Chapter 3

Implementation and System Overview

The chapter of implementation and system overview is made to give an overview over the working process and an explanation of the code produced. The planning and execution of the master thesis is presented first in this chapter, before the Prosper well-model is described. Further, the code for importing data from the well-model to Python is presented, including some theory that helps explaining the process. A mathematical formulation of the interpolation done is presented before the implementation of the code is described. Next, the mathematical formulation of the different optimization problems are derived leading up to the implementation of the code, which the next section is about. Section 3.8 is about the code made for plotting, before the structure of the system is presented in the last section. The structure of the system is mainly presented with the use of class- and flowchart diagrams.

3.1 Planning and Execution

To make a project schedule a Gantt chart was produced. This is a graphical depiction of a project schedule with bar charts showing start and finish dates [9]. A part of the Gantt chart is presented in figure 3.1. The figure shows different phases in the work and each phase consists of multiple tasks. In addition it presents how many weeks are planned used for each task. The last part of the chart is phase 3: *"Writing the report"* and is left out due to space limitations.

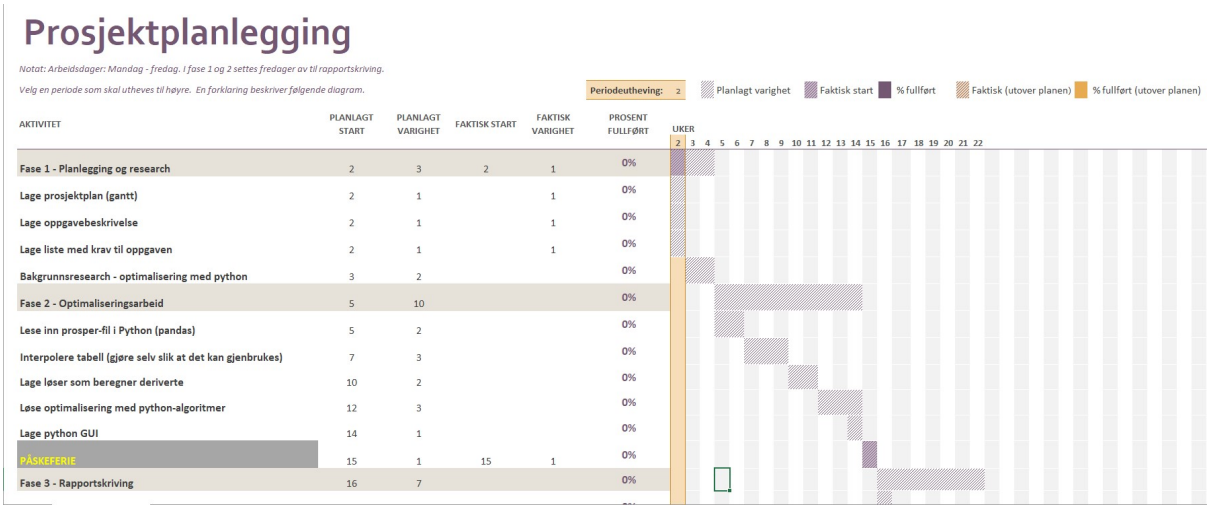

Figure 3.1: A screenshot of one part of the Gantt chart used in the planning of the master's thesis. The chart is made in Norwegian.

3.1.1 Project Execution

Throughout this project there has been regular meetings with the supervisors on Teams every other week. In addition there has been some extra physical meetings, when necessary, to discuss any difficulties in the project. For the regular meetings there have been mainly three points on the agenda:

- What has been produced since last meeting
- What should be done the next couple of weeks
- Input and questions

These meetings made sure that the progress was as planned by updating the chart presented in figure 3.1. In addition these meetings made discussions possible.

The work in this master's thesis have been done with basis in the Gantt chart in figure 3.1. However, some parts of the plan have been changed during the project. An updated Gantt chart is presented in figure 3.2. This shows if the tasks were completed within the deadline and which tasks were not prioritized due to time limitations.

As presented in figure 3.2 the first three weeks were part of phase 1 which involved planning and research of previous work. These tasks were completed on schedule. The next phase was optimization work, including reading the model-file into Python, interpolate the table and make a solver that optimize the different cases. The parts of calculating the gradients and producing a Python GUI was not prioritized because more time on the optimization was needed. The calculation of the derivative was not that important because the Scipy.optimizer made approximation of the derivatives (for some methods). This resulted in extra time spent on the interpolation and optimization (orange colors), while the other tasks were completed on schedule.

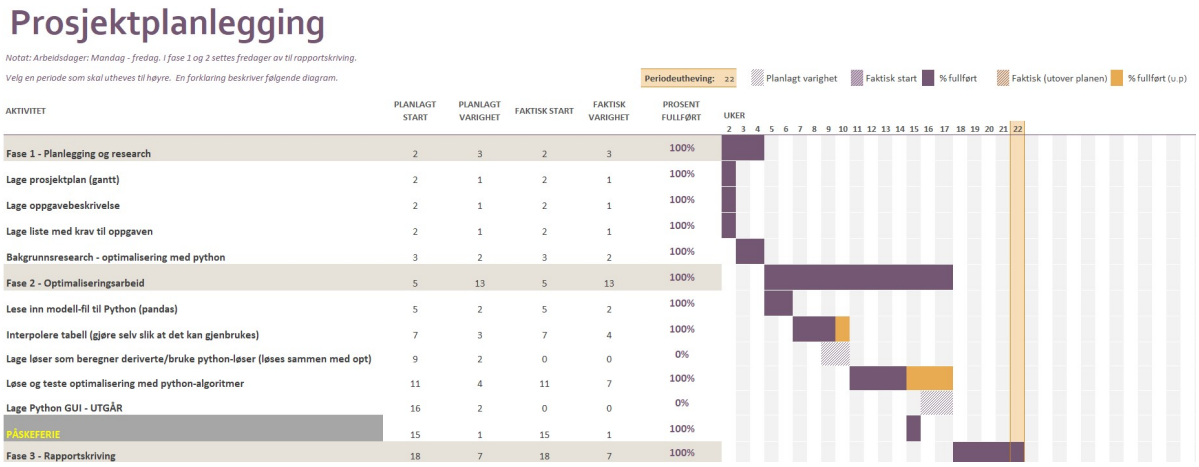

Figure 3.2: A screenshot of one part of the Gantt chart after the table is updated. The chart is made in Norwegian.

3.1.2 Research

At the start of the project it was necessary to do research on previous work on the subject. This was done to identify the gaps between previous work and the goals of this project. The research was done by searching for work done on gas lift optimization on Google Scholar, NTNU's University Library and previous master' thesis in NTNU's Open library. Some helpful resources was also handed out by the supervisors.

3.1.3 Development of Code

The process of building a code that solved the problem consisted of different parts. The process from the first meetings to the final code is summarized by the following list:

1. Define what the end goal is:
    - Make a code that optimize a well-network subject to various process constraints and give optimal GLIR as result
2. Define what tasks that have to be solved to reach the end goal.
    - Import data from the well-model into Python
    - Interpolate the data imported
    - Make different plots to understand the well-behavior
    - Make different optimization-cases in Python
    - Test the different optimization-cases and verify if they are correct
3. Solve the tasks in point 2.
4. Test if the code works and give result that matches the theory.
5. Make adjustments to the code to make it smarter and more efficient

When working with the code the main focus was to make a code that worked and solved the different tasks. When the code worked and had solved the main goal of the project the remaining time was used to improve the code. Some of the key elements that was changed to improve the code was:

- Clean the code to make it easier to understand.
- Make the objective function and the constraints dynamic so it changes when the number of wells changes, instead of multiple if/elif-statements representing each well.
- Test different ways of approximating the jacobian to find the fastest way to solve the problem. Different types of jacobian include:
  - 2-point method for approximation
  - 3-point method for approximation
  - Calculating the jacobian and give it as input to optimizer

3.2 Prosper Well-Model

The modelling of the well was done using Prosper, which offers artificial lift design and modelling capabilities for the user community. The modelling capabilities includes nodal analysis for wells with gas lift injection [19], which is used in this master's thesis.

The data for the well to be optimized was real and handed out by The well-model consists of five free variables and 24 calculated variables and is presented in a .TPD-file. The free variables are named Rates (liquid rate), Gaslift injection rate, WC, GOR, and Top node pressure and have respectively 20, 7, 12, 10 and 10 values each. This results in $20 \cdot 7 \cdot 12 \cdot 10 \cdot 10 = 168000$ rows of values for the calculated variables. The first 8 rows of result of calculated values are shown in figure 3.4, below the line with "# 4 Variable TPD Results".

```
# Number of Sensitivity Variables
4
# Numbers of :- Rates, Gaslift Gas Injection Rate values, Water Cut values, Gas Oil Ratio values, Top Node Pressure values
20, 7, 12, 10, 10
# Number of Calculated Values (columns)
24
# 5000 - Flowing Bottom Hole Pressure
# 5001 - Flowing Wellhead Temperature
# 5108 - GASLIFT - Injection Depth
# 5142 - GASLIFT - Top Node Depth
# 5143 - GASLIFT - Bottom Node Depth
# 5144 - GASLIFT - Valve Tubing Pressure
# 5145 - GASLIFT - Valve Tubing Temperature
# 5146 - GASLIFT - Valve Casing Pressure
# 5147 - GASLIFT - Casing Head Pressure
# 5148 - GASLIFT - Gas Injection Rate
# 5149 - GASLIFT - Critical Gas Injection Rate
# 5150 - GASLIFT - Critical Casing Head Pressure
# 5151 - GASLIFT - Orifice Diameter
# 5152 - GASLIFT - Thornhill-Craver DeRating Value
# 5153 - GASLIFT - GasLift Gas Gravity
# 5016 - Gauge 1 Pressure
# 5027 - Gauge 1 Temperature
# 5022 - C Factor
# 5100 - Mixture Velocity
# 5101 - Erosional Velocity
# 5102 - Maximum Grain Diameter
# 5104 - Erosion Flag
# 5154 - Erosion Rate
# 5155 - Corrosion Rate
5000, 5001, 5108, 5142, 5143, 5144, 5145, 5146, 5147, 5148, 5149, 5150, 5151, 5152, 5153, 5016, 5027, 5022, 5100, 5101, 5102, 5104, 5154, 5155
# Rate and Variable Types
# Rate Variable =  4000 - Liquid Rate
# Variable 4 =  22 - Gaslift Gas Injection Rate
# Variable 3 =  16 - Water Cut
# Variable 2 =  17 - Gas Oil Ratio
# Variable 1 =  27 - Top Node Pressure
```

Figure 3.3: First part of the well-model from Prosper showing the name of the calculated variables, among other things

One column in the results corresponds to the type presented two rows below "# Number of Calculated Values (columns)" in figure 3.3. There are 24 calculated variables, but it is only the 10 first columns that are presented in figure 3.4 (due to space limitations). In the same figure the free variables with its values are shown, from Rate Variables (Liquid rate) to Variable 1 (Top Node Pressure).

Something worth noting with the Prosper well-model is that Column 1 in the result is the WHP. In addition, the free variable "Variable 1: Top Node Pressure" is the BHP of the well.

```
4000,22,16,17,27
# Rate Values
314.491, 400.747, 510.662, 650.724, 829.202, 1056.63, 1346.44, 1715.73, 2186.32, 2785.97, 3550.09, 4523.79,
5764.55, 7345.62, 9360.34, 11927.6, 15199.1, 19367.8, 24679.9, 31449
# Variable 4 (Gaslift Gas Injection Rate) values
0, 0.354937, 0.709875, 1.41975, 2.48456, 3.54937, 4.96912
# Variable 3 (Water Cut) values
0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 99
# Variable 2 (Gas Oil Ratio) values
507.901, 598.194, 722.912, 1128.67, 1693, 2821.67, 4232.51, 5643.34, 7054.18, 8465.01
# Variable 1 (Top Node Pressure) values
1725.76, 1814.39, 1903.03, 1991.66, 2080.29, 2168.93, 2257.56, 2346.2, 2434.83, 2523.46
# Gauge 1 Depth
6590.16
# 4 Variable TPD Results
     16.0553,      40.7451,      5001.74,     -105.479,      6590.17,      1399.42,      153.559,      1399.42,      1126.02,            0,
      33.294,      42.0333,      5001.74,     -105.479,      6590.17,      1399.91,       155.02,      1399.91,      1129.22,            0,
     54.8966,       43.861,      5001.74,     -105.479,      6590.17,      1400.49,      156.633,      1400.49,      1132.98,            0,
     80.5033,       46.451,      5001.74,     -105.479,      6590.17,      1401.19,      158.461,      1401.19,      1137.51,            0,
     109.629,      49.9886,      5001.74,     -105.479,      6590.17,         1402,      160.461,         1402,      1142.87,            0,
     141.726,      54.6728,      5001.74,     -105.479,      6590.17,      1402.91,      162.567,      1402.91,      1148.99,            0,
     176.104,      60.6817,      5001.74,     -105.479,      6590.17,      1403.89,      164.699,      1403.89,      1155.74,            0,
     211.882,      68.1043,      5001.74,     -105.479,      6590.17,      1404.92,      166.777,      1404.92,      1162.85,            0,
```

Figure 3.4: Second part of the well-model from Prosper showing the name and values of the free variables and the values of the calculated variables

3.3 Read Data to Python

Before the implementation of reading data to Python is shown, some relevant theory is presented, including different data-structures and Pandas.

3.3.1 List, Numpy Array, Tuple and Dictionaries

When dealing with data processing in Python it is important to know the different data-structures and what the differences are. The different types (and most relevant for this work) are lists, arrays, tuples and dictionaries.

Lists are enclosed by square brackets [ ] and are a mutable sequences of objects. Mutable means that you can add, remove or update already existing elements in the list [25]. The list is ordered because it is a sequence, and the objects can be anything from integers to other lists or functions. Tuples are enclosed by paranthesis () and are an immutable sequence of objects. It thus consists of the same sequence of objects as for lists, but are immutable (can not change its content by removing or adding elements) [25]. Arrays are not built into Python so it is necessary to import a module to use this. These are a mutable sequence of similar type objects, so they are mutable like lists, but all objects have to be of the same type. The last datatype described in this section are dictionaries. These are enclosed by curly braces {} and are an unordered collection of key-value pairs. To be unordered means they cannot be indexed, but the values are accessed by the key. An item in an dictionary is a key-value pair [25].

3.3.2 Pandas - a Python Package for Data Analysis

Pandas is a Python package that makes Python powerful when it comes to data analysis. An object in Pandas consists of a two-dimensional tabular with labels for both rows and columns and is called a DataFrame [13, p. 4]. This is useful for reshaping, slicing and selecting subset of data, among other things. Pandas is built on top of NumPy which makes it suitable for NumPy-centric applications. NumPy is short for numerical Python and is an elementary Python package for scientific computation. This package includes the functionalities of multidimensional array object (*ndarray*), element-vise computations with arrays, linear algebra operations and many more [13, p. 4].

Series and DataFrame

Two frequently used data-structures in Pandas are *Series* and *DataFrame*, which are easy to use and works for most applications. While series are one-dimensional objects that contain data like an array, a DataFrame is a data-structure more like a spreadsheet that represents a tabular. A Series can be thought of as an ordered dictionary with fixed length, because the Series maps the index values to the data values [13, p. 113]. The DataFrame has both row and column index where the data it contains can be different types, like numeric, string or boolean [13, p. 115].

Data loading

Pandas has features that makes importing of CSV-files easy. A CSV-file is a format where the data are separated by comma, hence CSV (comma separated values) [12]. To import a CSV-file into a DataFrame (df) one can write the syntax:

$$\begin{aligned} &\text{import pandas as pd} \\ &\text{df = pd.read\_csv('file name')} \end{aligned} \tag{3.1}$$

3.3.3 Implementation

Because Python is the preferred programming language for solving the optimization problem it was necessary to read the data from well-model to Python. The code for this was made in a class in Python and called "Read_Model". The class consists of the methods "read_line", "read_rows", "read_free_variables", "read_TPD_results" and "read_all". When reading the explanations for implementation in this section and in 3.5 and 3.7 it is recommended to have a look at the code in Appendix A simultaneously.

The method "read_line" looks for the row consisting of a user-defined search-word and stores the row-number in a list. This is used in "read_free_variables" where the code stores the content in the correct row in a dictionary by using "read_rows_df". This method returns a Pandas dataframe and has two inputs; index (at which line to start importing data) and rows_imported (how many rows should the dataframe consist of/import). For "read_free_variables" the search-word is "values", because this occurs at every free variables. In the used well-model this word also occurs two times before the free variables, hence the two first elements in the list are removed. Number of rows imported here are 1, because it is only one line of interest for the free variables, as shown in figure 3.4.

The method "read_TPD_results" does the same as "read_free_variables", but for the calculated variables. For that reason the difference is the search word which is "variable tpd results" here, and the number of rows imported are unlimited (which means it imports all rows below the defined line).

"read_all" returns a list of two items: free variables in a dictionary of dataframes (one for each free variables) and the variable-results as a dataframe. The importing of the dataframe is possible because of the comma separated values (CSV)-format of the well-model.

Something that is worth noting for using this class on other well-models is that the search-words are defined within the class (and not as inputs to the methods). One should also remember that the number of elements removed in the list "self.row_numbers" are specified within the method. It is necessary to check if the search-word occurs other places than in the lines of interest when importing the data from other well-models.

3.4 Interpolation - Mathematical Formulation

Because the data described in the previous section are points and not continuous it is necessary to interpolate it. This is done so it is possible to get an approximated result for all combinations of the free

variables. The interpolation gives one value as result and is dependent on the free variables as presented in figure 3.5.

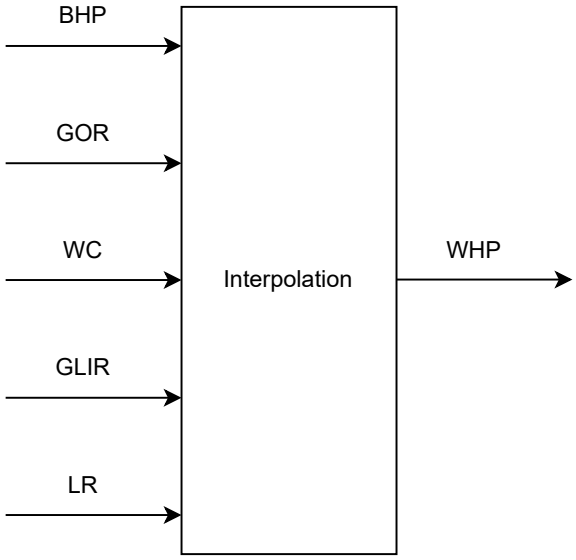

Figure 3.5: Illustration of the interpolation with five free variables and WHP as the resulting value

For the case with five free variables the interpolation can be described like this, mathematically:

$$\begin{aligned} \theta &= (\text{BHP}, \text{GOR}, \text{WC}, \text{GLIR}, \text{LR}) \\ y &= g(x, \theta) \end{aligned} \tag{3.2}$$

where θ consists of the constant inputs to the interpolating function y = WHP. In the optimization presented later, some of the constants in θ will be variables x, depending on the optimization problem.

3.5 Interpolation - Implementation

The class for interpolation converts the units and interpolates the data imported from the well-model. This is a necessary step to be able to use the data for optimization. The class consists of the methods "dict_conversion", "convert_points", "get_points", "convert_to_tup", "get_data" and "do_interpolation".

The method "dict_conversion" returns a dictionary where the keys are "Free_Variable_i" where i is the well-number. The corresponding value is a string describing the type of the variable, e.g. pressure or GLIR. The list describing what type of value it is must be defined by the user within the method. The list have the same length as number of free variables and the order from left to right is the same as how the free variables are presented from top to bottom in the well-model. This means that for the well-model used in this master's thesis the list looks like this: ['Rate values', 'GL rate', 'WC', 'GOR', 'Pressure'].

"convert_points" takes a list and different values for multiplication and addition as input, and return a list where the units are converted. The converting of a value x follows the formula in equation 3.3.

$$\text{res} = (x + \alpha) \cdot \beta + \gamma \tag{3.3}$$

where the constants α, β and γ are inputs to the method decided by the list in "dict_conversion".

The method "get_points" performs unit conversion of the points using "dict_conversion" and "convert_points", resulting in a list of 1-dimensional arrays. The method uses if-statements to choose what conversion-values to use before importing the free variables from the dictionary and converting and flatten the array. Then these are stored in a list and returned. The points are picked up with the last item in the dictionary first, so the interpolation gets correct. This is because the TPD-results are obtained with a combination of the free variables where the free variable varies from top to bottom, in the well-model-file. In scipy.interpolate.RegularGridInterpolator it is the last item in the tuple-input that vary first, and hence this procedure provides correct results.

The following values for α, β and γ are used for the different types of values:

- Rate values: BBL/day to m^3/day: $\alpha = 0$, $\beta = 0.159$, $\gamma = 0$
- Gas lift rate: MMSCF/day to m^3/day: $\alpha = 0$, $\beta = 28173.974$, $\gamma = 0$
- WC: No convertion done
- GOR: SCF/STB to Sm^3/Sm^3: $\alpha = 0$, $\beta = 0.178$, $\gamma = 0$
- Pressure: PSI to bar: $\alpha = 0$, $\beta = 0.069$, $\gamma = 1$

Because of the requirements of the regular grid interpolater, a method called "convert_to_tup" is made to convert the points in "get_points", from a list to a tuple.

"get_data" imports the data-values that represents the combination of points from "get_points" and converts it. The data is the columns in the TPD-results. Hence the method takes column-number as input (e.g. if one wants WHP as data one should have 0 as column number input) and returns a numpy array, with shape dependent of number of free variables and its lengths.

A conversion of the data is done first within the method, where it converts the two first columns, WHP and WHT (these are the only ones of interest in this master's thesis) with the following values:

- Column number 0 (WHP): PSI to bar: $\alpha = 0$, $\beta = 0.069$, $\gamma = 1$
- Column number 1 (WHT): °F to °C: $\alpha = -32$, $\beta = 0.556$, $\gamma = 0$

Because the shape of the resulting array is dependent on the number of free variables and its length there are if and elif-statements to make the correct shape. This code only accounts for 0-6 number of free variables. If the TPD-file have more free variables the message "*The number of free variables is not between 1 and 6. The code needs adjustment in the file 'interpolation' within the method 'get_data*" is returned.

The final method "do_interpolation" imports the points from "convert_to_tup" and "get_data" and use them in the the multi-dimensional interpolator "Regular grid interpolator", with a linear interpolation method. The method has column number as input and returns the interpolation result. However, to get a number out of the interpolation it is necessary to define an array of input values for the free variables/points.

3.6 Optimization Problems - Mathematical Formulation

In this master thesis there are three main optimization problems that are solved to answer the problem description in 1.1.1 in the best way possible. The last and most relevant optimization problem builds on the two presented first. In this section the mathematical formulations of the optimization problems are presented.

3.6.1 Minimize Gas Lift Injection Rate

An optimization problem where the goal is to minimize GLIR with constraint on minimum WHP is defined like this:

$$\begin{aligned} \min_{\{x\}} \quad & f(x) \\ \text{s.t.} \quad & c(x) \geq 0 \\ \text{where } & lb \leq x \leq ub \end{aligned} \tag{3.4}$$

where x is the gas lift injection rate, $f(x) = 1 * x$ and $c(x) = g(x, \theta) - y_{min}$ where $\theta = (\text{BHP}, \text{GOR}, \text{WC}, x, \text{LR})$ as described in section 3.4. y_{min} is the minimum required WHP and lb and ub are lower- and upper bounds.

3.6.2 Minimize More Complex Objective Function for One Well

This optimization problem was done to find the optimal GLIR and oil rate (OR) to minimize a more complex object function for one well. This objective function is more useful and forms the basis for extending the problem to multiple wells. Here there are two free variables in the optimization problem and in the interpolation. The optimization problem is described like this mathematically:

$$\begin{aligned} \min_{\{x\}} \quad & f(x) \\ \text{s.t.} \quad & c(x) \geq 0 \\ \text{where } & lb \leq x \leq ub \\ \text{where } & x = [x_0,\ x_1] = [\text{OR},\ \text{GLIR}] \end{aligned} \tag{3.5}$$

The objective function is given by:

$$f(x) = \omega_1 * (x_0 - \alpha)^2 + \omega_2 * (x_1 - \beta)^2 \tag{3.6}$$

$\alpha = OR_{target}$ and $\beta = GLR_{target}$. $c(x)$ is the same as in 3.6.1, but θ is different due to two variables instead of one:

$$\begin{aligned} \theta &= (BHP, GOR, WC, x_1, LR) \\ \text{where } LR &= \frac{x_0}{1 - \frac{WC}{100}} \end{aligned} \tag{3.7}$$

The objective function for this minimization problem is more complex than the one discussed in the previous section. This is made to minimize the deviation between the OR and the OR target plus the deviation between GLIR and GLIR target. There are also added two weights ω_1 and ω_2 so the user can prioritize which deviation to penalize the most.

3.6.3 Minimize More Complex Objective Function for Multiple Wells

This optimization case is similar to the one presented in 3.6.2, but with some adjustments on the objective function and the constraints. The optimization case for N number wells is given by the following formula, where $n = N - 1$ (because of 0-indexing in Python):

$$\begin{aligned} \min_{\{x\}} \quad & f(x) \\ \text{s.t.} \quad & c_1(x) \geq 0 \\ & c_2(x) \leq \text{GLR Total}_{max} \\ \text{where } & lb \leq x \leq ub \\ \text{where } & x = [x_0,\, x_1,\, ...,\, x_{n\cdot 2},\, x_{n\cdot 2+1}] = [OR_0,\, GLR_0,\, ...,\, OR_n,\, GLR_n] \end{aligned} \tag{3.8}$$

The objective function is given by:

$$f(x) = \sum_{i=0}^{i=N-1} \omega 1_i \cdot (OR_i - \alpha_i)^2 + \omega 2_i \cdot (GLR_i - \beta_i)^2 \tag{3.9}$$

and the pressure constraints $c_1(x)$ for every well i in total number of wells N are given by:

$$\begin{aligned} c_1(x) &= [c1_1(x), c1_2(x), ..., c1_N(x)] \\ c1_i(x) &= g(x, \theta_i) - y_{min_i} \\ \text{where } \theta_i &= (\text{BHP}_i, \text{GOR}_i, \text{WC}_i, x_{2\cdot i+1}, \text{LR}_i) \\ \text{where } LR_i &= \frac{x_{2\cdot i}}{1 - \frac{WC_i}{100}} \end{aligned} \tag{3.10}$$

where the constraint on total allowed GLIR $c_2(x)$ is:

$$c_2(x) = \sum_{i=1}^{i=n} x_{(2\cdot i-1)} \tag{3.11}$$

3.7 Optimization Problems - Implementation

The Python code for solving the optimization cases presented in 3.6.1 to 3.6.3 are done in a class called "*optimize*". This class separates the code for each optimization case with a line of comment describing it. The three optimization cases used in this thesis are Opt 1, Opt 3 and Opt 4 in the class. Opt 2 is another optimization which will not be described here.

Common for all the optimization problems is that they have one method for defining the objective function, one method for defining the constraint(s) and one method for running the optimization. By describing the code for the last and most complicated optimization problem the other two are covered by this. The optimization problem presented in 3.6.3 have six methods in the code, "obj_dec_mult", "presscons_dev_mult", "glr_tot_cons", "calc_jac", "calc_hess" and "min_dev_multiplewells". The first takes x, OR_target, GLR_target, w1, w2 and number of wells as input and returns the objective function. The second method takes x, BHP, GOR, WC, minimum WHP and a counter as input and returns a constraint describing minimum WHP for one well.

The method "glr_tot_cons" defines the constraint for maximum total GLIR. It takes the value of the limit and number of wells as input and returns the constraint.

The next two methods returns the jacobian and hessian for the specific objective function defined in the method "obj_dev_mult". The jacobian and hessian are calculated by hand for the specific objective

function and then inserted to these methods. It is important to notice that these does not change automatically by changing the objective function.

The last method called "min_dev_multiplewells" uses the previous methods to define the optimization problem. This adds constraints for minimum WHP dependent on number of wells. At last the optimization problem is defined by using the minimizer from Scipy.optimize. The setup of this method is done in a way so the user can choose most of the values outside of the class. However, the method for approximating the jacobian and hessian have to be defined within this method.

The input to the optimizer are the variables: algorithm, num_wells, x0, BHP, GOR, WC, presscons_val, glr_max_limit, OR_target, GLR_target, w1, w2, lb and ub. Here, algorithm is a string, num_wells and glr_max_limit are integers and the rest are lists. The length of x0, lb_var and ub_var are $2 \cdot$ num_wells and the length of BHP, GOR, WC, presscons_vals, OR_target, GLR_target, w1 and w2 are equal to num_wells.

3.8 Plotting

In the code built there is a class called "plotting", which consists of different methods for making different plots. These methods are made specific for each plot, with one method for plotting WHP versus GLIR or LR for one constant (of GLIR/LR) and one method for plotting the same but for multiple constants of GLIR/LR. The third and fourth plot uses methods for plotting oil rate versus GLIR for one or multiple WC values, while the fifth and last method plots the running time. All these plots are used in the results, presented later in this master thesis. The code in this class is not presented in detail.

3.9 Structure of the System

In this part of the thesis the structure of the system previously presented will be described using class- and flowchart diagram. This is done make give an overview of the system and so it is easier to understand the different connections.

3.9.1 Class Diagram

A Class Diagram in Unified Modeling Language (UML) is used in software engineering to describe a systems structure by showing classes, their attributes (including datatype) and method. A Class Diagram also shows the relationship between the different classes, including inheritance, simple association, aggregation, composition or dependency [35]. The Class Diagram representing the python-code used to solve the optimization case is shown i figure 3.6.

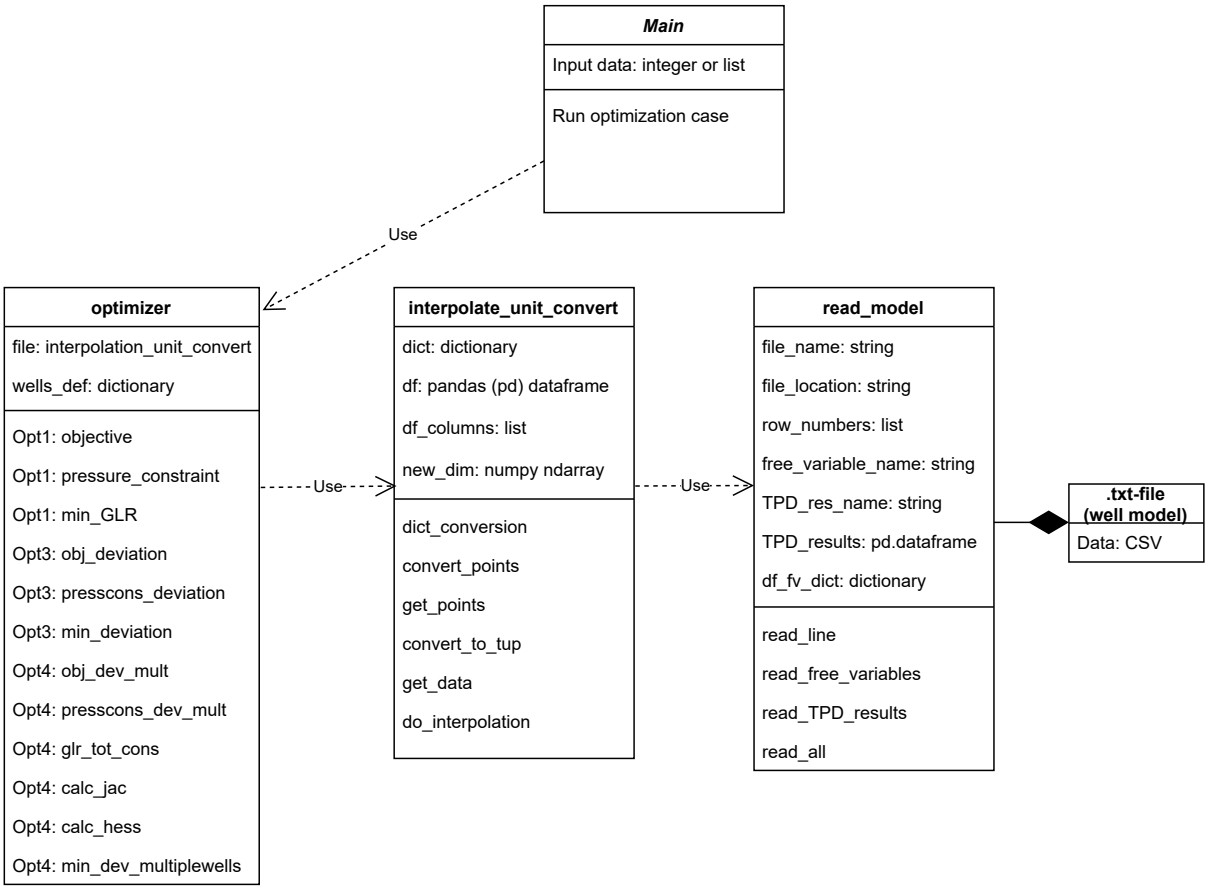

Figure 3.6: Class diagram of the optimization system showing the different classes and their attributes (and data type), and methods

As figure 3.6 shows, the optimization system consists of five classes; main, optimizer, interpolate_unit_convert, unit_model and the .txt-file. The four first have a dependency-relationship, while the relationship between read_model and .txt-file is that read_model is composition of the .txt-file.

The sixth class called 'plotting' is not shown in these diagrams because it is not that relevant to the optimization.

3.9.2 Flowchart Diagram

A flowchart is a diagram that presents the sequential order of different steps in a process. Different bodies (rectangles, ovals, diamonds etc.) represents various types of meanings, like starting point, operations, decisions, data etc. Flowcharts help visualizing a complex processes [34]. A flowchart diagram for the code solving the optimization problem is presented in figure 3.7.

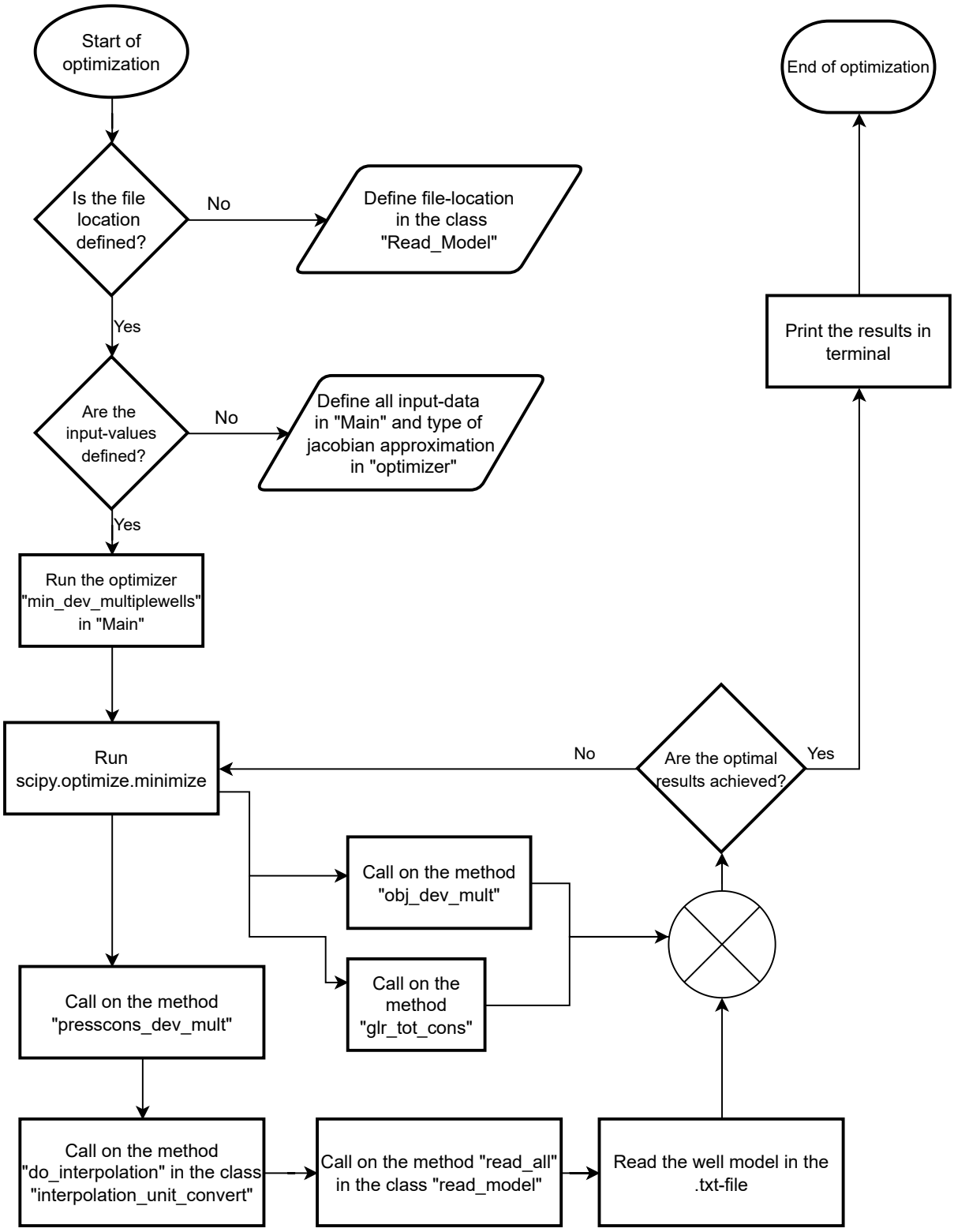

Figure 3.7: Flowchart diagram of the optimization system showing the different activities in the optimization process

The flowchart diagram shows that the steps needed before running the code for optimization, is to define the file location (and name), and define all input-data. The input data includes number of wells, WC-values for each well, max-limit for gas lift injection rate, weights and many more.

Chapter 4

Results

In this part of the thesis the results are presented to answer the problem formulation described in section 1.1.1 and the objectives presented in section 1.2. Firstly, the results about importing data to Python and interpolating it are presented with the code for calculation of derivatives. The optimization results are presented in two main parts describing different optimization cases for one and multiple wells. In the first part some characteristics for one well are presented in addition to the optimization results. For the second part the optimization results for multiple wells are described, with some results regarding running time and different weights. In every optimization case there are different parameters that represent the wells, and these are presented for each case. The two last sections in this chapter presents results on the different methods in scipy.optimize and the GUI. The results presented in this chapter are discussed in the next chapter.

4.1 Import Well-Model, Interpolate Data and Calculate Derivatives

The reading of the data from the well-model to Python is working as intended. The importing of data handles as many free and calculated variables as needed as long as the well-model have a structure like the file used in this master's thesis. This is mainly about having a search-word that occurs at each free variable and one word next to the calculated variables. The code for reading data have not been tested for other types of well-models as input, but should work as long as the file contain of comma separated values. Because this is not tested it is not possible to conclude that the reading is modular.

The class made for interpolation of the data points is working as desired. This is verified by testing with input points from the table and checking that input between two points results in a value between the values in the TPD-results in the well-model. To show that the interpolation done is correct one example will be presented. The first two lines in the TPD-results have the values:

$$\begin{aligned} \text{Line 1: WHP} &= 1.15 \text{ bar} \\ \text{Line 2: WHP} &= 2.3 \text{ bar} \end{aligned} \tag{4.1}$$

The following combination of free variables gives the BHP presented above.

$$\begin{aligned} \text{Line 1: BHP} &= 119.99 \text{ bar, GOR} = 90.46, \text{ WC} = 0, \text{ GLIR} = 0, \text{ LR} = 50\ Sm^3/day \\ \text{Line 2: BHP} &= 119.99 \text{ bar, GOR} = 90.46, \text{ WC} = 0, \text{ GLIR} = 0, \text{ LR} = 63.71\ Sm^3/day \end{aligned} \tag{4.2}$$

This shows that the only variable that changes for the two lines is LR. Note that the values presented here are not the same as shown in figure 3.4 because these values are converted to correct unit. For the interpolation to work correctly, an interpolation using BHP = 119.99, GOR = 90.46, WC = 0, GLIR = 0 and LR = 52 as input should give a WHP between 1.15 and 2.3 bar. When running the interpolator this gives 2.28 bar which indicates that the interpolation performed is correct.

4.2 Optimization for One Well

There are two different optimization cases for one well, as described in 3.6. The first optimization case is the simplest, where the goal is to find the minimum GLIR needed to obtain a minimum limit for WHP. The second optimization case has the aim of minimizing a more complex objective function as presented in 3.6.2. The latter makes the basis for solving the end goal with an optimization case for multiple wells. To understand the behaviour of the well-models and the optimization results, some characteristics for one well are presented in the following section.

4.2.1 Characteristics for One Well

The behaviour of the well-model is presented in the form of multiple plots. The following plots that are presented:

- WHP versus GLIR at different constants of liquid rate
- WHP versus LR at different constants of GLIR
- Oil rate versus GLIR at constant WHP

The first two plots (4.1 - 4.2) shows how the well perform with constant values on BHP, WC and GOR and varying GLIR and LR. The next three plots (4.3 - 4.5) presented have different combinations of BHP, WC and GOR for varying GLIR and LR. These constant values are used in the plots 4.1 - 4.2:

- BHP = 120 bar
- WC = 50 %
- GOR = 91 Sm^3/Sm^3

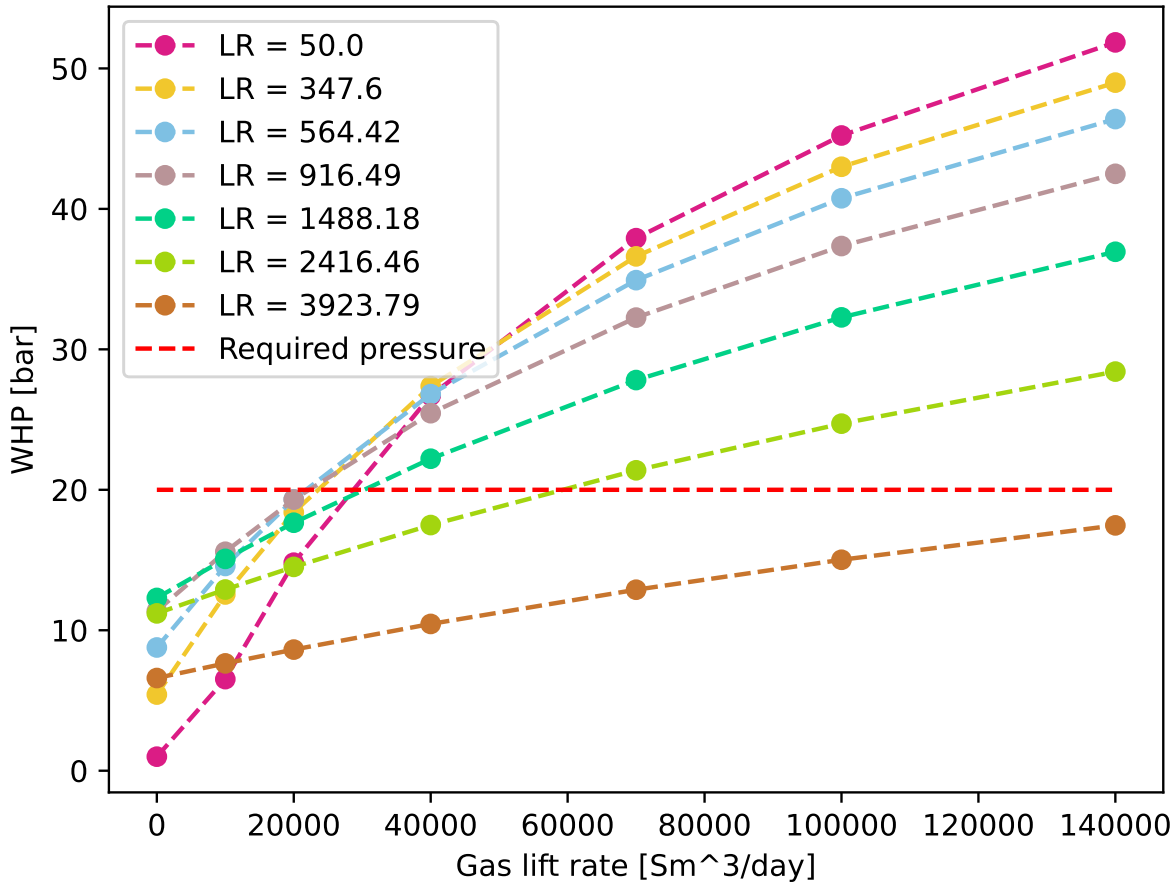

Figure 4.1: Wellhead pressure versus gas lift rate for different constants of liquid rate and the required pressure at 20 bar

The red dotted line in figures 4.1 and 4.2 represents the minimum WHP. Figure 4.1 presents that with increased gas lift rate the WHP increases. Higher LR gives lower WHP, and for LR = 3923.79 Sm^3/day there is not enough GLIR available (in the well-model) for the well to obtain WHP = 20 bar.

The plot in figure 4.2 can be divided into two parts with pivot at the stationary points (for the ones that have it) for each curve (different point for each function). The first part shows that increased LR gives increased WHP. For the second part it is the opposite, where increased LR gives decreased WHP. The figure also presents that increased GLIR gives increased WHP, as figure 4.1 also indicated.

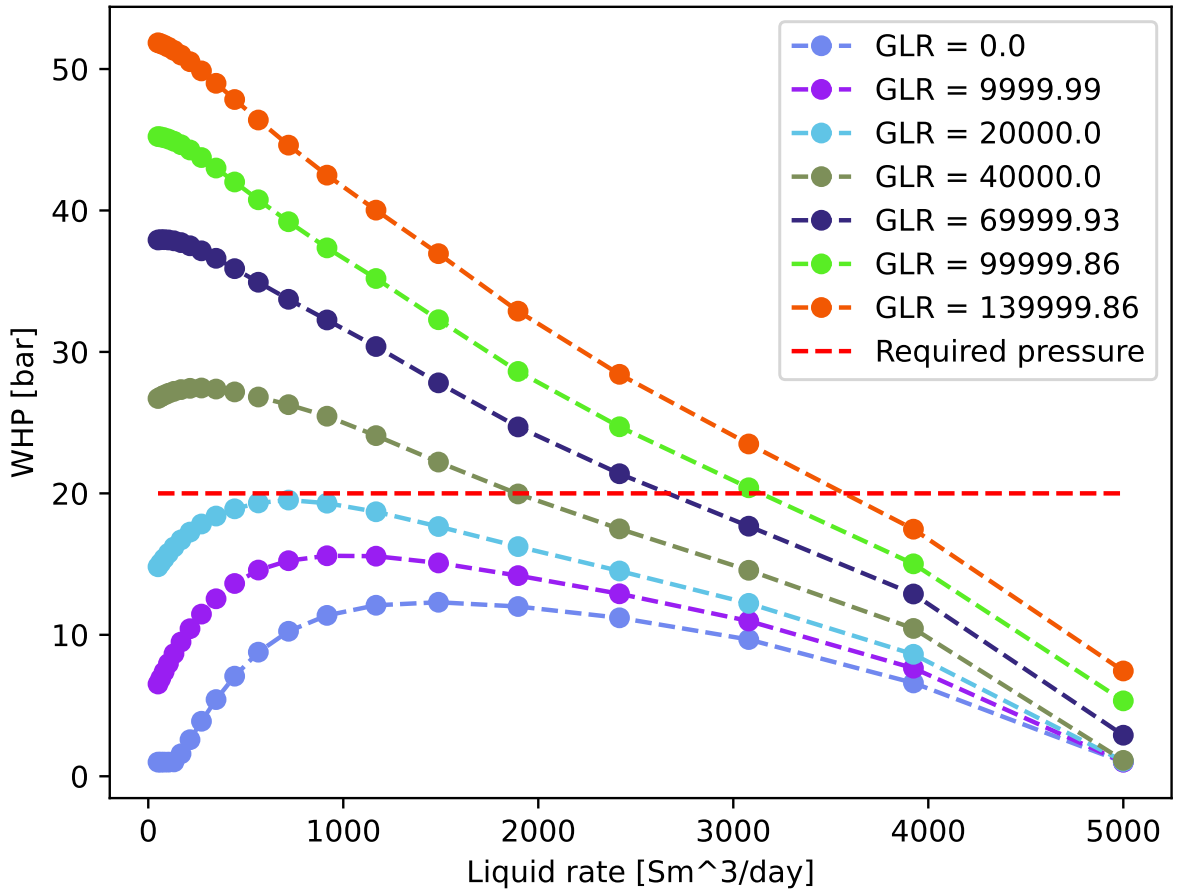

Figure 4.2: Wellhead pressure versus liquid rate for different constants of gas lift injection rate

The figures 4.3 - 4.5 shows oil rate versus GLIR for three different combinations of BHP, GOR and WC. In these plots, WHP = 20 bar. These plots are made by solving the optimization problem described in equation 3.4 for different values of the constant LR. For each iteration, the optimization problem is solved where the optimal GLIR, to achieve the minimal required WHP (the constraint), is returned. This is done with different LR at each iteration. By doing this, the connection between LR (thus oil rate) and GLIR is obtained.

opt proben is solved → optim GLIR required to ave WHP_{min}

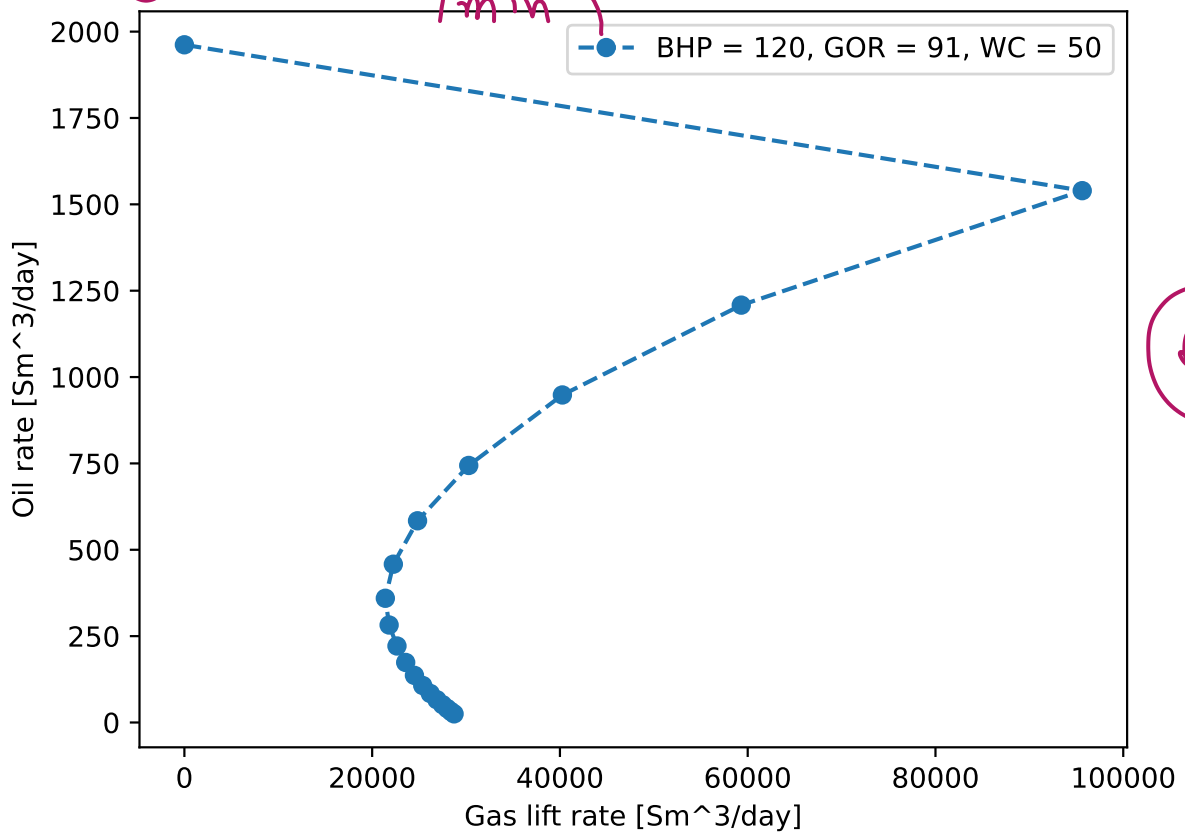

Figure 4.3: Oil rate versus gas lift injection rate for the constant values BHP = 120 bar, GOR = 91, WC = 50

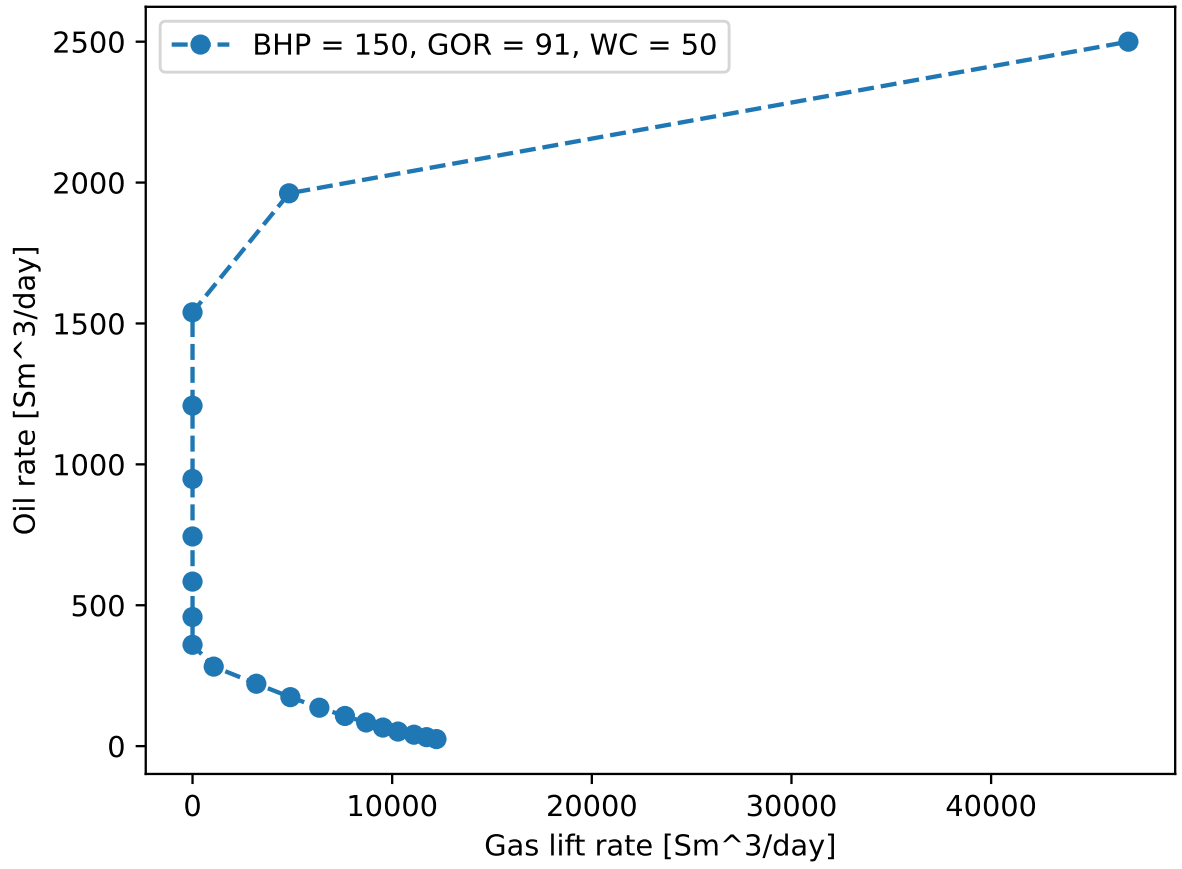

Figure 4.4: Oil versus gas lift injection rate for BHP = 150 bar, GOR = 91, WC = 50

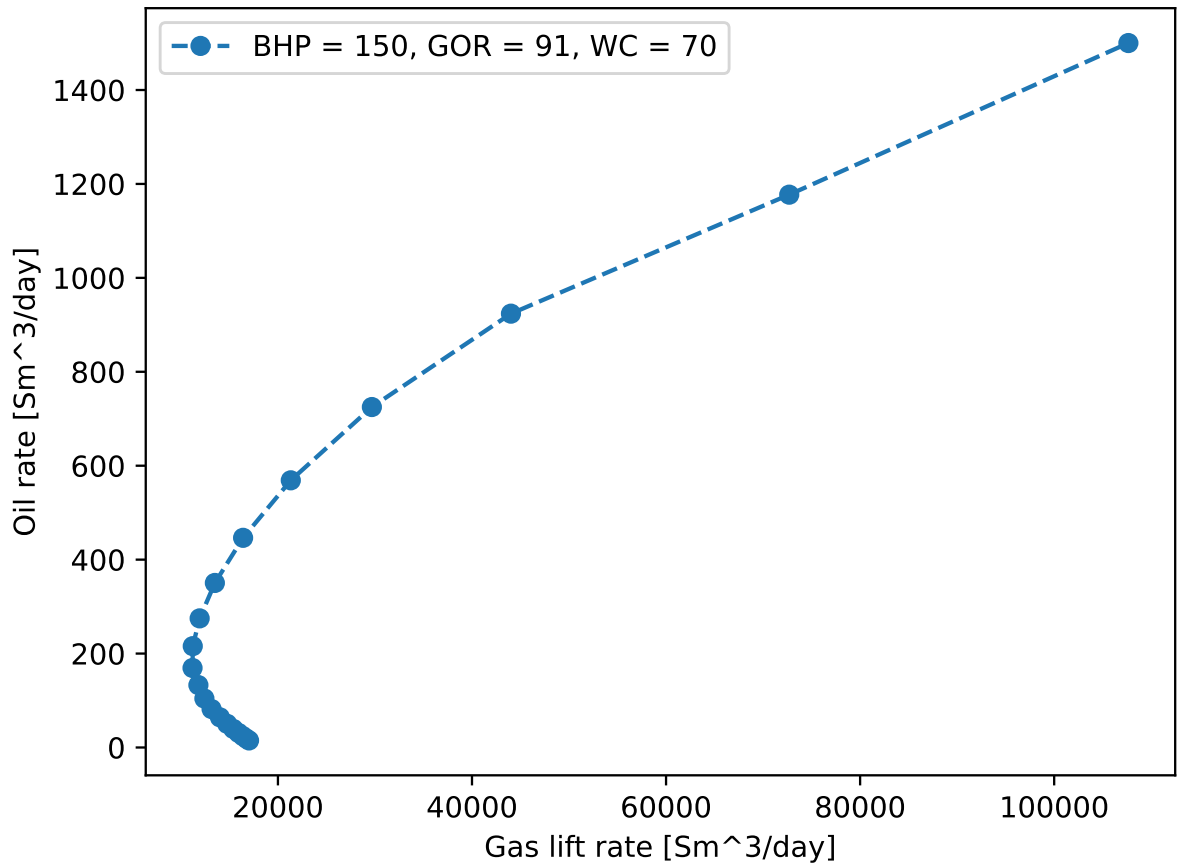

Figure 4.5: Oil rate versus gas lift injection rate for the constant values BHP = 150 bar, GOR = 91, WC = 70

Figure 4.3 shows that GLIR $\geq$ 20000 (approximately) is necessary to produce oil for this combination of BHP, GOR and WC and obtain WHP of 20 bar. When increasing BHP to 150 bar as in figure 4.4, the well is able to reach WHP of 20 bar without the use of gas lift until the oil rate becomes higher than approximately 1500 Sm^3/day. For the plot presented in figure 4.5 the BHP is 140 bar, GOR is 91 and WC is 60. For this combination of input there are required some gas lift to reach the required pressure at the beginning, and increased GLIR gives increased oil rate in the well.

4.2.2 Minimize Gas Lift Injection Rate

The first optimization case, presented in 3.6.1, was done to check that the results corresponds to the plot shown in figure 4.3. One optimization case with specific input values and corresponding optimization result is summarized in table 4.1. This optimization case is done with the goal of minimizing the GLIR with constraint on minimum WHP. GLIR is also the objective function and the variable.

Input values	**Value/name**	**Unit**
Variable	x = Gas lift injection rate	Sm^3/day
Bounds	(0, 119 880)	Sm^3/day
x_0	40 000	Sm^3/day
BHP	120	bar
GOR	91	Sm^3/Sm^3
WC	50	%
LR	2000	Sm^3/day
p_{min}	20	bar
Method	SLSQP	-
Results		
f(x)	43 499	Sm^3/day
x	43 499	Sm^3/day
nfev	16	-
nit	7	-
njev	7	-
Execution time	0,64	seconds

Table 4.1: Input values and optimization results for optimization case presented in equation 3.4

4.2.3 Minimize Objective Function for One Well

The optimization case for minimizing an objective function for one well is presented in 3.6.2. The reason for doing this was to check that solving this optimization case for one well worked as intended, before adding multiple wells to the case. The results of this optimization case can be compared to figure 4.3 which shows oil rate versus GLIR.

The results of optimization case for this problem with specific input values are summarized in table 4.2:

Input values	**Value/name**	**Unit**
Variables	x_0 = OR, x_1 = GLR	Sm^3/day
Lower bounds (lb)	[375, 25 000]	Sm^3/day
Upper bounds (ub)	[1899, 110 000]	Sm^3/day
x_0	[500, 50 000]	Sm^3/day
BHP	120	bar
GOR	91	Sm^3/Sm^3
WC	50	%
α	[1500]	
β	[80 000]	
$\omega = [\omega_0, \omega_1]$	[1, 1]	
p_{min}	20	bar
Method	SLSQP	-
Results		
f(x)	6447	$(Sm^3/day)^2$
x	[1420, 80 001]	Sm^3/day
Jacobian	[-161, 1.3]	-
nfev	23	-
nit	7	-
njev	7	-
Execution time	0,68	seconds

Table 4.2: Input values and optimization results for optimization case presented in 3.6.2

4.3 Optimization for Multiple Wells

The optimization problem that includes multiple wells use a similar objective function as used in the results in section 4.2.3. The difference is that it adds more wells to the objective function and has constraint on total gas lift used in addition. For this master's thesis the same input file is used for each well, but the wells are separated by different values for WC. The code is built so one can choose as many wells as wanted before running the optimization case.

The results from two specific optimization cases are presented first, before some results on running time for different number of wells and different weights are presented.

4.3.1 Optimization Case

An example of running this optimizer for five wells with specific inputs and the corresponding result is summarized in table 4.3. In this case the constraint on maximum total GLIR is above the sum of the GLIR_target for each well, implying that the constraint should not be active. The optimization results returned from scipy.optimize includes the jacobian, which in this case is:

$$[\partial f/\partial OR0, \partial f/\partial GLR0, \partial f/\partial OR1, ...] \tag{4.3}$$

because the variables are ordered in this way. To be able to compare these results with the theory presented in 2.4 where the jacobian is ∂OR/∂GLIR it is necessary to do some algebra to get the results on the desired shape:

$$\frac{\partial f/\partial x_{i+1}}{\partial f/\partial x_i} = \frac{\partial x_i}{\partial x_{i+1}} = \frac{\partial \text{OR}}{\partial \text{GLIR}} \tag{4.4}$$

And these are the result presented in table 4.3 and 4.4. For the two optimization cases presented in these tables, the WC-values varies from 40-60 for the five wells. The behaviour of the different wells and how oil rate changes with GLIR is presented in figure 4.6.

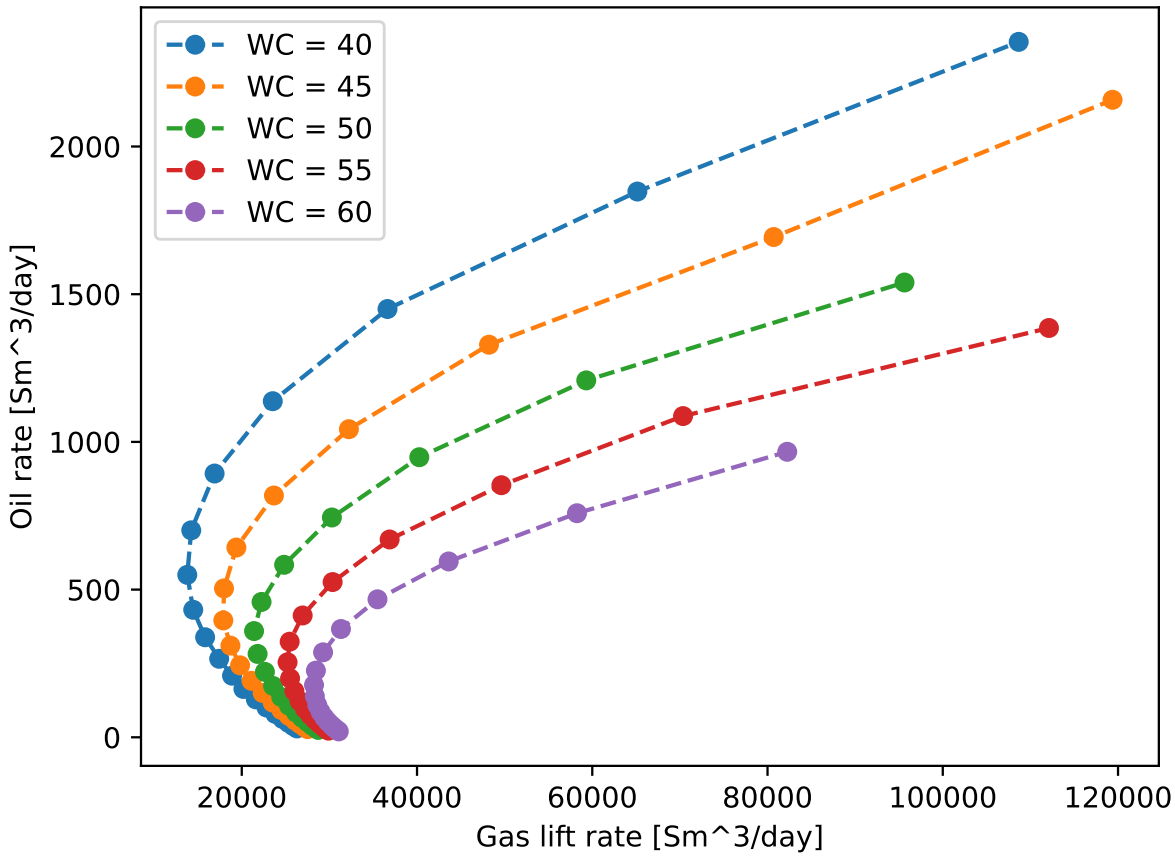

Figure 4.6: Oil vs. gas lift rate for BHP = 120 bar, GOR = 91 and different values for WC

Input values	Value/name	Unit
Variables	x = $[x_0, x_1, ..., x_9] = [OR_0, GLR_1, ..., GLR4]$	Sm^3/day
Lower bounds (lb)	[40, 0, 40, 0, 40, 0, 40, 0, 40, 0]	Sm^3/day
Upper bounds (ub)	[1899, 110000, 1899, 110000, 1899, 110000, 1899, 110000, 1899, 110000]	Sm^3/day
x_0	[500, 30000, 500, 30000, 500, 30000, 500, 30000, 500, 30000]	Sm^3/day
BHP	[120, 120, 120, 120, 120]	bar
GOR	[91, 91, 91, 91, 91]	Sm^3/Sm^3
WC	[40, 45, 50, 55, 60]	%
α	[1600, 1400, 1400, 1400, 1400]	Sm^3/day
β	[35000, 35000, 35000, 35000, 35000]	Sm^3/day
$\omega 1$	[1, 1, 1, 1, 1]	-
$\omega 2$	[1, 1, 1, 1, 1]	-
p_{min}	[20, 20, 20, 20, 20]	bar
GLR_{max}	200 000	Sm^3/day
Method	SLSQP	-
Jacobian	'2-point'	-
Results		
f(x)	1 898 119	$(Sm^3/day)^2$
x	[1418, 35004, 1107, 35006, 851, 35011, 633, 35016, 457, 35021]	Sm^3/day
Jacobian	[-363.7, 7.3, -586.9, 12.8, -1097.0, 22.5, -1533.7, 31.5, -1885.9, 41.2]	-
$(\partial x_{i+1})/(\partial x_i)$	[-0.02, -0.02, -0.02, -0.02, -0.02]	-
nfev	68	-
nit	10	-
njev	6	-
Total oil production	4466	Sm^3/day
Execution time	10.5	seconds

Table 4.3: Input values and optimization results for optimization case presented in section 3.6.3

Table 4.3 shows that the wells that have the lowest oil rate (and highest WC) are prioritized when it comes to GLIR. The optimal values of GLIR are higher for the wells with highest WC. The derivatives $(\partial x_{i+1})/(\partial x_i)$ are equal for each well.

By testing the optimizer with the same input values as presented in table 4.3 except one value, the value for maximum total gas lift injection rate, the following results presented in table 4.4 are obtained.

Input values	Value/name	Unit
GLR_{max}	150 000	Sm^3/day
Results		
f(x)	1 27 645 453	$(Sm^3/day)^2$
x	[1309, 29986, 993, 29989, 737, 29996, 515, 30006, 318, 30022]	Sm^3/day
Jacobian	[-582, -10027, -815, -10022, -1326, -10007, -1770, -9988, -2164, -9955]	-
$(\partial x_{i+1})/(\partial x_i)$	[17.2, 12.3, 7.5, 5.6, 4.6]	-
nfev	77	-
nit	11	-
njev	7	-
Total oil production	3871	Sm^3/day
Execution time	12.8	seconds

Table 4.4: Optimization results for case with same input as in table 4.3 except GLR_max is changed

In table 4.4 the GLIR used on each well is reduced because the constraint is tighter. It is still the well with highest WC that recieves most GLIR. For this optimization case the derivatives $(\partial x_{i+1})/(\partial x_i)$ are

not equal for each well. The number of function evaluations are increased and therefore the execution time as well.

4.3.2 Running Time

Running time for the optimization case is presented for two situations. One for different jacobian as input and one for how the running time varies with varying number of wells.

Different Jacobian

By running the same test as presented in table 4.3 with different inputs for the jacobian the resulting values where equal but the running time was different:

- Jacobian: '2-point' = 10,5 seconds
- Jacobian: '3-point' = 196 seconds
- Jacobian: Callable function as input to optimizer: 145 seconds
- Jacobian: None = 52 seconds

The callable function is the jacobian of the objective function, calculated by hand and inserted to a callable function in Python. The jacobian J_f of the objective function f presented in section 3.6.3 is given by:

$$J_f(x) = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_{2\cdot i}}, \frac{\partial f}{\partial x_{2\cdot i+1}}\right] \tag{4.5}$$

where i = N - 1 and N is the number of wells. When the objective function for N number of wells is:

$$f(x) = \sum_{i=0}^{i=N-1} \omega_{1_i} \cdot (x_{2\cdot i} - \alpha_i)^2 + \omega_{2_i} \cdot (x_{2\cdot i+1} - \beta_i)^2 \tag{4.6}$$

The resulting jacobian equals:

$$J_f(x) = [2 \cdot \omega_{1_0} \cdot (x_0 - \alpha_0),\ \omega_{2_0} \cdot (x_1 - \beta_0), \dots, 2 \cdot \omega_{1_i} \cdot (x_{2\cdot i} - \alpha_i),\ \omega_{2_i} \cdot (x_{2\cdot i+1} - \beta_i)] \tag{4.7}$$

This jacobian is the one used as input to the optimzer when using the "callable function".

Different Number of Wells

The complexity of the optimization problem increases as the number of wells increase. The various running times for different number of wells are presented here where the optimization case is as presented in section 3.6.3. The results presented here is made by using Jacobian = '2-point' because this was the fastest solver as presented in the previous section. In addition, the constraint is chosen so it is not an active constraint. The reasoning behind this is that when comparing running time for different wells, it is more likely to show the difference in a good way if this constraint does not impact the result different for each test, when number of wells changes. The different running times for different number of wells is as follows:

- 1 well: 0.5 seconds
- 2 wells: 1.5 seconds
- 3 wells: 5 seconds

- 4 wells: 10 seconds
- 5 wells: 13 seconds
- 6 wells: 26 seconds
- 7 wells: 47 seconds
- 8 wells: 71 seconds

WC for the test ran above had the following values:

$$WC = [WC_1, WC_2, WC_3, WC_4, WC_5, WC_6, WC_7, WC_8] = [45, 47, 49, 51, 53, 55, 57, 59] \tag{4.8}$$

For many of the cases the running time vary a lot depending on the WC-values and the constraint on GLIR. In this case the GLR_max is changed based on what $\beta \cdot$(number of wells) are, and the constraint is set just above this number so the constraint is not active. The different running times are presented in figure 4.7. The figure presents the running time for this data-set compared to an actual quadratic running time. The running time looks quadratic with respect to number of wells. There are some differences for the two but this is expected. The running time can vary a lot depending on the different input values and sometimes the solver do not converge to a solution and needs to be stopped by the user.

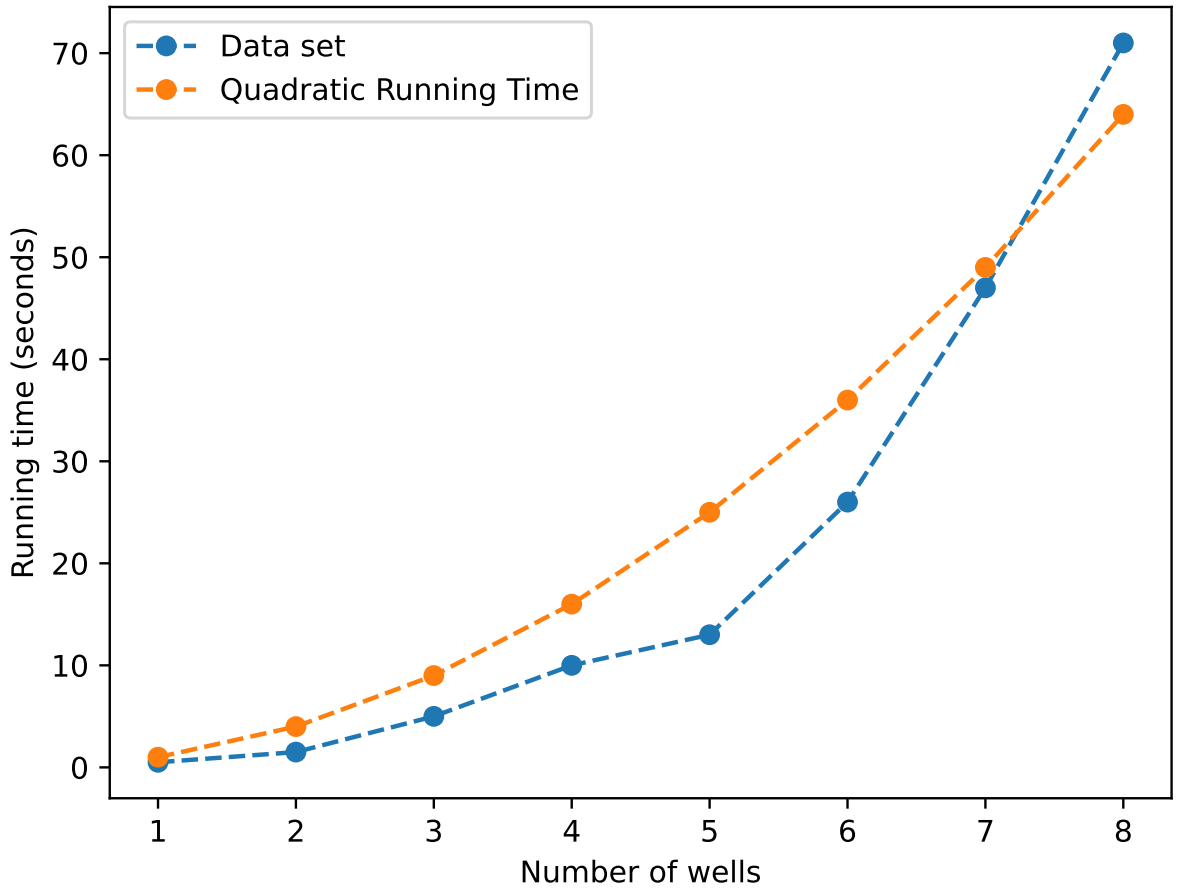

Figure 4.7: Running time for the optimization case presented in 3.6.3 for different number of wells

4.3.3 Use of Different Weights in the Objective Function

As presented in the mathematical formulation in section 3.6.3 there are some weights α and β that can be chosen to prioritize which difference to penalize the most. The differences are the difference in OR and OR target decided by α, and difference in GLIR and GLIR target decided by β. When running the the same optimization case as presented in table 4.3 but with $\alpha = 50$ instead of 1, the oil production in well 1 goes from 1418 to 1421 Sm^3/day and the GLIR is increased from 35 004 to 35 177 Sm^3/day. It is worth noting that the GLR_{max} in this case is 200 000 Sm^3/day so it is not active and there are more gas lift "available" to the field of wells.

4.4 Different Methods in Scipy.optimize

This section presents the different methods in scipy.optimize.minimize, and how they fit the optimization problem for multiple wells. Some methods are expected not to work for our problem because they cannot handle bounds or constraints. Some of these methods were presented in the theory in section **??**. All methods was tested as input to the optimizer, and the warning returned by running the code is presented in the following list, with some explanation.

- Nelder-Mead: Cannot handle constraints
- Powell: Cannot handle constraints
- CG: Cannot handle constraints
- BFGS: Cannot handle constraints
- Newton-CG: Jacobian required, cannot handle bounds/constraints
- Nelder-Mead: Cannot handle constraints
- L-BFGS-B: Cannot handle constraints
- TNC: Cannot handle constraints
- COBYLA: Cannot handle bounds
- SLSQP: This method works
- trust-constr: Have to define constraint-input to the optimizer in another way than for COBYLA or SLSQP here. Was not able to get the constraints on correct form for this type so could not compare trust-constr to SLSQP
- Dogleg: Jacobian required, cannot handle bounds/constraints
- trust-ncg: Jacobian required, cannot handle bounds/constraints
- trust-exact: Jacobian required, cannot handle bounds/constraints
- trust-krylov: Jacobian required, cannot handle bounds/constraints

For every method where the jacobian is required, these where tested with the callable function presented in section 4.3.2. All of these methods could not handle bounds or constraints so they do not fit the optimization case for multiple wells.

4.5 Graphical User Interface (GUI)

The work with implementing a GUI was not prioritized due to lack of time in the end of the master's thesis. After discussion with the supervisors it was concluded that it was more important to work on the development of the optimizer.

Chapter 5

Discussions

The results presented in chapter 4 are discussed with basis in the theory presented in chapter 2 in this part of the thesis. After the results are discussed the strengths, weaknesses and limitations of the results are presented. At last the opportunities of the work are introduced.

5.1 Discussing the Results

The first part in the chapter is the discussion of the results and how it corresponds to the theory, including characteristics for one well, minimization of GLIR for one well and minimization of the more complex objective function for one and multiple wells.

5.1.1 Characteristics for One Well

Before discussing the optimization results for one well the characteristics of the well presented in figures 4.1 - 4.5 are explained. The characteristics of the well used in the optimization problem are explained using plots of WHP versus GLIR, WHP versus liquid rate and lastly oil rate versus GLIR.

Wellhead Pressure Versus Gas Lift Rate for Constants of Liquid Rate

Figure 4.1 shows WHP versus GLIR for multiple values of liquid rate where the red dotted line represents the required pressure of 20 bar. The plot shows that with increased liquid rate in the well more gas lift is necessary to reach the required WHP. This corresponds to the theory of head loss presented in chapter 1.1.1.

Wellhead Pressure Versus Liquid Rate for Different Constants of Gas Lift Rate

Plot of wellhead pressure versus liquid rate for different constants of GLIR is presented in figure 4.2. This figure shows that if the gas lift injection rate is 20 000 Sm^3/day or lower the WHP will never reach the required pressure of 20 bar. The plot also shows that increased GLIR gives increased WHP. This corresponds with figure 4.1 discussed in the previous part. Figure 1.1 shows that increased GLIR eventually reduce net oil production rate which implies reduced WHP. For this theory to correspond with figure 4.2 the WHP would eventually decrease when increasing GLIR. This do not happen, where increased GLIR increase WHP for the whole plot. This have to do with the well-model used in this master thesis and will be explained further in the next part discussing oil rate versus GLIR.

Oil Rate Versus Gas Lift Injection Rate

The figures 4.3 - 4.5 all shows oil rate versus GLIR, for different combinations of BHP, GOR and WC, at WHP = 20 bar. There are some discussion points that are equal for all three, and something that is special for each plot. The differences between them are discussed first, before some common points are presented.

Figure 4.3 shows that for the combination BHP = 120 bar, GOR = 91 and WC = 50 there are needed approximately 21 000 Sm^3/day of gas lift to produce oil. When oil rate is between approximately 400 and 1700 Sm^3/day the plot shows behavior that corresponds with the first part of the gas lift performance curves in figure 1.1, where increased GLIR gives increased oil rate. However, the last data point breaks this pattern. This happens because when the oil rate is above approximately 1500 Sm^3/day the well is not able to reach the limit of WHP = 20 bar with the GLIR available in the well-model. This can be understood by looking at figure 4.1 where it is shown that for LR = 3923.79 Sm^3/day there is not enough gas lift to reach the limit.

The next plot shown in figure 4.4 have the combination of inputs BHP = 150 bar, GOR = 91 and WC = 50, so the BHP is increased by almost 30 bar. This makes the well reach WHP = 20 bar without use of gas lift, until the production reaches approximately 1500 Sm^3/day. After this, there is required gas lift to increase the production and maintaing WHP = 20 bar. The well can produce almost 2500 Sm^3/day of oil with approximately 50 000 Sm^3/day of gas lift, while the first well was not close to this with doubled amount of GLIR. This shows that increased BHP makes it possible to produce more oil and still reaching the required pressure, as expected when increasing BHP.

The last plot of oil rate versus GLIR is presented in figure 4.5, and is made with the values BHP = 150 bar, GOR = 91 and WC = 70. The BHP and GOR is equal to the last plot, and WC is increased. Here it is shown that the well almost always needs gas lift injection to reach the pressure limit. The well is also able to reach the pressure limit for every combination of GLIR and OR (unlike the first combination of input-values).

Common for the the three plots is that there are two values of GLIR that gives two different values of oil rate (for some interval of GLIR). This corresponds to figure 4.2 which have two values for gas lift injection rate that gives WHP = 20 bar for the functions with gas lift injection rates between 20 000 and 30 000 Sm^3/day.

The last important point to discuss about these plots is that none of them have a stationary point representing a maximum. This would be expected, as presented in figure 1.1, where the oil production first increases before it is reduced with increased GLIR. The plots are made using the well-model from the .TPD-file and all values are used. This means that the well-model does not show how the well reacts to gas lift injection rates above 139 999,9 Sm^3/day which is the last value of the free variable gas lift injection rate in the model-file presented in 3.2. This limits the optimization that will be discussed later, but it does not make it impossible. If the well-model represented behaviour as in figure 1.1 it would be a possibility to relax the bounds and constraints and verify that the optimizer found the stationary point (maximum) for each well. Because this is not possible with the well-model used in this thesis the optimization is done with bounds on the OR and GLIR. It is still possible to find an optimal gas lift injection rate with our case and constraints, but the verification of the solutions can be more difficult.

5.1.2 Minimize Gas Lift Injection Rate for One Well

By comparing the result for the minimization problem presented in table 4.1 to the plot 4.1 it indicates that the results are correct. The optimization case is done with LR = 2000 Sm^3/day. In figure 4.1 this is a line between the lime green and green lines. It looks correct that this (imaginary) line crosses the required pressure line (at 20 bar) at GLIR = 43 499 Sm^3/day. In addition, the scipy-optimize.minimize returns "Optimization terminated successfully" which indicates correct results.

5.1.3 Minimize Objective Function for One Well

As presented in table 4.2 one can see that with targets of [OR_target, GLR_target] = $[1500, 80000]$ and constraint on minimum WHP the optimal solution was $x^* = [1420, 800001]$. This shows that the deviation of GLIR is just 1 while the deviation for oil rate is 80.

It is difficult to verify that the results are correct, but one can compare the results with figure 4.3 and see that the resulting combination of oil rate and gas lift injection rate is a point on the graph which indicates that the optimizer gives correct values. When running this optimization the weights had the values 1, and was not tested with different values on the weights for this case.

The conclusion for this optimization result is that it is correct and this set-up of objective function and constraints can be used when adding multiple wells to the optimization problem.

5.1.4 Optimization for Multiple Wells

The various results for optimization for multiple wells will be discussed and compared to theory in this section. The optimization case will be discussed first before the different running times, and factors changing it, are examined.

Optimization Case

When discussing the optimization for multiple wells it is important to compare the results to the theory presented in 2.4 and verify that the gradient $\frac{\partial OR}{\partial GLR}$ is equal for every well in the optimum (as shown in equation 2.16).

The optimization is done with two free variables for each well, OR and GLIR. When these two are free, it means that the other constants to the interpolating function are constants, like BHP, GOR and WC. As mentioned in the background in section 1.1 the BHP lowers when injecting gas into a well. Because of this it could be an idea to have the BHP as a free variable in the optimization as well, but this is not done in this work. future work

For the optimization case with a constraint on total gas lift injection = 200 000 Sm^3/day presented in table 4.3 it is shown that the gradient $[\partial x_i/\partial x_{i+1}] = -0.02$ for every well i. The constraint is 200 000 Sm^3/day, where the gas lift target for each well is 35 000 Sm^3/day, which gives a total GLIR of $35000 \cdot 5 = 175000$ Sm^3/day if the target is reached. This implies that the constraint is not active. In this case the results corresponds with the theory saying that the gradient should be equal for each well.

When adding a constraint on total gas lift injection = 150 000 Sm^3/day it is below the total gas lift injection target, implying that the constraint is active (this is also verified when summing the results for gas lift injection for each well where this equals 150 000 Sm^3/day). For this case the gradient $[\partial x_i/\partial x_{i+1}]$ varies from 17.2 to 4.6 in the results. This is interesting because it does not correspond with the theory presented in 2.4. It is not easy to conclude what the reason is but one explanation is that the well models are made by using linear interpolation between the data points. This can result in different values for the jacobian in the optimum. Another reason can be that the optimization result is wrong when adding a constraint on total gas lift injection rate. It would be strange if this is the case because the optimizer returns "*Optimizer terminated successfully*" so the first explanation seems more correct. A third reason for the different jacobians can be that the theory presented in chapter 2.4 are done with a simple objective function f that represents the oil production. The objective function in this case is more complex, so it could be that the theory is not correct any more. However, this is not a very credible theory either, because the case with no active constraint provides same jacobian for each well.

The optimization results shows that the optimizer prioritize some wells over another. Table 4.4 shows that the well with lowest WC-value (40) gets the smallest amount of GLIR (29 986 Sm^3/day), while well five, with highest WC-value (60), gets GLIR of 30 022 Sm^3/day.

Running Time

The results in section 4.3.2 shows that the running time increase when the number of wells in the optimization problem increase. There is nothing strange with this as the complexity increases for each well added. The running time seems to be quadratic as presented, but the running time for each case can vary a lot (especially when number of wells becomes above five). Because the running time changes a lot it seems reasonable that the changes comes from some weird input-values, rather than wrong code for the optimization. It could also imply that the solver have convergence-issues for some combinations of input-values. The testing done in this project has not unveiled one reason for this, but a "bad" combination of input-values could be the explanation when the optimizer becomes slow. The scipy.optimize.minimize seems a bit vulnerable to some combinations of input.

Experience from running a lot of different testing of the optimizer is the background for the following discussion. There are a lot of factors that plays a role when it comes to running time, and some of these are presented in the list:

- Type of approximation of the Jacobian of objective function ('2-point'. '3-point' or callable function). This was presented in section 4.3.1 and shows that the running time can vary from 10 to 196 seconds for a case with five wells.
- The constraint on total GLIR. If this limit is too low the optimizer runs into a problem and the optimizer either uses very long time and gives the message: 'Positive directional derivative for linesearch' instead of 'Optimization terminated successfully', or runs until the program running Python (VSCode in this case) stops working and needs restart.
- Initial guess x0. The running time decreases with better initial guess x0.
- Weights. By using different values for the weights than 1, the running time clearly increases. E.g. by using $w2_2 = 10$ instead of 1 in the case presented in table 4.3, the running time increases to 58 seconds.
- Different WC-values. This is also a factor that have an impact on the running time. Sometimes a specific WC-value for a well increase the running time.
- Other reasons. Sometimes the same optimization case runs in 10 seconds one time, and 8 seconds the next time.

Use of Different Weights in Objective Function

The results presented in section 4.3.3 shows that by changing the weights, the results changes as desired. The optimizer uses this change in input and handle it correctly. The oil production increases with increased weight, by increasing the GLIR. This happens because it is more "important" to increase the oil production so the difference between OR and the OR target becomes as small as possible, to minimize the objective function. The structure of the objective function works as intended.

5.1.5 Different Methods in Scipy.optimize

As presented in the results in section 4.4 there were only two methods that could work for our optimization problem, with constraints and bounds; 'trust-constr' and 'SLSQP'. Scipy.optimize.minimize demands the constraint of 'trust-constr' to be defined as a single object or a list of object, which differs from the definition of constraint for COBYLA or SLSQP. After trying to make this work for a while I had to move on without managing to get the constraint on correct format. For this reason it was not possible to compare the two solvers against each other when it comes to convergence and running time, unfortunately. future work

5.2 Strengths

One strength of the results in this master thesis is that the code developed for gas lift optimization works good for five wells with running time around 10 seconds, and the running time increases approximately quadratic with increased number of wells. The code is also made dynamic so it works for any number of wells as input, and the objective function and constraints will adapt to this change. It is pretty easy to run the optimizer, where every input-values can be inserted to the optimizer on the same page as the code is run.

Another strength of the result is that the optimizer is able to give different gas lift injection rate to each well based on the different WC-values. In addition, the objective function is well formulated so different weights as input can easily change the result.

Some other strengths of the result is that many of the objectives are achieved. The reading of the model-file can handle any number of free- and calculated variables. The table is interpolated and the units are converted correctly by the code implemented. In addition the code solve the gas lift optimization using scipy.optimize where number of wells can vary.

5.3 Weaknesses

A weakness of the result is that there was not much time left to test the code for various input-values and compare the results. The reasoning behind why the running time sometimes increase a lot is not conclusive because the different factors were not tested against each other so it is difficult to be certain.

Another weakness of the result is the fact that I was not able to test the method 'trust-constr' against the method 'SLSQP'. It would be interesting to see the differences in performance of the two methods, but no such result was obtained.

The fact that the results for the optimization do not optimize five different wells, only the same well with different constants of WC is also a weakness. The most important was to make an optimizer that worked, and this was prioritized. Because of this the values of the results presented is not that interesting because there are not different wells as in real life. The resulting values can show if the optimizer works (as it seems to). future work

A GUI was not made in this master thesis due to lack of time. This was part of the objectives but had to be down prioritized to get the optimizer complete. future work

Another possible weakness of the result is that I am not that experienced when it comes to coding, and this was my first large coding-project. With this reservation the code is maybe not as efficient as it could have been. However, the skills were developed throughout the spring and the results should not been affected much by this.

5.4 Limitations

Most of the limitations of the results are presented previously in this thesis, but they are summarized in the following list:

- The code only accounts for up to six free variables in the model-file when interpolating the data points in Python.
- The wells are separated with different WC-value, and not with different model-files for each well as an optimization case of a real well-network would have.
- The well-model does not have a stationary point (a maximum) with the GLIR used in the model. This makes verification of the results harder and not as credible as it could have been.

5.5 Opportunities

The opportunities for for further development of this work is large. There are not much change needed to make the code ready to optimize the wells with one well-model for each well so it can represent a real case. When this is done, it will be possible to run the optimization case for different combinations of input-values that corresponds with different well-behavior over time. One could e.g. run the optimizer with different values of BHP to model how the well will produce oil in the years to come.

Chapter 6

Conclusions and Further Work

The last chapter of this master's thesis is divided into two main parts, summary and conclusions, and further work. The first part provides a list of the main results achieved in the work and some discussion on them, while the second part gives an overview over the remaining tasks and further work. These are tasks that are a natural extension of the work done. Both the conclusion and the further work are mainly presented in lists to make it easier to separate the different results, discussion points and future tasks.

6.1 Summary and Conclusion

To sum up the work done in this master's thesis some points are presented:

- The well-model was efficiently read into Python with the use of Python Pandas. The code worked well for any number of free- and calculated variables.
- The code for importing data searches for some keywords to find the correct data. Therefore the code makes some demands to the structure of the model-file (e.g. that the same word is present in the headline for all the free variables).
- The class for interpolation worked very well for for the data imported from the well-model. The multi-dimensional interpolation works good for any number of calculated variables, but the code only accounts for 0-6 number of free variables. This number can be increased by adding if-statements to the code.
- The production of a solver that calculates the derivative for any function has not been prioritized in this work. The class "Optimize" has a method that returns the jacobian for the specific objective function used for the optimization case for multiple wells.
- The work have provided three optimization cases, two for one well and one for multiple wells, where the latter is of greatest interest. The optimizer for multiple wells performs good for 1-5 wells, with running time around 10 seconds for five wells and approximately quadratic running time for increased number of wells. The different inputs, including BHP, GOR, WC, weights, oil rate-target, GLIR-target, min WHP-limit for each well and total GLIR-limit for all wells are defined before running the optimizer. All optimization cases returns optimal production rates and GLIR which seems correct with the verification methods used in this master's thesis.
- For some combination of input-values the running time suddenly increases drastically. The reason for this is not conclusive, but it is assumed that it is some combination of input that does not fit with the well-model used.

- The other methods scipy.optimize.minimize provide are not compared to SLSQP which is the one used in the results in this work. The only other method that can work for our case with constraints and bounds is the method 'trust-constr'. The definition of constraints for this are defined as a single object or a list of objects, which is different than for SLSQP. I was not able to make this work so the methods are not compared against each other.

6.2 Recommendations for Further Work

The recommendations for further work consists of two parts. One main part of further work, and one part for smaller code improvements that have been noted during the work. These code improvements are mostly for making the code smarter and more efficient.

6.2.1 Further Work

The work with this optimization problem is not finished for , and there are multiple tasks that needs to be solved before implementing the results. For further work I recommend the following tasks (where the order can be changed depending on priorities):

- Have different wells as inputs to the optimizer, and not only different WC. The code for optimization needs to be changed a bit to make this possible, but the reading and interpolation works for different wells already.
- Test with realistic and different values for BHP, GOR, WC, oil rate target, GLIR-target and GLIR-total limit.
- Test for more than eight wells and different (more realistic) values to figure out if the running time improves with this measure.
- Run the optimization problem with BHP as a free variable instead of as a constant.
- Run the case for different values that represents future well-conditions. By making a loop that runs the optimizer for these different values it would be possible to see how much gas lift and oil production that are needed in the future to minimize the objective function. One could also plot this result.
- Make a GUI to make the optimization easier to manage for the user

6.2.2 Smaller Code Improvements

The tasks presented in this section are points that I have noted during the work and wanted to finish if there was time in the end. The time was spent on other tasks, hence the list still exists. The following list presents the different tasks:

- Class 'read_model': The lines in the document to be read can be "wrapped". This means that the values on a line can continue on the next line for the free variables. The code does not account for this as it is, and only reads one line beneath the desired line. One could make the code more robust by accounting for this issue.
- Class 'read_model': Make a code that converts the .TPD-file to a .txt-file automatically when reading it. As it is now, the user have to manually open the file and save it as a .txt-file.
- Class 'interpolate_unit_convert': Make a method that calculates the bounds for the free variables after the conversion of the units are done. E.g. returns maximum and minimum value of the free variables (values above or below these are not usable as input to the interpolator).
- Class 'optimizer': The second optimization problem for one well can be made smarter, by putting all defining values as input to the method so the user can define them outside the class when

running the optimizer. However, this is not that important because the optimizer for multiple wells also works for one well, so this can be used in this case (and all the variables are input to the function).

Acronyms

BBL Barrels of crude oil per day. 23

BHP Bottom hole pressure. 1, 2, 4, 20, 22, 24–26, 30, 31, 33, 35, 44, 45, 48–50

CSV Comma separated values. 21

DFO Derivative free optimization. 11, 12

GLIR Gas lift injection rate. VI, 1, 2, 4, 13, 18, 22–26, 30–33, 35, 36, 38, 39, 41–47, 49, 50

GOR Gas-oil ratio for a stream: $GOR = \frac{\text{Gas rate}}{\text{Oil rate}}$. 4, 19, 22–26, 30, 31, 33, 35, 44, 45, 49, 50

GUI Graphical user interface. 5, 30, 42, 47, 50

KKT Kusher-Kuhn-Tucker. 13, 14

LP Linear Programming. 12

LR Liquid rate. 2, 4, 22, 24–26, 30–33

MMSCF Million standard cubic feet. 23

NTNU Norges Tekniske- og Naturvitenskapelige Universitet. I

OR Oil rate. 24, 38, 42, 44–46

PSI Pounds per square inch. 23

SCF/STB Cubic foot per barrel. 23

SQP Sequential Quadratic Programming. 5, 12, 13

UML Unified Modeling Language. 26

WC Water cut of a stream: $WC = \frac{\text{Water rate}}{\text{Water rate + oil rate}}$. VI, 4, 19, 22–26, 29–31, 33, 35, 37–39, 41, 44, 45, 47, 49, 50

WHP Wellhead pressure. VI, 1, 2, 20, 22–26, 30–33, 35, 43–45, 49

WHT Wellhead temperature. 2, 23

Bibliography

[1] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 2019.

[2] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021.

[3] Eduardo Camponogara and Paulo H.R. Nakashima. Solving a gas-lift optimization problem by dynamic programming. *European Journal of Operational Research*, 174(2):1220–1246, 2006.

[4] Håvard Devold. *Oil and gas production handbook*. ABB Oil and Gas, P.O. Box 6359 Etterstad, 2013.

[5] Kunal Dutta-Roy, Santanu Barua, and Adel Heiba. Computer-Aided Gas Field Planning and Optimization. All Days, 03 1997. SPE-37447-MS.

[6] Kunal Dutta-Roy and James Kattapuram. A New Approach to Gas-Lift Allocation Optimization. All Days, 06 1997. SPE-38333-MS.

[7] Donald F. Elger, Barbara A LeBret, Clayton T. Crowe, and John A.Robertson. *Engineering Fluid Mechanics*. John Wiley & Sons, Singapore Pte. Ltd., 2016.

[8] Ronald L Graham and F Frances Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4(4):324–331, 1983.

[9] Grant, Mitchell. Gantt Chart. `https://www.investopedia.com/terms/g/gantt-chart.asp`, 2022. Online; accessed 9 May 2022.

[10] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bereau of Standards*, 49:409–436, 12 1952.

[11] Bendik Netland Hulløen. *Project Thesis: Gas lift optimization using UniSim Design*. NTNU, 2021.

[12] S M Hasan Mahmud, Md Altab Hossin, Hosney Jahan, Sheak Rashed Haider Noori, and Touhid Bhuiyan. Csv-annotate: Generate annotated tables from csv file. In *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 71–75, 2018.

[13] Wes McKinney. *Python for Data Analysis*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, 2012.

[14] K. Millman and M. Aivazis. Python for scientists and engineers. *Computing in Science & Engineering*, 13(02):9–12, mar 2011.

[15] Manickam S. Nadar, Tim S. Schneider, Kathy L. Jackson, Calum J. N. McKie, and Javad Hamid. Implementation of a Total-System Production-Optimization Model in a Complex Gas-Lifted Offshore Operation. *SPE Production & Operations*, 23(01):5–13, 02 2008.

[16] Rit Nanda, Shashank Gupta, Himanshu Shukla, Pulkita Rohilla, Kush Patel, and Ajit Kumar Shukla. Development of intermittent gas lift optimum time module using automatic pneumatic system. *Journal of Petroleum Engineering and Technology*, 2:2231–1785, 04 2012.

[17] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Science + Business Media, LLC, 233 Spring Street, New York, NY 10013, USA, 2006.

[18] Tega Odjugo, Yahaya Baba, Aliyu Aliyu, Ndubuisi Okereke, Lekan Oloyede, and Olabisi Onifade. Optimisation of artificial lifts using prosper nodal analysis for barbra-1 well in niger delta. *Nigerian Journal of Technological Development*, 17:150–155, 10 2020.

[19] Petroleum Experts. MULTIPHASE WELL AND PIPELINE NODAL ANALYSIS. `https://www.petex.com/products/ipm-suite/prosper/`, 2022. Online; accessed 13 May 2022.

[20] PyPi. PuLP 2.6.0. `https://pypi.org/project/PuLP/`, 2022. Online; accessed 24 February 2022.

[21] Kashif Rashid, William Bailey, and Benoît Couët. A survey for gas-lift optimization. *Modelling and Simulation in Engineering*, 2012:16, 2012.

[22] Clément W. Royer, Michael. O'Neill, and Stephen J. Wright. A newton-cg algorithm with complexity guarantees for smooth unconstrained optimization. *Mathematical Programming*, 180:451–488, 03 2020.

[23] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., Department of Econometrics, Tilburg University, P.O. Box 90153, 5000 LE Tilburg, The Netherlands, 1986.

[24] Scipy Community. Community. `https://scipy.org/community/`, 2022. Online; accessed 22 May 2022.

[25] Susan Maina. Ultimate Guide to Lists, Tuples, Arrays and Dictionaries for Beginners. `https://towardsdatascience.com/ultimate-guide-to-lists-tuples-arrays-and-dictionaries-for-beginners-8d1497f9777c`, 2020. Online; accessed 26 May 2022.

[26] Benjamin Kwanen Tapley. Lecture notes in "introduction to programming and numerics (tdt4127), November 2020. Last updated 27/11/2020.

[27] The SciPy Community. Optimization and root finding. `https://docs.scipy.org/doc/scipy/reference/optimize.html`, 2022. Online; accessed 16 March 2022.

[28] The SciPy Community. scipy.interpolate.RegularGridInterpolator. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RegularGridInterpolator.html`, 2022. Online; accessed 11 March 2022.

[29] The SciPy Community. scipy.optimize.basinhopping. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.basinhopping.html#scipy.optimize.basinhopping`, 2022. Online; accessed 16 March 2022.

[30] The SciPy Community. scipy.optimize.brute. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brute.html#scipy.optimize.brute`, 2022. Online; accessed 16 March 2022.

[31] The SciPy Community. scipy.optimize.differential_evaluation. `https://docs.scipy.org/doc/scipy/reference/optimize.html`, 2022. Online; accessed 16 March 2022.

[32] The SciPy Community. scipy.optimize.minimize. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize`, 2022. Online; accessed 16 March 2022.

[33] Richart Vazquez-Roman and Pablo Palafox-Hernández. A New Approach for Continuous Gas Lift Simulation and Optimization. All Days, 10 2005. SPE-95949-MS.

[34] Visual Paradigm. Flowchart Tutorial (with Symbols, Guide and Examples). https://www.visual-paradigm.com/tutorials/flowchart-tutorial/, 2022. Online; accessed 20 May 2022.

[35] Visual Paradigm. What is Class Diagram? https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/, 2022. Online; accessed 20 May 2022.

[36] Pengju Wang, Michael Litvak, and Khalid Aziz. Optimization of Production Operations in Petroleum Fields. All Days, 09 2002. SPE-77658-MS.

[37] Stephen J. Wright. Continuous optimization (nonlinear and linear programming). 2012.

[38] Ya-xiang Yuan. Recent advances in trust region algorithms. *Mathematical Programming*, 151, 06 2015.

Appendix A

Python Code

This appendix provides the code used to solve the problem. This includes a "main" file where the optimization problem is run, a class called "Read_Model" for importing the data from the well-model to Python and a class called "interpolation" that converts units and interpolates the imported data. The class for the different optimization problems called "optimize" is also included in this appendix. The appendix is divided into one part for each class.

A.1 Main

the purpose of Main file is to give data free variables and by solving opt problem get the total oil + jacobian Qoil/QGLIR + total GLIR

```
### This is the main file for the masters thesis
import time
from optimizer import optimize

start = time.time()

## Run optimizer from this script

# Define object
opt_case1 = optimize()

# Input to optimizer
numberofwells = 5 #If this value is less than number of inputs below the result for
                                        these values in the results are x = x0
x0 = [500, 30000, 500, 30000, 500, 30000, 500, 30000, 500, 30000]
BHP = [120, 120, 120, 120, 120]
GOR = [91, 91, 91, 91, 91]
WC = [40, 45, 50, 55, 60]
presscons_val = [20, 20, 20, 20, 20]
glr_max_limit = 200000
OR_target = [1600, 1400, 1400, 1400, 1400]
GLR_target = [35000, 35000, 35000, 35000, 35000]
w1 = [50, 1, 1, 1, 1] #Weight on OR - OR_target for each well
w2 = [1, 1, 1, 1, 1] #Weight on GLR - GLR_target for each well
lb_var = [40, 0, 40, 0, 40, 0, 40, 0, 40, 0]
ub_var = [1899, 110000, 1899, 110000, 1899, 110000, 1899, 110000, 1899, 110000]

# Run optimization
res = opt_case1.min_dev_multiplewells('slsqp',numberofwells, x0, BHP, GOR, WC,
                                        presscons_val, glr_max_limit, OR_target,
                                        GLR_target, w1, w2, lb_var, ub_var)
print(res)

# Print extra information of the result
```

```
##Find total oil
Totaloil = 0
for i in range(numberofwells):
    Totaloil = Totaloil + res.x[2*i]
print(f'Total oil production is {round(Totaloil, 2)}')

## Find Jacobian dQoil/dQgaslift
der = []
for i in range(numberofwells):
    result = (res.jac[i*2 + 1])/(res.jac[i*2])
    der.append(result)
print(f'The derivative dQoil/dQgaslift for the wells are {der}')

## Print total gas lift injection used:
TotalGL = 0
for i in range(numberofwells):
    TotalGL = TotalGL + res.x[2*i + 1]
print(f'Total gas lift injection used is {round(TotalGL, 2)}')

end = time.time()

print(f'Execution time is {end - start}')
```

A.2 Read Model

```
## This file is to read the .txt file to python using pandas
import pandas as pd

class Read_Model:
    def __init__(self, file_name, file_location):
        self.file_name = file_name
        self.file_location = file_location
        self.row_numbers = []
        self.free_variable_name = 'values'
        self.TPD_res_name = 'variable tpd results'
        self.TPD_results = pd.DataFrame()
        self.df_fv_dict = {}

    def read_line(self, search_word):
        with open(f'{self.file_location}{self.file_name}') as openfile:
            i = 0
            self.row_numbers.clear()
            for line in openfile:
                i += 1
                if search_word in line.lower():
                    self.row_numbers.append(i)
            return self.row_numbers

    def read_rows_df(self, index, rows_imported):
        return pd.read_csv(f'{self.file_location}{self.file_name}', skiprows=self.
                                                    row_numbers[index], nrows=
                                                    rows_imported, header=None) #4
                                                    Variable TPD results starts at row
                                                    214

    def read_free_variables(self):
        self.read_line(self.free_variable_name)     # find lines with the "values" in it
                                                      and remove the first two lines
        n = 2
        del self.row_numbers[:n]
        i = 0
        for i in range(len(self.row_numbers)):
            self.df_fv_dict['Free_variable_'+str(i + 1)] = self.read_rows_df(i, 1)
        return self.df_fv_dict
```

```
    def read_TPD_results(self):
        self.read_line(self.TPD_res_name)
        self.TPD_results = self.read_rows_df(0, None)
        return self.TPD_results

    def read_all(self):
        free_var = self.read_free_variables()
        TPD_res = self.read_TPD_results()
        return free_var, TPD_res

## Inputs
text_file_1 = '\A06_FromDHG2WH.txt' # Has to add a \ at the start
file_location_1 = r'C:\Users\Bendik Hulloeen\Documents\Skole\Master\Masters_thesis'

M1_file1 = Read_Model(text_file_1, file_location_1) #Define object
file1_read = M1_file1.read_all()
```

A.3 Interpolation

```
# This interpolator includes convertion of the units

from scipy import interpolate as irp
from read_model import file1_read
import numpy as np

class interpolation:
    def __init__(self):
        self.dict = file1_read[0]
        self.df = file1_read[1]
        self.df_columns = []
        self.new_dim = []

    def dict_convertion(self):
        #The order of unit_list is decided after free variable in well-model from top to
                                                    bottom.
        unit_list = ['Rate values', 'GL rate', 'WC', 'GOR', 'Pressure'] #Have to have
                                                   same length as number of free
                                                   variables.
        dict_conv = {}
        for i in range(len(self.dict)):
            dict_conv['Free_variable_'+str(i + 1)] = unit_list[i]
        return dict_conv

    def convert_points(self, list_2bconverted, f1_add, f2_mult, f3_add):
        list = []
        for i in range(np.size(list_2bconverted[0])):
            res = (list_2bconverted[0][i] + f1_add)*f2_mult + f3_add #Unit for
                                                        converting Liquid rate
            res_2 = round(res, 2)
            list.append(res_2)
        list2arr = np.array(list)
        return list2arr

    def get_points(self):
        l = []
        dict_conversion = self.dict_convertion()
        for i in range(len(self.dict)):
            points_dict = []
            df_to_arr = []
            df_to_arr_converted = []
            if dict_conversion['Free_variable_'+str(len(self.dict) - i)].lower() == '
                                                        rate values':
```

```
            points_dict = self.dict.get('Free_variable_'+str((len(self.dict)) - i))
                                                    #Get points from the
                                                    dictionary they're stored in
                                                    . Have to get the last
                                                    element first in this list
                                                    for the interpolation to be
                                                    correct
            df_to_arr = points_dict.to_numpy() #Converts df to numpy array
            df_to_arr_converted = self.convert_points(df_to_arr, 0, 0.158987294928,
                                                    0) #Multiplication for rate
                                                    values unit conversion
            new_arr = df_to_arr_converted.flatten()
        elif dict_conversion['Free_variable_'+str(len(self.dict) - i)].lower() == '
                                                gl rate':
            points_dict = self.dict.get('Free_variable_'+str((len(self.dict)) - i))
                                                    #Get points from the
                                                    dictionary they're stored in
                                                    . Have to get the last
                                                    element first in this list
                                                    for the interpolation to be
                                                    correct
            df_to_arr = points_dict.to_numpy() #Converts df to numpy array
            df_to_arr_converted = self.convert_points(df_to_arr, 0, 28173.
                                                    97429124846, 0) #
                                                    Multiplication for GL rate
                                                    unit conversion
            new_arr = df_to_arr_converted.flatten()
        elif dict_conversion['Free_variable_'+str(len(self.dict) - i)].lower() == '
                                                wc': #NO conversion for WC
            points_dict = self.dict.get('Free_variable_'+str((len(self.dict)) - i))
                                                    #Get points from the
                                                    dictionary they're stored in
                                                    . Have to get the last
                                                    element first in this list
                                                    for the interpolation to be
                                                    correct
            df_to_arr = points_dict.to_numpy() #Converts df to numpy array
            new_arr = df_to_arr.flatten()
        elif dict_conversion['Free_variable_'+str(len(self.dict) - i)].lower() == '
                                                gor':
            points_dict = self.dict.get('Free_variable_'+str((len(self.dict)) - i))
                                                    #Get points from the
                                                    dictionary they're stored in
                                                    . Have to get the last
                                                    element first in this list
                                                    for the interpolation to be
                                                    correct
            df_to_arr = points_dict.to_numpy() #Converts df to numpy array
            df_to_arr_converted = self.convert_points(df_to_arr, 0, 0.
                                                    17810760667903525, 0) #
                                                    Multiplication for GOR unit
                                                    conversion
            new_arr = df_to_arr_converted.flatten()
        elif dict_conversion['Free_variable_'+str(len(self.dict) - i)].lower() == '
                                                pressure':
            points_dict = self.dict.get('Free_variable_'+str((len(self.dict)) - i))
                                                    #Get points from the
                                                    dictionary they're stored in
                                                    . Have to get the last
                                                    element first in this list
                                                    for the interpolation to be
                                                    correct
            df_to_arr = points_dict.to_numpy() #Converts df to numpy array
            df_to_arr_converted = self.convert_points(df_to_arr, 0, 0.0689475729, 1)
                                                     #Multiplication for top
                                                    node pressure unit
```

```
                                                        conversion (and addition of
                                                        1)
                new_arr = df_to_arr_converted.flatten()
            l.append(new_arr)
        return l

    def convert_to_tup(self):
        list_input = self.get_points()
        self.convert_tuple = tuple(list_input)
        return self.convert_tuple

    def get_data(self, col_numb): #Get data to the points defined: Columns in TPS Res.
                                              This data should be on the regular grid
                                              in n dimensions (by def for interpolator
                                              )
        self.pnts = self.get_points()
        for i in range(len(self.df.columns)):
            col = self.df[i].to_numpy()
            #The conversion under only works for col=0 or col=1: If not, the conversion
                                                    is not done
            if col_numb == 0: #If col = 0: assumes this means pressure. The con
                res = col*0.0689475729 + 1
            elif col_numb == 1:
                res = (col - 32)*5/9
            else:
                res = col
            self.df_columns.append(res)
        # If statement with many cases for new shapes (depending on number of free
                                                  variables). This code accounts for:
                                                  0 <= free variables <= 6 and returns
                                                   error if above
        if len(self.pnts) == 0:
            return "The number of free variables needs to be at least 1"
        elif len(self.pnts) == 1:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0])))
            return self.new_dim
        elif len(self.pnts) == 2:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0]), len(self.pnts[1])))
            return self.new_dim
        elif len(self.pnts) == 3:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0]), len(self.pnts[1]), len(
                                                    self.pnts[2])))
            return self.new_dim
        elif len(self.pnts) == 4:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0]), len(self.pnts[1]), len(
                                                    self.pnts[2]), len(self.pnts[3])
                                                    ))
            return self.new_dim
        elif len(self.pnts) == 5:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0]), len(self.pnts[1]), len(
                                                    self.pnts[2]), len(self.pnts[3])
                                                    , len(self.pnts[4])))
            return self.new_dim
        elif len(self.pnts) == 6:
            self.new_dim = np.reshape(self.df_columns[col_numb], newshape=(len(self.pnts
                                                    [0]), len(self.pnts[1]), len(
                                                    self.pnts[2]), len(self.pnts[3])
                                                    , len(self.pnts[4]), len(self.
                                                    pnts[5])))
            return self.new_dim
        else:
```

```
            return "The number of free variables is not between 1 and 6. The code needs
                                                    adjustment in the file '
                                                    interpolation' and function '
                                                    get_data'."

    #Intepolate using linear interpolation
    def do_interpolation(self, col_number):
        self.pnts = self.convert_to_tup()
        self.data = self.get_data(col_number)
        self.result = irp.RegularGridInterpolator(points=self.pnts, values=self.data,
                                                  method="linear")
        return self.result
```

A.4 Optimize

```
#This file defines the class for optimizing
from scipy import optimize as sciopt
from interpolate_unit_convert import interpolation as int
from scipy.optimize import Bounds as bnd
import numpy as np
import time

#start = time.time()

class optimize:
    def __init__(self):
        self.file = int()
        self.wells_def = {}

    ## Opt 1: Minimize GLR to reach minimum WHP
    def objective(self, x):
        GLrate = x[0]
        return GLrate

    def pressure_constraint(self, x, LR, BHP, GOR, WC, pressure_cons_val): #Constraint
                                                  on minimum 20 bar pressure
        GLR = x[0]
        input_val = (BHP, GOR, WC, GLR, LR) #TNP, GOR, WC and LR are set constant to
                                                    find optimal GLrate to minimize WHP.
                                                     Values: 120, 91, 50, GLR, 2000
        interpol = self.file.do_interpolation(0) #Col 0 = WHP
        res = interpol(input_val)
        cons = res - pressure_cons_val
        return cons

    def min_GLR(self, x0, LR, BHP, GOR, WC, pressure_cons_val):
        fun = self.objective
        cons_fun = self.pressure_constraint #Fix this after editing def constraint with
                                                    more inputs
        bnds = [(0, 119880)] #bounds on GLR = 138 000 old bound
        cons = ({'type': 'ineq', 'fun': cons_fun, 'args': [LR, BHP, GOR, WC,
                                                    pressure_cons_val]})
        res = sciopt.minimize(fun, x0, method='slsqp', bounds=bnds, constraints=cons)
        return res

    ## Opt 2: Not presented in this thesis, not relevant

    ## Opt 3: minimize deviation with two free variables, one well
    def obj_deviation(self, x, OR_target, GLR_target, w1, w2):
        OR = x[0] #Possible OR
        GLR = x[1] #Possible GLR
        fun = w1*((OR - OR_target)**2) + w2*((GLR - GLR_target)**2)
        return fun
```

in the call to minimize() pass five arguments
1.objective function
2.X0 initial guess;random value
3.args
4.constraints
5.bounds

```
    def presscons_deviation(self, x, BHP, GOR, WC, pressure_cons_val): #Constraint on
                                                minimum WHP = X bar
        OR = x[0]
        GLR = x[1]
        LR = OR/(1-(WC/100)) #LR expressed by OR (the free variable)
        input_val = (BHP, GOR, WC, GLR, LR) #TNP, GOR, WC are constants
        interpol = self.file.do_interpolation(0) #Col 0 = WHP
        res = interpol(input_val)
        cons = res - pressure_cons_val
        return cons

    def min_deviation(self, x0, BHP, GOR, WC, presscons_val, OR_target, GLR_target, w1,
                                                w2):
        func = self.obj_deviation
        cons_fun = self.presscons_deviation
        lb = [375, 25000]
        ub = [1899, 110000]
        bnds = bnd(lb, ub)
        cons = ({'type': 'ineq', 'fun': cons_fun, 'args': [BHP, GOR, WC, presscons_val]}
                                                )
        argu = (OR_target, GLR_target, w1, w2)
        res = sciopt.minimize(func, x0, args=argu, method='slsqp', bounds=bnds,
                                                constraints=cons)
        return res

    ## Opt 4: minimize deviation with two free variables for multiple wells
    def obj_dev_mult(self, x, OR_target, GLR_target, w1, w2, num_wells): #same input as
                                                in opt 3, but now these are lists.
        fun = 0
        for i in range(num_wells):
            res = w1[i]*((x[2*i] - OR_target[i])**2) + w2[i]*((x[2*i + 1] - GLR_target[i
                                                ])**2)
            fun = fun + res
        return fun

    def presscons_dev_mult(self, x, BHP, GOR, WC, pressure_cons_val, counter, num_wells)
                                                : #same input as in opt 3, but here the
                                                input are lists
        LR = x[2*counter]/(1-(WC[counter]/100)) #LR expressed by OR (the free variable)
        GLR = x[2*counter + 1]
        input_val = (BHP[counter], GOR[counter], WC[counter], GLR, LR) #TNP, GOR, WC are
                                                constants
        interpol = self.file.do_interpolation(0) #Col 0 = WHP
        res = interpol(input_val)
        cons = res - pressure_cons_val[0]
        return cons

    def glr_tot_cons(self, x, glr_max_limit, num_wells):
        glr_tot = 0
        for i in range(num_wells):
            glr_tot = glr_tot + x[2*i + 1]
        res = glr_max_limit - glr_tot
        return res

    def calc_jac(self, x, OR_target, GLR_target, w1, w2, num_wells): #Calculated from
                                                the objective function
        jac_list = []
        for i in range(num_wells):
            der1 = 2*w1[i]*(x[2*i] - OR_target[i]) # dfun/dx[i], OR for i = 0, 2, 4, 6,
                                                8
            der2 = 2*w2[i]*(x[2*i+1]-GLR_target[i]) # dfun/dx[j], GLR for j = 1, 3, 5, 7
                                                , 9
            jac_list.append(der1)
            jac_list.append(der2)
        return jac_list
```

```
    def calc_hess(self, x, OR_target, GLR_target, w1, w2, num_wells): #Calculated from
                                                the jacobian. Needs x, OR_target,
                                                GLR_target as inputs even if they are
                                                not used
        hess_arr = np.zeros((2*num_wells, 2*num_wells), int)
        hess_list = []
        for i in range(num_wells):
            der1 = 2*w1[i]
            der2 = 2*w2[i]
            hess_list.append(der1)
            hess_list.append(der2)
        np.fill_diagonal(hess_arr, hess_list)
        return hess_arr

    def min_dev_multiplewells(self, algorithm, num_wells, x0, BHP, GOR, WC,
                                                presscons_val, glr_max_limit, OR_target,
                                                 GLR_target, w1, w2, lb, ub):
        func = self.obj_dev_mult
        cons_fun_press = self.presscons_dev_mult
        cons_fun_glr = self.glr_tot_cons
        jacobian = self.calc_jac
        hessian = self.calc_hess
        bnds = bnd(lb, ub)
        cons1 = {'type': 'ineq', 'fun': cons_fun_glr, 'args': [glr_max_limit, num_wells]
                                                    } #Constraint on maximum GLR
        cons = [cons1]
        for i in range(num_wells):
            counter = i
            presscons = {'type': 'ineq', 'fun': cons_fun_press, 'args': [BHP, GOR, WC,
                                                        presscons_val, counter,
                                                        num_wells]}
            cons.append(presscons)
        argu = (OR_target, GLR_target, w1, w2, num_wells)
        res = sciopt.minimize(func, x0, args=argu, method=algorithm, jac='2-point', hess
                                                    =None, bounds=bnds, constraints=cons
                                                    )
        return res
```