Abstract

The work in this master's thesis aims to solve the problem of maximizing oil production, subject to various constraints like pressure, gas lift injection rate and other process variables. The well-model describing the behaviour of one well, when injecting gas lift, is delivered by Prosper. This is a software for multi-phase well and pipeline analysis. The optimization is done using Python algorithms, which gives rise for some work for importing and manipulating the data. The well-model consists of some free variables like bottom hole pressure, gas oil ratio, water cut, gas lift injection rate and liquid rate, and some calculated variables that depends on the free variables. These calculated variables include wellhead pressure and temperature. Before optimizing the data it is necessary to import it into Python, convert the units and interpolate the data-points.

A code that reads the well-model into Python in an efficient manner has been produced in this work. It accounts for and can handle any number of free and calculated variables. The code for reading data to Python have some requirements to the form of the well-model-file, to work optimally. A multidimensional interpolation of the data-points from the well-model are performed by running the class for interpolation. The code also converts units to follow correct Norwegian standard. The code for interpolation accounts for 0-6 number of free variables, but as many calculated variables as necessary are handled. Number of free variables can be increased by adding some lines of code in the method.

Three optimization problems have been solved throughout the work with this thesis, including two cases for one well and a more complex optimization case for multiple wells. The optimization problems for one well was solved to verify that the optimizer used, scipy.optimize.minimize, gave results that corresponded with plots made for the well-behaviour (wellhead pressure versus gas lift injection rate, e.g.). The optimization case for multiple wells can handle any number of wells, and was solved to represent a real well network and investigate the performance. Each well was represented by the same well-model, but they were separated by different constants for water cut.

The results of the work are that both the importing of the well-model to Python, and the multidimensional interpolation of the data-points are done in an efficient and satisfying way. The two optimization cases for one well show good performance and the results are verified sufficiently. When it comes to optimization for multiple wells, the results show good performance for 1-5 wells. The running time increases approximately quadratic with increased number of wells, where the case of five wells has running time of approximately 10 seconds. Number of wells tested in this work is eight. The resulting values also seem reasonable and correct. There are some uncertainty when the constraint on total gas lift injection tightens, where the results does not agree with the theory.

The solver made can be used to test different optimization cases, where it is easy to manage the input variables for the user, and has potential to be used for an optimization case with real well-models for each well.

Sammendrag

Arbeidet utført i denne masteroppgaven har hatt som mål å maksimere produksjon av olje innenfor begresninger på trykk, gassløftrate og andre prosessbegrensninger. Brønn-modellen, som beskriver oppførselen til en brønn når det blir utført gassløft, er levert av Prosper. Dette er en programvare for analyse av flerfasestrømning i brønner og rørledninger. Optimaliseringen blir utført ved brukt av Python algoritmer, noe som gir grunnlag for arbeid med å importere og manipulere data levert fra Prosper. Brønnmodellen består av noen frie variabler som bunnhullstrykk, forholdet mellom gass og olje, vannkutt, gassløft injeksjonsrate og væskerate. I tillegg er det noen utregnede variabler i filen, som brønnhodetrykk og -temperatur. Før dataen er klar for optimalisering blir den importert til Python og interpolert. Før interpoleringen blir det også utført enhetskonvertering av dataen.

Gjennom arbeidet med denne oppgaven er det produsert en Python-kode som leser inn brønnmodellen til Python på en effektiv måte. Koden takler så mange frie og utregnede variabler som ønsket. Den stiller allikevel noen krav til formen på filen, for å sikre at riktige verdier blir lest inn. Ved å kjøre koden for interpolasjon blir den flerdimensjonale interpolasjonen av datapunktene utført. Koden utfører og enhetskonvertering for at man kan få resultater med korrekt norsk standard for verdiene. Interpolasjonen tar høyde for at antall frie variabler er mellom 0 og 6, mens antall utregnede variabler kan være så mange som nødvendig. Antall frie variabler tillatt kan økes ved å endre litt på koden som er skrevet, dersom dette er nødvendig.

Tre optimaliseringsproblemer er løst gjennom dette arbeidet. Det inkluderer to optimaliseringersproblemer for en brønn, og et problem for flere brønner. Problemene for en brønn ble løst for å sikre at optimaliseringsløseren, scipy.optimize.minimize, ga resultat som stemte overens med grafer for oppførselen til brønnen (for eksempel brønnhodetrykk mot gassløft injeksjonsrate). For å representere et reelt nettverk av brønner og undersøke ytelsen, ble det laget en kode for optimalisering av flere brønner. Koden som ble utviklet takler så mange brønner som nødvendig. Brønnmodellen for en brønn ble også brukt i dette optimaliseringsproblemet, men for å skille på de forskjellige brønnene ble det brukt forskjellige verdier for vannkutt.

Resultatet av arbeidet er at både importeringen av brønn-modellen og den flerdimensjonale interpolasjonen ble utført effektivt og med tilfredsstillende ytelse. De to optimaliseringsproblemene for en brønn ble løst på en god måte hvor resultatene stemte overens med teori og grafer. For optimaliseringsproblemet med flere brønner viser resultatene god ytelse for 1-5 brønner. Kjøretiden øker tilnærmet kvadratisk med økende antall brønner, hvor problemet med fem brønner har kjøretid på omtrent 10 sekunder. Maksimalt antall brønner testet i dette arbeidet er åtte. De resulterende verdiene for optimal gassløftrate og oljeproduksjon stemte også overens med grafer og teori. Det er derimot noe usikkerhet tilknyttet optimaliseringsproblemet når grensen for total gassløft-injeksjonsrate innskrenkes, hvor resultatene ikke stemmer helt med teorien.

Optimaliseringsløseren produsert kan blir brukt til å teste forskjellige optimaliseringsproblem. Denne er brukervennlig ettersom input-verdiene enkelt kan endres. Det er potensial for å bruke denne koden til å optimalisere et brønn-nettverk hvor alle brønner har sin egen brønn-modell.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As an introduction to the master's thesis the background is presented, including the problem formulation and related work. Further, the objectives, approach and contributions will be described before an outline presents the structure of the thesis.

1.1 Background

The background for this thesis is curiosity regarding gas lift optimization for their field. They have a model of a well-network, made in Prosper, and need an efficient solver to optimize the field to maximize oil production, subject to a various process constraints.

Production wells for oil and gas are divided into two types, free flowing or lifted [4]. Free flowing wells are able to reach a required wellhead pressure (WHP) by itself with an acceptable well-flow. For lifted wells, the downhole pressure is too low and artificial lift is needed [4, p. 30]. There are different methods for artificial lift like rod pumps, electrical submerged pump (ESP), plunger lift and gas lift [4, p. 32], where the latter gives the foundation for this problem. When injecting lift-gas to a well the density and the hydrostatic pressure of the fluid column decreases, which makes the flowing bottomhole pressure (BHP) lower. This increases the differential in pressure across the reservoir and BHP, which helps the fluid reach the surface [21]. Gas lift can be divided into two types, intermittent or continuous. Intermittent gas lift is used when the oil flow into the wellbore is not continuous and operating valves are on/off at some interval. Continuous gas lift is used when the oil flow from the reservoir to the wellbore is continuous [16], and this is the case for the well-model used in this master's thesis.

In the project thesis [11], which was the pre-work of the master's thesis, the opportunities for gas lift optimization, using the software UniSim Design, was investigated. The lessons learned were that the optimization algorithms seemed efficient, but the solver for the process model in the software was too slow. This resulted in very slow running time (minutes) for simple cases with only two wells. With this experience it was reasonable to optimize using other process models. One task for the masters' thesis could be to import a Prosper-model to UniSim Design and use the optimizer it provides. Another task, which was the one chosen for this masters' thesis, was to import the model-data from Prosper to Python, and use existing optimization algorithms here. This was chosen because it was more professionally relevant for the masters' degree in Industrial Cybernetics and seemed more exciting.

1.1.1 Problem Formulation

The gas lift optimization problem can be described using figure 1.1. This shows how the oil production varies with different gas lift injection rate (GLIR) to one well. The technical optimum is the absolute maximum net oil production rate. The economic optimum is at a lower GLIR, because increasing gas

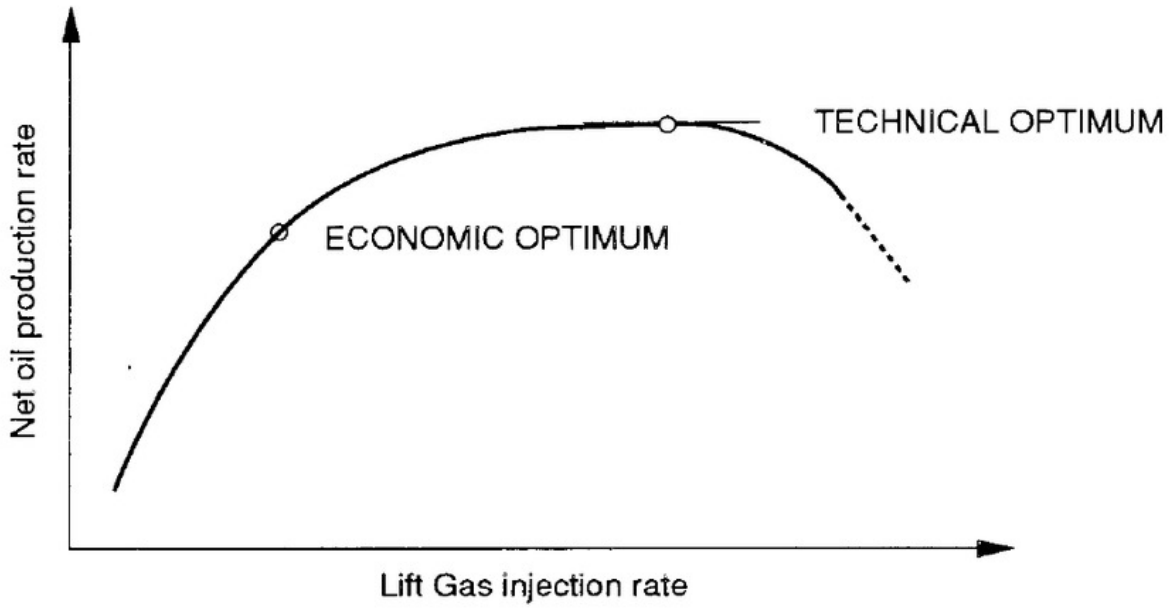

Figure 1.1: Gas lift performance curves showing oil production versus gas lift injection rate GLIR. Picture from Shell (1993)

lift from the economic optimum causes a small change in net oil production rate, compared to the price of increasing GLIR. The figure also shows that when the GLIR becomes large, the net oil production rate drops. The reasoning behind this is that when the GLIR increases, the pipe friction increases. When this becomes large enough, the pressure drop becomes too large so that increasing the GLIR gives lower net oil production rate. This is called the head loss of the system, which is the conversion of useful mechanical energy to waste thermal energy through viscous action [7, p. 214]. The head loss can be calculated by the energy equation.

The goal of this master's thesis is to maximize the oil production of a real well-network, consisting of n number of wells. This is done with the use of gas lift, subject to constraints like pressure, GLIR and other process constraints. An example of a subsea well-network consisting of four wells and two templates are presented in figure 1.2. This shows an oil rig, the wells connected in the templates at the sea bottom and a common pipeline. The well-model used in this work represents the part from the GLIR point to the wellhead located at the template. A template is a subsea structure that allows subsea wells to be operated and drilled remotely from a surface. It also protects the wells from damage, for instance made by trawlers [4, p. 34].

The well-model, representing the behaviour when the well is artificially lifted with gas, is made with Prosper and produce a .TPD-file. Prosper is a software for multi-phase well- and pipeline analysis and use nodal analysis [19]. The file, delivered by consists of a table that represents a number of calculated variables, including WHP and wellhead temperature (WHT). The calculated variables are the results of a combination of some free variables like GLIR, liquid rate (LR) and BHP. In the file used to represent one well in this work, the number of free variables are five. This can vary in different well-models.

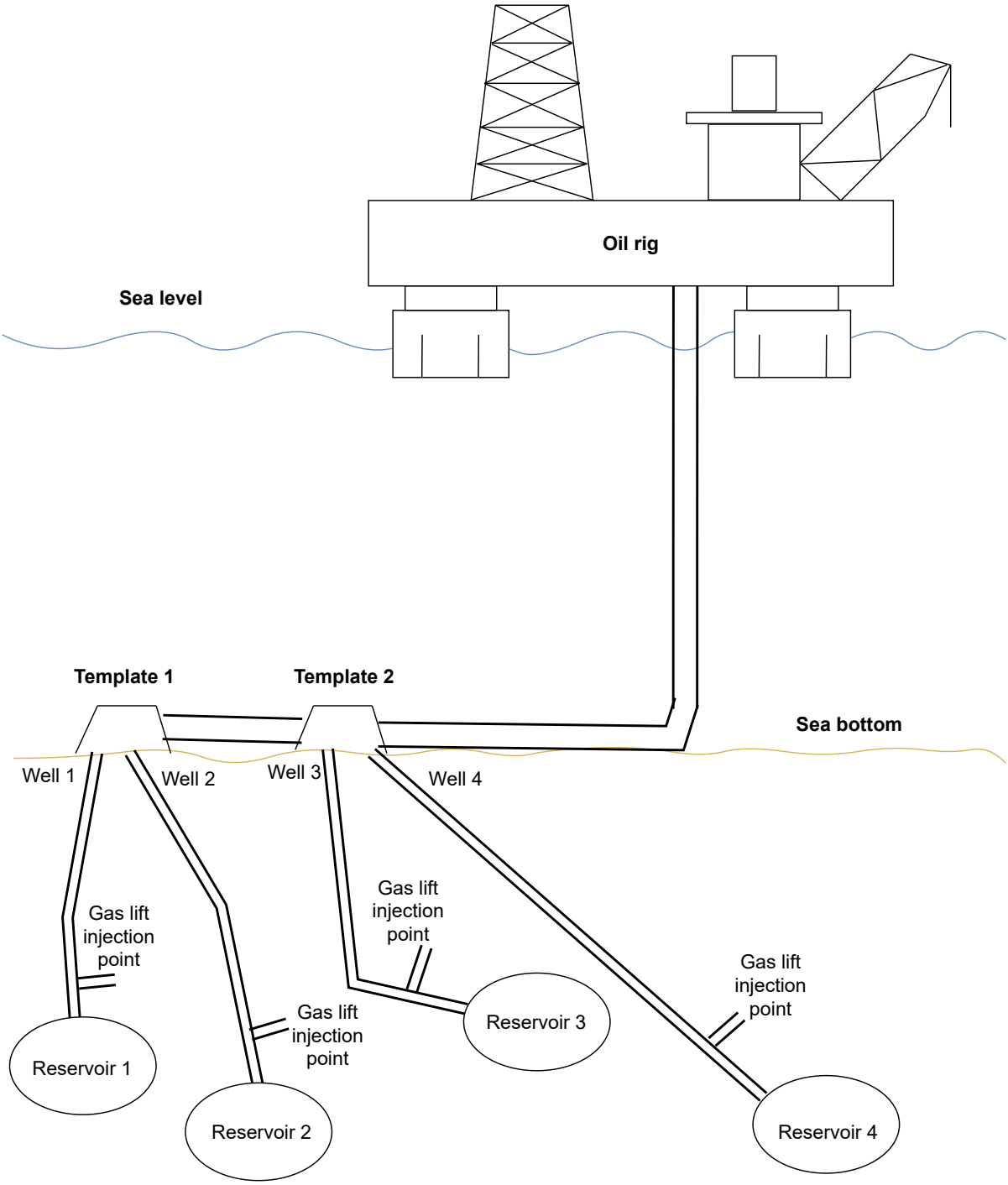

Figure 1.2: A sketch of an fictional well-network consisting of two templates and four wells

The first step in this work is to read the model-file into the preferred programming language, Python. It is important that this encoding is done in an efficient manner because the tables can be large (e.g. 168 000 rows of data-points). The model file is a .TPD file, but needs to be changed to .txt before imported to Python. The free variables in the file used in this work are BHP, water cut (WC), gas oil ratio (GOR), GLIR and LR with corresponding values. The code for importing the model-data should account for a varying number of free and calculated variables.

The imported table is made of discrete points, therefore the second step is to make a code that interpolates these points. It is necessary to make this in a way that makes it easy to re-use for other cases. The interpolation will be multi-dimensional because of the number of free variables. The next step is to make a solver that calculates the derivatives for the interpolated table. This can be time consuming, so one opportunity is to use an existing Python-solver for this problem. When this is done, the final step is to perform gas lift optimization on the case. This is done by using existing Python-algorithms like scipy.optimize. Different methods within scipy.optimize should be used and tested against each other to compare convergence and algorithm running time. The optimization should provide optimal oil- and gas lift injection rates for minimizing the defined objective function.

1.1.2 Related Work

As mentioned in section 1.1 the project thesis [11] was the first work regarding gas lift optimization for well-network. Lessons learned from this project made it necessary to use a different model of the wells to make the optimization more efficient. This resulted in the use of a Prosper well-model and Python-code for optimization.

There are a variety of open-source packages to use when optimizing in Python, like Pyomo, SciPy.optimize, CasADi, PuLP, where some are more relevant to this project than others. PuLP is a linear program modeller [20]. Our problem will not be linear, so this package can be disregarded. Both Pyomo [2, p. 3] and CasADi [1] provides a software framework for optimization. It is preferred that the optimizer is as flexible as possible and can be made specific for our problem, hence the optimizer provided by the community-driven open source project, Scipy [24], is tested first in this thesis.

The gas lift optimization problem is well known for wells with low or no production [21], and there has been done a lot of work on the area including single well analysis, network based solutions and network- and reservoir-based solutions [21].

Work on single well analysis have been done by Vazquez-Roman and Palafox-Hernandez (2005) regarding a well-model that looks to produce good results when the injection point depth is defined [33]. This model is based on mass, energy and momentum balance [21]. A study, provided by Dutta-Roy and Kattapuram in 1997, stated that single-well analysis has limited validity for optimizing a network consisting of multiple wells. This means that finding the optimum for each well seperately does not imply that this is the optimum for the network of wells [6]. The study also concluded that when wells have common flowlines the optimal rate of injection decreases due to lower efficiency of the gas lift injection. By calculating the optimal GLIR for a well-network by solving the field wide allocation problem simultaneously, considerable time- and analysis-savings can be made [6].

Network based solutions includes Dutta-Roy et al. (1997) work on computer-aided gas field planning and optimization, which concludes that varying gas price can change the optimal operating strategy significantly [5]. Another work by Nadar et al. (2008) stated that implementing software packages for full-field optimization reduced operating cost and gave production gain where the model was accurate for a complex production network [15]. Camponogara and Nakashima (2006) solved the gas-lift optimization problem using dynamic programming for small- to medium-sized instances (10-20 wells). The solution approach is approximating by nature, but can still give near-optimal solutions [3]. The dynamic programming formulation was used for well-rate and lift gas allocation problem [21].

Network- and reservoir-based solutions for gas lift optimization problems include Wang et al. (2002) work resulting in a new formulation for optimization problem of allocating well rates and lift-gas rates

[36]. Flow interactions between wells are considered in the new formulation and it can handle situations where some wells are not able to produce without gas lift while others can. The optimization method used were sequential quadratic programming (SQP), which is derivative-based [36].

When it comes to gas lift optimization using Prosper well-models, there have been some work. In 2020, Odjugo et al. investigated which method for artificial lift that gave the highest production rate to some given well-conditions, using the flow simulator Prosper to analyse a production well [18]. The work was done on one well and concluded that electrical submersible pump gave highest production rates, but with power costs. They also discussed that using continuous gas lift was applicable, while intermittent gas lift would not improve the production rates significantly. Another note they made was that the production optimization is very complex and every well cannot be optimized individually to achieve full field optimization and maximize income.

1.2 Objectives

The overall problem description was presented in 1.1.1 and the following list represents the summarized objectives for the master's thesis. These objectives should be solved and presented in the result-part of the thesis.

- Read the model-file into Python
  - For this project it is a .TPD file, but the reading should be modular so it can have different models as input
  - The importing has to account for that the number of free and calculated variables can vary
- Interpolate the table
- Make a code that calculate the derivatives of the interpolated table (or use existing Python-packages)
- Solve gas lift optimization using Python algorithms (scipy.optimize)
  - Solve two optimization problems for one well
  - Solve one optimization proble for multiple wells, returning optimized production- and gas lift injection rates for each well to minimize the objective function.
  - Test different methods against each other with respect to convergence and running time
- Make a simple graphical user interfaceGUI to manage the optimization-case:
  1. Specify wells with belonging Prosper-file
  2. Specify bounds and constraints
  3. Specify type of optimization-algorithm
  4. Start/stop optimization
  5. Plot results

1.3 Approach

The objectives presented in 1.2 are all solved different ways, but every result are obtained using Python. To import the data, the Pandas-package is used. The interpolation of data is based on the use of Scipy and its Regular Grid Interpolator. Scipy is also preferred for the optimization where Scipy.optimize.minimize is used.

It is important to verify that the results obtained are correct. This is done different for each objective, and these methods of verification are explained in the following. The objective of reading the model-file into Python is verified if it is possible to use the desired data from the model-file in Python. If this is possible, the task is sufficiently solved. The part-objective of having a modular reader is verified by testing it for different well-models, while the one for handling multiple variables are verified by testing it using different number of variables in the code.

Interpolation of the table can be verified by testing the interpolator for different values. The data that are interpolated are made by different combinations of the free variables. By varying one of the free variables as input until a pattern is obtained (which free variable vary first), one can use this to verify the results. This will be illustrated with an example. If we have two free variables x and y, and a resulting function f where the data-points are:

$$\begin{cases} x = [1, 2, 3] \\ y = [4, 5, 6] \\ f(x, y) = [5, 6, 7, 6, 7, 8, 7, 8, 9] \end{cases} \tag{1.1}$$

Here $f(x, y) = x + y$, and the pattern is that x varies before y (and y is constant when x i varying). To verify that an imaginary, linear interpolation of f(x, y) is correct, one could first check that the combination x = 1, y = 4 gave f(x, y) = 5 and the combination x = 2, y = 4 gave f(x, y) = 6. If this was correct and f(1.5, 4) = 5.5 it would substantiate that the interpolation is correct. The same as presented in this example is done for the interpolation of the well data to verify the results.

The verification of the optimization results obtained in this master's thesis is done in three different ways. For the two cases for one well there are two approaches used to verify the result; checking that the optimizer returns "*The optimizer terminated successfully*" and comparing the results with plots made. If the results from the optimizer corresponds with plots, it implies that the results obtained are correct. For the last and more complex optimization problem with multiple wells, the results are mainly verified by comparing the derivative to the theory (section 2.4) saying that the derivative should be equal for each well in the optimum.

1.4 Contributions

The main contributions of this masters thesis is as presented in the following list:

- A code for reading a well-model from Prosper (.txt-file) into Python that handle any number of free and calculated variables
- An interpolation-code for multidimensional interpolation, which can handle varying number of free variables (up to six) and as many calculated variables as needed. Number of free variables handled can be increased by performing small adjustments to the code
- A class consisting of multiple methods for defining and solving different optimization problems, including one for minimizing an objective function for n number of wells (as many wells n as needed)
- Different plots presenting well behaviour for different combinations of input-data

1.5 Outline

The organization of the master's thesis is as follows. Chapter 2 covers relevant theory, mainly of interpolation and optimization. The next chapter is called "Implementation and System Overview" and presents planning and execution of the work, the well-model used in the optimization and how it is

imported to Python and interpolated. This chapter also covers mathematical formulations of both interpolation and the optimization problems before the structure of the system produced is shown in some diagrams. Chapter 4 provides the results obtained in this work, with basis in the problem formulation and objectives presented. Chapter 5 gives a discussion of the results obtained including strengths, weaknesses, limitations and opportunities. At last, in chapter 6, the conclusions and further work are presented.

Chapter 2

Theory

In this chapter relevant theory will be presented to substantiate the master's thesis with important subjects like interpolation and optimization. This chapter focuses on these two main topics of theory, while other subjects are presented in different parts of the thesis where it is reasonable. The section regarding interpolation covers general interpolation theory and interpolation in Python with the use of Scipy. The optimization theory is presented through four parts including modelling, general optimization, gas lift optimization and optimization in Python.

2.1 Interpolation

Interpolation is about approximating the values between given data points, or making approximations to a dataset [26, p. 42]. An example of where it is necessary to make approximations can be that you measure the temperature outside every other hour for a day. By letting y_i denote the temperature and x_i denote the time one obtain the dataset (x_i, y_i) for $i = 2, 4, ..., 24$, shown in figure 2.1 . If one wants to find the temperature at a time where there is no corresponding data point, interpolation is necessary to connect the points.

There are several ways of performing interpolation of a dataset, including methods like nearest neighbor, linear or higher order interpolation using Lagrange polynomials [26].

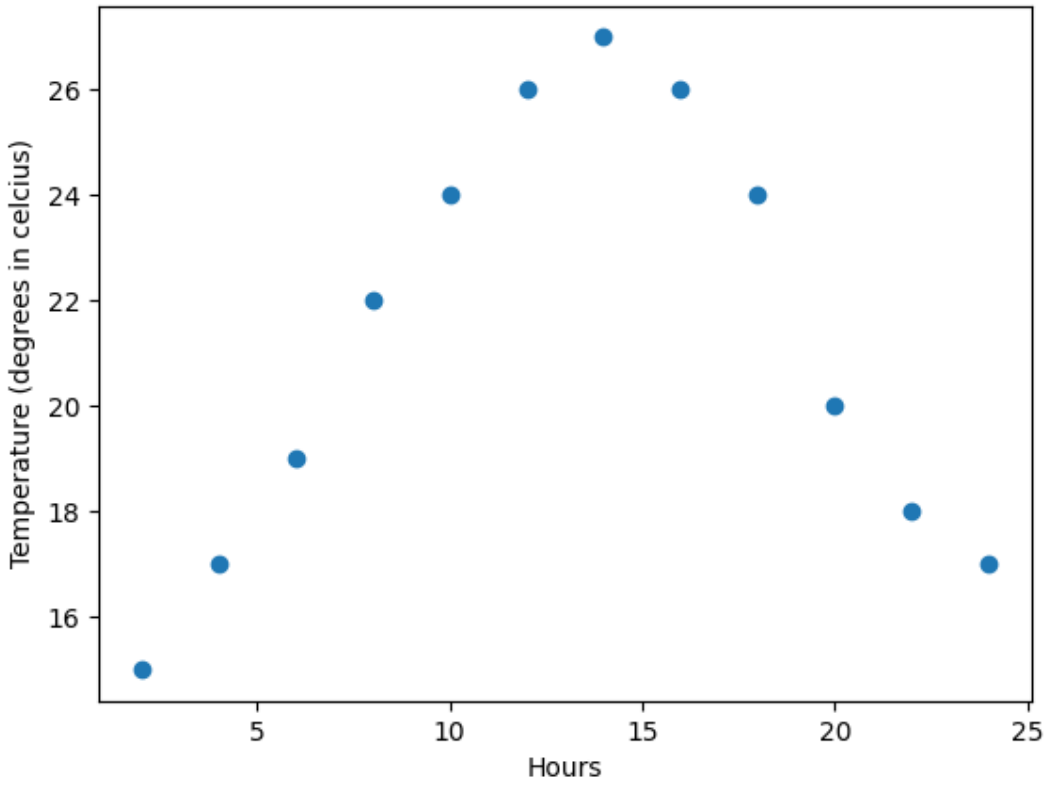

Figure 2.1: Dataset (x_i, y_i) describing measured temperature given an hour

2.1.1 Nearest Neighbour Interpolation

This method for interpolation use the value for the nearest data point when doing interpolation. At figure 2.1 there are temperatures corresponding to time every other hour. To extract the temperature after 9,5 hours and using nearest neighbor interpolation, one woulds say that the temperature is 24 degrees. This is the same as the nearest neighbour (the data point at x = 10 hours). This shows how the nearest neighbour-method works.

If the function $f(x)$ represents the exact outside temperature, the error for this type of interpolation can be shown to be [26, p. 43]:

$$\text{error} \;=\; C\Delta x\, f'(c) \tag{2.1}$$

Where C and c are constants. Equation 2.1 shows us that the error is dependent on both the derivative of the function and the distance between the samples. This means that if we measure something that changes fast and have few samples, then the nearest neighbour interpolation is bad. Decreased value of the derivative and more samples gives smaller error.

Another aspect to this nearest neighbour interpolation is that it does not represent the reality in a good manner. The discontinuous jumps it produce does not represent the reality, because it is not physical possible with infinite derivative (as discontinuous jumps represent) [26, p. 43].

2.1.2 Liner Interpolation

Linear interpolation connects the data points by constructing a straight line $f(x)$ from the data point (x_1, y_1) to the data point (x_2, y_2). A requirement on $f(x)$ is that it needs to satisfy:

$$f(x_1) = y_1 \text{ and } f(x_2) = y_2$$

and the line is given by:

$$f(x) = y_2 \frac{x - x_1}{x_2 - x_1} + y_1 \frac{x - x_2}{x_1 - x_2} \tag{2.2}$$

This line only represent the straight line from one point to another, so this has to be done for every interval $[x_i, x_{i+1}]$ in the dataset. For linear interpolation there is no discontinuous jumps like for nearest neighbour and the plot shown in figure 2.2 looks more realistic. But the sudden jumps in the derivatives on the edges corresponding to the data points are worth noting.

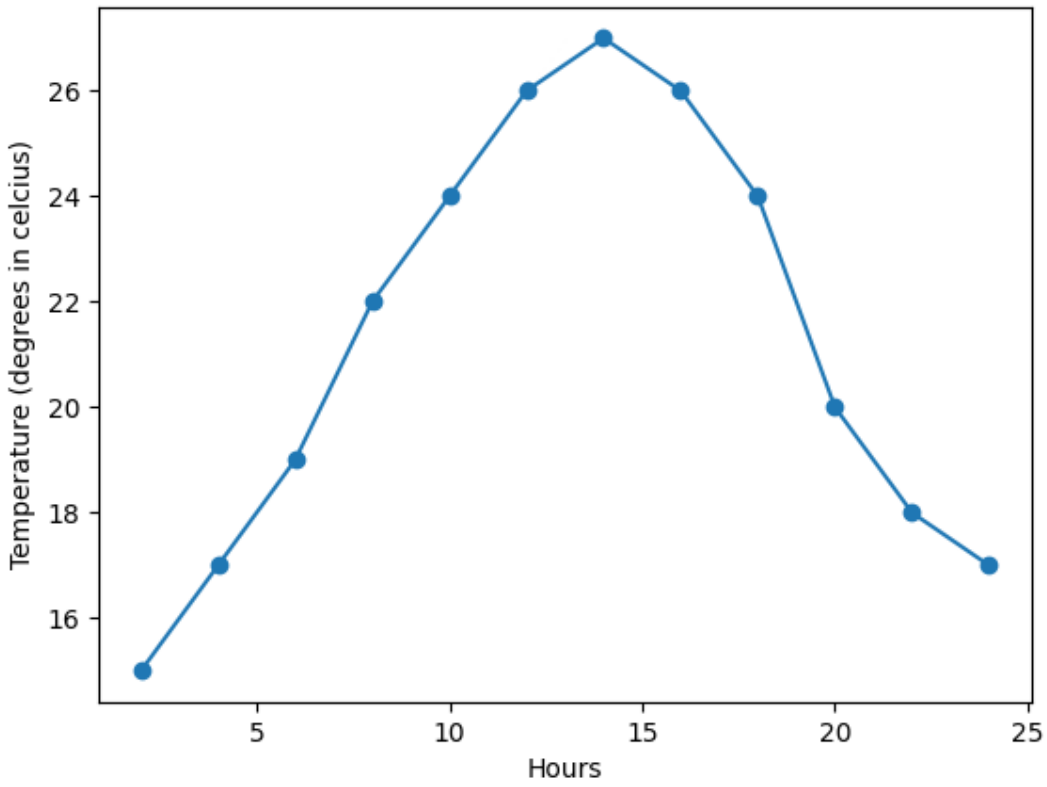

Figure 2.2: Dataset (x_i, y_i) describing measured temperature given an hour with linear interpolation.

The error for linear interpolation is given by the equation [26, p. 44]:

$$\text{error } = C\Delta x^2 \, f''(c) \tag{2.3}$$

This equation shows that the error is now dependent of the second derivative of function f(x), and the interval between the samples, squared. If the second derivative is zero, then the error is zero which makes sense because a second derivative of zero gives a linear function and the interpolation is identical to the actual function. Increased second derivative gives increased error, because it is more difficult to approximate a parabola that is changing a lot with a straight line. Another result from equation 2.3 is that by doubling Δx (distance between samples), the error is increased by four.

2.1.3 Higher Order Interpolation Using Lagrange Polynomials

Higher order polynomials can be used to join the data points with a smoother curve than explained in section 2.1.1 and 2.1.2. One way of doing this is by using Lagrange polynomials, where the polynomial is given by the following [26, p. 46]. For a set of $n+1$ nodes x_i for $i = 1, ..., n+1$, the $n+1$ Lagrange polynomials are:

$$\ell_i(x) = \prod_{j=1, j\neq i}^{n+1} \frac{x - x_j}{x_i - x_j}, \text{ for } i = 1, ..., n+1 \tag{2.4}$$

To construct an interpolating function of degree n that goes through the data point (x_i, y_i) for $i = 1, ..., n+1$ the following function is used:

$$f(x) = \sum_{i=1}^{n+1} y_i \ell_i(x) \tag{2.5}$$

This can be used for as many data points as wanted. One thing to consider when using this is that higher degree polynomials may produce something called Runge phenomenon where the polynomials shows some wiggles and the error increases [26, p. 47].

2.2 Regular Grid Interpolator

Regular grid interpolator is a class from the sub-package scipy.interpolate developed by the SciPy community. This community is a well-established and growing group of researchers, engineers and scientists that use, extends and promote SciPy as a open-source library for scientific research [14, p. 9]. The interpolator provides interpolation on regular grid in arbitrary dimensions and is accessed in Python with the following syntax [28]:

scipy.interpolate.RegularGridInterpolator(points, values, method, bounds_error, fill_value) (2.6)

The data have to be defined on a regular grid, while the grid spacing can be uneven. There are different requirements on the input, for instance that the points have to be a tuple of ndarray of float (these points defines the regular grid in n dimensions) and that the values are array-like. This data are on the regular grid in n dimensions. The regular grid interpolator supports nearest neighbor and linear as interpolation methods [28].

2.3 Optimization

There are different categories of optimization problems like unconstrained, constrained and optimization where the derivatives are not available (derivative free optimization (DFO)). Within constrained optimization problems there are different classes, like linear-, quadratic- and nonlinear programming [17,

p. 529]. Linear programming (LP) are concerned with maximizing or minimizing linear function over a polyhedron. LP was created and got its name after the work done by Dantzig, Kantorovich, Koopmans and von Neumann in the 1940's [23]. In a quadratic program the aim is to maximize or minimize a multivariate quadratic objective function, subject to linear constraints [37]. Note that a quadratic program is a special case of nonlinear programming, because the objective function is nonlinear.

Nonlinear programming is in general non-convex. The practical meaning of this is that an optimum is not necessarily a global optimum [17, p. 7]. A general nonlinear program is given by:

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & g(x) = 0 \\ & h(x) \leq 0 \end{aligned} \tag{2.7}$$

$$\min_{x \in \mathbb{R}^n} \quad f(x) \quad \text{subject to} \quad \begin{aligned} c_i(x) &= 0, \quad i \in \mathcal{E} \\ c_i(x) &\geq 0, \quad i \in \mathcal{I} \end{aligned} \tag{2.8}$$

The optimum x^* of the optimization problem in equation 2.8 is in the feasible region of the optimization problem. This is the set of points satisfying all the constraints or the area between two constraints c_1 and c_2 boundaries [17, p. 3].

Some optimization algorithms from the different classifications of optimization problems are described in the following to give a description on how optimization problems can be solved. The methods for solving optimization problems presented include Nelder-Mead simplex reflection, Newton-CG, Trust region methods, BFGS and Sequential Quadratic Programming (SQP).

2.3.1 Nelder-Mead Simplex Reflection

The Nelder-Mead simplex-reflection method is part of the class of optimization problems called derivative free optimization (DFO). This means that the method does not use the derivatives to determine the new iteration [17, p. 221]. The method keeps track of the n + 1 points of interests in $\mathbb{R}^3$ where the convex hull forms a simplex. A convex hull is defined as the smallest convex region enclosing a planar set of points P [8, p. 324]. The Nelder-Mead method is built up by iterations where the goal is to remove the vertex with the worst function value and replace it with a point of better value. The method finds a better valued point by expanding, reflecting or contracting the simplex along the line from the worst vertex with the centroid of the remaining vertices. If this does not work the algorithm retain the vertex with the best value and shrink the simplex [17, p. 238].

2.3.2 Newton-CG

Newton-CG stands for Newton-conjugate gradient which is a procedure developed in the 1980s [22]. The method is used for unconstrained optimization problems and applies to non-convex, smooth functions. It uses the conjugate gradient optimization method to the second-order Taylor-series approximation of f around each iterate x_k. The conjugate gradient method is a method for solving a system of linear equations $Ax = k$ [10].

2.3.3 Trust Region Methods

Trust region methods are used to solve optimization problems numerically and can be used for unconstrained- and constrained optimization, nonlinear equations, nonlinear least squares, nonsmooth optimization and DFO. These methods compute a trial step by calculating a trust region sub problem.

Within this trust region, a model function is minimized. This is unlike search-methods that does a line-search at each iteration [38]. These methods can be applied to non convex and ill-conditioned problems, and are powerful for ensuring global convergence [38, p. 20].

2.3.4 BFGS

BFGS is a quasi-Newton algorithm, and the most popular. Quasi-Newton methods only require the gradient at each iteration (and not hessian) to converge to local minima. Because of this it can be more efficient than Newton's method [17, p. 136]. BFGS next iteration is given by:

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.9}$$

where the search direction p_k is given by:

$$p_k = -B_k^{-1} \nabla f_k \tag{2.10}$$

where B_k is an $n \times n$ symmetric positive definite matrix that is updated at each iteration and f_k is the objective function. The α_k in 2.9 is chosen to satisfy the Wolfe condition [17, p. 34], which states the conditions for what step-length that is acceptable. The iteration shown in 2.9 differs from Newton method because in BFGS an approximation of the Hessian B_k is used instead of the true Hessian.

2.3.5 Sequential Quadratic Programming

The sequental quadratic programming (SQP)-algorithm is part of the classes of constrained optimization and belongs to nonlinear programming [17, p. 529]. SQP-algorithms are the workhorse for solving constrained, nonlinear optimization [21]. As the name states, the SQP-algorithm solves quadratic sub-problems and generates steps. This algorithm works well for small and large problems, and is powerful when the constraints are significant nonlinear.

To explain SQP, a practical line search SQP-method [17, p. 545] is explained briefly. It is important to notice that there can be a variety of ways to calculate the hessian approximation, the step-acceptance mechanism and other features of the algorithm [17, p. 545]. The first part of the SQP-algorithm is there to compute a search direction. The second part of algorithm performs a line search where the algorithm have to take both the objective and the constraint into consideration. The concept of merit function is used, which controls the step-size and gives an indication if it is acceptable or not. Finally the SQP-algorithm updates the approximation of the hessian.

A useful technique in SQP is BFGS (named after its inventors, Broyden, Fletcher, Goldfarb and Shanno), which is one of the most popular formulas for updating the Hessian approximation. The BFGS update gives an approximation that is positive definite when the initial approximation is positive definite [17, p. 24].

2.4 Gas Lift Optimization

In gas lift optimization there are some theory that can confirm if the result of an optimization case is correct. In an optimization case where one wants to maximize oil production by changing the GLIR (x_n) for n number of wells the theory shows that the optimum x_n^* has the same derivative for each well. The theory behind this will be derived with basis in the First-Order Necessary Conditions for constrained optimization, also known as the Karush-Kuhn-Tucker (KKT)-conditions [17, p. 321]. These conditions describes behaviour for the first derivative vectors for constraints and objective functions in optimization cases.

The Lagrangian function is necessary to define before stating the KKT-conditions [17, p. 320]:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x). \tag{2.11}$$

The KKT-conditions is then stated as [17, p. 321]:

$$\nabla \mathcal{L}(x^*, \lambda^*) = 0 \tag{2.12a}$$
$$c_i(x^*) = 0, \text{ for all } i \in \mathcal{E}, \tag{2.12b}$$
$$c_i(x^*) \geq 0, \text{ for all } i \in \mathcal{I}, \tag{2.12c}$$
$$\lambda_i^* \geq 0, \text{ for all } i \in \mathcal{I}, \tag{2.12d}$$
$$\lambda_i^* c_i(x^*) = 0, \text{ for all } i \in \mathcal{E} \cup \mathcal{I}. \tag{2.12e}$$

The KKT-conditions will be explained and tested on an idealized example of gas lift optimization. This optimization problem have n number of wells:

$$\min_{x_1, x_2, \ldots, x_i} \quad \sum_{i=1}^{i=n} f_i(x_i) \quad \text{s.t.} \quad \sum_{i=1}^{i=n} x_i \leq M \tag{2.13}$$

Where f_i represents the oil production of well i. The variable x_i represents the gas lift injection for well i and M is the limit for total gas lift injection. By using the definition of the Lagrangian function in 2.11 we obtain the following:

$$\mathcal{L}(x, \lambda) = \sum_{i=1}^{i=n} f_i(x_i) - \lambda(M - \sum_{i=1}^{i=n} x_i) \tag{2.14}$$

By applying the KKT-conditions on the optimization problem presented in 2.13 and 2.14 the following is obtained:

$$\nabla_{x_i} \mathcal{L} = \nabla f_i(x_i^*) - \lambda^* = 0 \implies \lambda^* = \nabla f_i(x_i^*) \tag{2.15a}$$
$$M - \sum_{i=1}^{i=n} x_i^* \geq 0 \tag{2.15b}$$
$$\lambda^* \geq 0 \tag{2.15c}$$
$$\lambda^*(M - \sum_{i=1}^{i=n} x_i^*) = 0 \tag{2.15d}$$

From equation 2.15a it can be seen that the gradient $\frac{\partial f_i}{\partial x_i}$ have to be equal to λ^*. In an optimization case with two wells this gives:

$$\nabla f_1(x_1^*) = \nabla f_2(x_2^*) = \lambda^* \tag{2.16}$$

This is an important finding and can be used to verify that the result of a gas lift optimization case is correct.

2.5 Optimization in Python - Scipy.optimize

With the use of Scipy.optimize it is possible to use its built-in functions for minimizing objective functions subject to various constraints [27]. The minimize function supports a variety of methods for local, multivariate optimization, including Nelder-Mead, BFGS, trust-constr and SLSQP.

The minimize-function have several inputs, and the required parameters are:

- fun: This is the objective function to be minimized. It is a function with inputs x (1D array with shape (n,)) and args (a tuple of fixed parameters that completely specifies the function). This function have to return a float.
- x0: This is an ndarray of real elements with size (n,) which represents the initial guess to the optimizer. n is the number of independent variables.

There are also different optional inputs, like extra arguments to the objective function, specifying type of solver, method for computing the gradient vector and hessian matrix, bounds and constraints among others. The result of running the minimizer is an *OptimizeResult*-object and contains the solution array, a Bolean flag indicating success/failure and a message describing the reason for termination [32]. The constraints in Scipy.optimize should be defined in Python so that they are equal to or larger than zero, like this:

$$\begin{aligned} c(x) &= 0 \\ \text{or } c(x) &\geq 0 \end{aligned} \tag{2.17}$$

When it comes to method for computing the gradient vector there are five options; a callable function, None/False, "2-point", "3-point" or "cs". By defining a callable function the user can specify the gradient vector. By using None/False as input, the estimation of the gradient is done with a 2-point finite difference estimation, with an absolute step-size [32]. This is different from specifying "2-point", "3-point" or "cs", because this can be done to choose a relative step-size for the finite difference scheme for numerical estimation [32].

The different methods for optimization that can be chosen for scipy.optimize.minimize are 'Nelder-Mead', 'Powell', 'CG', 'Newton-CG', 'L-BFGS-B', 'TNC', 'COBYLA', 'SLSQP', 'trust-constr', 'dogleg', 'trust-ncg', 'trust-ncg', 'trust-exact', 'trust-krylov' or custom. If the custom method is chosen it is inserted as a callable object that defines the method [32]. The different methods should be chosen based on the optimization problem to be solved.

For global optimization, scipy.optimize.minimize includes the methods basinhopping, brute and differential_evolution, among others [27]. Basinhopping works well for problems that are similar to the natural process of energy minimizations of clusters of atoms, because this is what the algorithm is designed for [29]. The use of the brute force method is time consuming and inefficient because the number of grid points increase exponentially, so even moderate size problems can take a long time or terminate because of limitations in memory [30]. The method differential evolution finds the global minimum of a multivariate function. This does not use gradient evaluations to find minimum, and is stochastic in nature. The method normally requires many function evaluations compared to methods using gradients in the algorithm [31].