets the list of
     personal best cases.
70          void update_personal_best_cases(); //!< Updates the personal
    best cases
71          void select_neighborhood_topology(); //!< Selects
    neighborhoods defined in settings
72          void set_local_best_cases(QList<Case *>); //!< Sets the local
    best case for each particle
73          void create_global_best_case_list(); //!< Creates a list of
    global best case
74          std::vector<std::vector<int >>
    create_ring_communication_matrix(int); //!< Create ring communication
    matrix. Every particle communicates with itself and  two adjacent
    particles.
75          void update_local_best_cases_ring(); //!< Updates the local
    best cases for each particle based on ring topology
76          std::vector<std::vector<int >>
    create_random_communication_matrix(int); //!< Create random
    communication matrix. Every particle communicates with itself and
    other particles chosen randomly.
77          void update_local_best_cases_random(); //!< Updates the local
    best cases for each particle based on random topology.
```

```
78            QList<Case *> update_particles(); //!< Updates particles (
    cases and velocities) and returns list of updated cases.
79            QList<Case *> get_local_best_cases(); //!< Get the list of
    local best cases
80            void apply_penalty(Case *c);
81            void update_base_case_pso();
82            void absorb_particles(QList<Case *>);
83            void change_velocity(Case *c, QUuid id);
84            QList<int> get_cell_indices_of_regions(int imin, int imax, int
     jmin, int jmax, int kmin, int kmax);
85        protected:
86            void handleEvaluatedCase(Case *c) override;
87            boost::random::mt19937 gen_; //!< Random number generator with
     the random functions in math.hpp
88        };
89    }
90 }
91
92
93
94 #endif //FIELDOPT_PSO_H
```

Listing 6.2: PSO.h

```
1  //
2  // Created by chingiz on 09.02.17.
3  //
4
5  #include "Particle.h"
6  namespace Optimization{
7      namespace Optimizers{
8
9          Particle::Particle(Case *c, const QHash<QUuid, double> &velocity)
    {
10             particle_case_=c;
11             particle_velocity_=velocity;
12             velocity_id_index_map_=particle_velocity_.keys();
13         }
14     }
15 }
```

Listing 6.3: Particle.cpp

```
1  //
2  // Created by chingiz on 09.02.17.
3  //
4
5  #ifndef FIELDOPT_PARTICLE_H
6  #define FIELDOPT_PARTICLE_H
7
8  #include "Optimization/case.h"
9
10 namespace Optimization{
11     namespace Optimizers{
12
13         /*!
```

```
14           * @brief The Particle class holds the case and associated
     velocitiy.
15           * The purpose of the Partilce class is to assign a unique
     velocity for each case.
16          */
17
18          class Particle {
19          public:
20              /*!
21               * @brief The Particle constructor receives the case and
     velocity and assigns them to a Particle object.
22               */
23              Particle(Case* c, const  QHash<QUuid , double> &velocity);
24              Case* get_case() const { return particle_case_; } //! Get case
      from particle object
25              QHash<QUuid,double > get_particle_velocity() const {return
     particle_velocity_;} //!< Get particle velocity
26              void set_particle_velocity(const QUuid id, const double val){
     particle_velocity_[id]=val;} //! Set particle velocity
27          private:
28              QHash<QUuid , double> particle_velocity_;
29              QList<QUuid > velocity_id_index_map_;
30              Case *particle_case_;
31          };
32      }
33  }
34
35  #endif //FIELDOPT_PARTICLE_H
```

Listing 6.4: Particle.h

```
1  {
2      "Global": {
3          "Name": "OLYMPUS",
4          "BookkeeperTolerance": 2.0
5      },
6      "Optimizer": {
7          "Type": "PSO",
8          "Mode": "Maximize",
9          "Neighborhood": "Global",
10         "Parameters": {
11             "MaxEvaluations":100,
12             "NumberOfParticles":20.0,
13             "Coefficient1": 1.193,
14             "Coefficient2": 1.193,
15             "InertiaWeight1": 1.2,
16             "InertiaWeight2": 0.2
17         },
18         "Objective": {
19             "Type": "WeightedSum",
20             "WeightedSumComponents": [
21                 {
22                     "Coefficient": 1.0, "Property": "
     CumulativeOilProduction", "TimeStep": -1,
23                     "IsWellProp": false
24                 },
25                 {
```

```
26                          "Coefficient": −0.2, "Property": "
    CumulativeWaterProduction", "TimeStep": −1,
27                          "IsWellProp": false
28                      }
29                  ]
30              },
31              "Constraints": [
32                  {
33                      "Wells": ["PROD1", "PROD2", "PROD3","PROD4","PROD5"],
34                      "Type": "
    CombinedWellSplineLengthInterwellDistanceReservoirBoundary",
35                      "MinLength": 400,
36                      "MaxLength": 1100,
37                      "MinDistance" :200,
38                      "MaxIterations": 5,
39                      "BoxImin": 0,
40                      "BoxImax": 117,
41                      "BoxJmin": 0,
42                      "BoxJmax": 180,
43                      "BoxKmin": 0,
44                      "BoxKmax": 15
45                  }
46              ]
47          },
48         "Simulator": {
49              "Type": "ECLIPSE",
50              "ExecutionScript": "csh_eclrun"
51          },
52          "Model": {
53              "ControlTimes": [0, 50],
54              "Reservoir": {
55                  "Type": "ECLIPSE"
56              },
57              "Wells": [
58                  {
59                      "Name": "PROD1",
60                      "Group": "GROUP1",
61                      "Type": "Producer",
62                      "DefinitionType": "WellSpline",
63                      "PreferredPhase": "Oil",
64                      "WellboreRadius": 0.1905,
65                          "SplinePoints": {
66                          "Heel": {
67                              "x": 526275,
68                              "y": 6179700,
69                              "z": 2053.27,
70                              "IsVariable": true
71                          },
72                          "Toe": {
73                              "x": 525474,
74                              "y": 6180380,
75                              "z": 2063.72,
76                              "IsVariable": true
77                          }
78                      },
79                      "Controls": [
80                          {
```

```
81                         "TimeStep": 0,
82                         "State": "Open",
83                         "Mode": "BHP",
84                         "BHP": 175.0
85                     }
86                 ]
87             },
88 {
89                 "Name": "PROD2",
90                 "Group": "GROUP1",
91                 "Type": "Producer",
92                 "DefinitionType": "WellSpline",
93                 "PreferredPhase": "Oil",
94                 "WellboreRadius": 0.1905,
95                                "SplinePoints": {
96                     "Heel": {
97                         "x": 524288,
98                         "y": 6179780,
99                         "z": 2035.16,
100                        "IsVariable": true
101                    },
102                    "Toe": {
103                        "x": 523913,
104                        "y": 6180080,
105                        "z": 2046,
106                        "IsVariable": true
107                    }
108                },
109                "Controls": [
110                    {
111                        "TimeStep": 0,
112                        "State": "Open",
113                        "Mode": "BHP",
114                        "BHP": 175.0
115                    }
116                ]
117            },
118 {
119                "Name": "PROD3",
120                "Group": "GROUP1",
121                "Type": "Producer",
122                "DefinitionType": "WellSpline",
123                "PreferredPhase": "Oil",
124                "WellboreRadius": 0.1905,
125                   "SplinePoints": {
126                    "Heel": {
127                        "x": 523435,
128                        "y": 6179700,
129                        "z": 2043.51,
130                        "IsVariable": true
131                    },
132                    "Toe": {
133                        "x": 522760,
134                        "y": 6179990,
135                        "z": 2066.2,
136                        "IsVariable": true
137                    }
```

```
138                     },
139                     "Controls": [
140                         {
141                             "TimeStep": 0,
142                             "State": "Open",
143                             "Mode": "BHP",
144                             "BHP": 175.0
145                         }
146                     ]
147                 },
148 {
149                     "Name": "PROD4",
150                     "Group": "GROUP1",
151                     "Type": "Producer",
152                     "DefinitionType": "WellSpline",
153                     "PreferredPhase": "Oil",
154                     "WellboreRadius": 0.1905,
155                    "SplinePoints": {
156                         "Heel": {
157                             "x": 524374,
158                             "y": 6179290,
159                             "z": 2056.5,
160                             "IsVariable": true
161                         },
162                         "Toe": {
163                             "x": 523648,
164                             "y": 6179720,
165                             "z": 2076.3,
166                             "IsVariable": true
167                         }
168                     },
169                     "Controls": [
170                         {
171                             "TimeStep": 0,
172                             "State": "Open",
173                             "Mode": "BHP",
174                             "BHP": 175.0
175                         }
176                     ]
177                 },
178 {
179                     "Name": "PROD5",
180                     "Group": "GROUP1",
181                     "Type": "Producer",
182                     "DefinitionType": "WellSpline",
183                     "PreferredPhase": "Oil",
184                     "WellboreRadius": 0.1905,
185                    "SplinePoints": {
186                         "Heel": {
187                             "x": 523593,
188                             "y": 6179230,
189                             "z": 2035.31,
190                             "IsVariable": true
191                         },
192                         "Toe": {
193                             "x": 523500,
194                             "y": 6179270,
```

```
195                     "z": 2055.74,
196                     "IsVariable": true
197                 }
198             },
199             "Controls": [
200                 {
201                     "TimeStep": 0,
202                     "State": "Open",
203                     "Mode": "BHP",
204                     "BHP": 175.0
205                 }
206             ]
207         },
208 {
209             "Name": "PROD6",
210             "Group": "GROUP1",
211             "Type": "Producer",
212             "DefinitionType": "WellSpline",
213             "PreferredPhase": "Oil",
214             "WellboreRadius": 0.1905,
215             "SplinePoints": {
216                 "Heel": {
217                     "x": 522680,
218                     "y": 6179130,
219                     "z": 2039.99,
220                     "IsVariable": false
221                 },
222                 "Toe": {
223                     "x": 522696,
224                     "y": 6179560,
225                     "z": 2051.77,
226                     "IsVariable": false
227                 }
228             },
229             "Controls": [
230                 {
231                     "TimeStep": 0,
232                     "State": "Open",
233                     "Mode": "BHP",
234                     "BHP": 175.0
235                 }
236             ]
237         },
238
239 {
240             "Name": "PROD7",
241             "Group": "GROUP1",
242             "Type": "Producer",
243             "DefinitionType": "WellSpline",
244             "PreferredPhase": "Oil",
245             "WellboreRadius": 0.1905,
246            "SplinePoints": {
247                 "Heel": {
248                     "x": 521534,
249                     "y": 6178900,
250                     "z": 2060.43,
251                     "IsVariable": false
```

```
252                         },
253                         "Toe": {
254                             "x": 521184,
255                             "y": 6179090,
256                             "z": 2066.91,
257                             "IsVariable": false
258                         }
259                     },
260                     "Controls": [
261                         {
262                             "TimeStep": 0,
263                             "State": "Open",
264                             "Mode": "BHP",
265                             "BHP": 175.0
266                         }
267                     ]
268                 },
269  {
270                     "Name": "PROD8",
271                     "Group": "GROUP1",
272                     "Type": "Producer",
273                     "DefinitionType": "WellSpline",
274                     "PreferredPhase": "Oil",
275                     "WellboreRadius": 0.1905,
276                     "SplinePoints": {
277                         "Heel": {
278                             "x": 526541,
279                             "y": 6180150,
280                             "z": 2044.62,
281                             "IsVariable": false
282                         },
283                         "Toe": {
284                             "x": 526654,
285                             "y": 6180150,
286                             "z": 2090.4,
287                             "IsVariable": false
288                         }
289                     },
290                     "Controls": [
291                         {
292                             "TimeStep": 0,
293                             "State": "Open",
294                             "Mode": "BHP",
295                             "BHP": 175.0
296                         }
297                     ]
298                 },
299  {
300                     "Name": "PROD9",
301                     "Group": "GROUP1",
302                     "Type": "Producer",
303                     "DefinitionType": "WellSpline",
304                     "PreferredPhase": "Oil",
305                     "WellboreRadius": 0.1905,
306                     "SplinePoints": {
307                         "Heel": {
308                             "x": 524895,
```

```
309                         "y": 6180210,
310                         "z": 2053.39,
311                         "IsVariable": false
312                     },
313                     "Toe": {
314                         "x": 524905,
315                         "y": 6180200,
316                         "z": 2088.22,
317                         "IsVariable": false
318                     }
319                 },
320                 "Controls": [
321                     {
322                         "TimeStep": 0,
323                         "State": "Open",
324                         "Mode": "BHP",
325                         "BHP": 175.0
326                     }
327                 ]
328             },
329 {
330                 "Name": "PROD10",
331                 "Group": "GROUP1",
332                 "Type": "Producer",
333                 "DefinitionType": "WellSpline",
334                 "PreferredPhase": "Oil",
335                 "WellboreRadius": 0.1905,
336                 "SplinePoints": {
337                     "Heel": {
338                         "x": 522946,
339                         "y": 6180570,
340                         "z": 2064.64,
341                         "IsVariable": false
342                     },
343                     "Toe": {
344                         "x": 522949,
345                         "y": 6180570,
346                         "z": 2083.48,
347                         "IsVariable": false
348                     }
349                 },
350                 "Controls": [
351                     {
352                         "TimeStep": 0,
353                         "State": "Open",
354                         "Mode": "BHP",
355                         "BHP": 175.0
356                     }
357                 ]
358             },
359
360
361
362              {
363                 "Name": "INJ1",
364                 "Group": "GROUP1",
365                 "Type": "Injector",
```

```
366                 "DefinitionType": "WellSpline",
367                 "PreferredPhase": "Water",
368                 "WellboreRadius": 0.1905,
369                 "SplinePoints": {
370                     "Heel": {
371                         "x": 525166,
372                         "y": 6180780,
373                         "z": 2054.3,
374                         "IsVariable": false
375                     },
376                     "Toe": {
377                         "x": 525215,
378                         "y": 6180790,
379                         "z": 2080.58,
380                         "IsVariable": false
381                     }
382                 },
383                 "Controls": [
384                     {
385                         "TimeStep": 0,
386                         "Type": "Water",
387                         "State": "Open",
388                         "Mode": "BHP",
389                         "BHP": 235.0
390                     }
391                 ]
392             },
393             {
394                 "Name": "INJ2",
395                 "Group": "GROUP1",
396                 "Type": "Injector",
397                 "DefinitionType": "WellSpline",
398                 "PreferredPhase": "Water",
399                 "WellboreRadius": 0.1905,
400                 "SplinePoints": {
401                     "Heel": {
402                         "x": 523809,
403                         "y": 6180540,
404                         "z": 2053.51,
405                         "IsVariable": false
406                     },
407                     "Toe": {
408                         "x": 523816,
409                         "y": 6180540,
410                         "z": 2084.51,
411                         "IsVariable": false
412                     }
413                 },
414                 "Controls": [
415                     {
416                         "TimeStep": 0,
417                         "Type": "Water",
418                         "State": "Open",
419                         "Mode": "BHP",
420                         "BHP": 235.0
421                     }
422                 ]
```

```
423                 },
424                 {
425                     "Name": "INJ3",
426                     "Group": "GROUP1",
427                     "Type": "Injector",
428                     "DefinitionType": "WellSpline",
429                     "PreferredPhase": "Water",
430                     "WellboreRadius": 0.1905,
431                     "SplinePoints": {
432                         "Heel": {
433                             "x": 522086,
434                             "y": 6180360,
435                             "z": 2056.03,
436                             "IsVariable": false
437                         },
438                         "Toe": {
439                             "x": 522088,
440                             "y": 6180360,
441                             "z": 2081.07,
442                             "IsVariable": false
443                         }
444                     },
445                     "Controls": [
446                         {
447                             "TimeStep": 0,
448                             "Type": "Water",
449                             "State": "Open",
450                             "Mode": "BHP",
451                             "BHP": 235.0
452                         }
453                     ]
454                 },
455                 {
456                     "Name": "INJ4",
457                     "Group": "GROUP1",
458                     "Type": "Injector",
459                     "DefinitionType": "WellSpline",
460                     "PreferredPhase": "Water",
461                     "WellboreRadius": 0.1905,
462                     "SplinePoints": {
463                         "Heel": {
464                             "x": 521956,
465                             "y": 6179340,
466                             "z": 2056.73,
467                             "IsVariable": false
468                         },
469                         "Toe": {
470                             "x": 522008,
471                             "y": 6179320,
472                             "z": 2083.8,
473                             "IsVariable": false
474                         }
475                     },
476                     "Controls": [
477                         {
478                             "TimeStep": 0,
479                             "Type": "Water",
```

```
480                     "State": "Open",
481                     "Mode": "BHP",
482                     "BHP": 235.0
483                 }
484             ]
485         },
486         {
487             "Name": "INJ5",
488             "Group": "GROUP1",
489             "Type": "Injector",
490             "DefinitionType": "WellSpline",
491             "PreferredPhase": "Water",
492             "WellboreRadius": 0.1905,
493             "SplinePoints": {
494                 "Heel": {
495                     "x": 520840,
496                     "y": 6178760,
497                     "z": 2067.93,
498                     "IsVariable": false
499                 },
500                 "Toe": {
501                     "x": 520758,
502                     "y": 6178740,
503                     "z": 2086.28,
504                     "IsVariable": false
505                 }
506             },
507             "Controls": [
508                 {
509                     "TimeStep": 0,
510                     "Type": "Water",
511                     "State": "Open",
512                     "Mode": "BHP",
513                     "BHP": 235.0
514                 }
515             ]
516         },
517         {
518             "Name": "INJ6",
519             "Group": "GROUP1",
520             "Type": "Injector",
521             "DefinitionType": "WellSpline",
522             "PreferredPhase": "Water",
523             "WellboreRadius": 0.1905,
524             "SplinePoints": {
525                 "Heel": {
526                     "x": 526782,
527                     "y": 6180700,
528                     "z": 2064.84,
529                     "IsVariable": false
530                 },
531                 "Toe": {
532                     "x": 526784,
533                     "y": 6180700,
534                     "z": 2080.45,
535                     "IsVariable": false
536                 }
```

```
537                     },
538                     "Controls": [
539                         {
540                             "TimeStep": 0,
541                             "Type": "Water",
542                             "State": "Open",
543                             "Mode": "BHP",
544                             "BHP": 235.0
545                         }
546                     ]
547                 }
548 
549             ]
550         }
551 }
```

Listing 6.5: PSO Driver file