Master Thesis

Modelling and Production Optimisation of Oil Reservoirs

Dariusz Lerch

Danmarks Tekniske Universitet

Department of Informatics and Mathematical Modelling
Group of Scientific Computing
Centre for Energy Resources Engineering

DTU Informatik
Institut for Informatik og Matematisk Modellering

CERE
Center for Energy Resources Engineering

Kongens Lyngby, 2012
IMM-M.Sc-2012-124

Technical University of Denmark
Department of Informatics and Mathematical Modelling (IMM)
Group of Scientific Computing
Centre for Energy Resources Engineering (CERE)
Building 321, DK-2800, Kongens Lyngby, Denmark
reception@imm.dtu.dk
www.imm.dtu.dk
www.cere.dtu.dk
IMM-M.Sc.-2012-124

Summary

Constrained Numerical Optimisation is a very wide and broad field of interest which nowadays can be utilised for solving many engineering problems, where some adjustment of the input values is necessary for obtaining more efficient results from the system. A very challenging part of it is constraining the problem by relating it to mathematical model which represents the real life system. In most cases such a model is a set of differential - algebraic equations (DAEs), taking states of the system as function arguments, for which we look for the most optimal input variables (controls). It is very important that such a model both preserves the physical properties of the system which occur in the real world, e.g. conservation of mass, momentum, energy and is not too complex at the same time so it can be computed in a straightforward manner. Handling mentioned requirements in case of production optimisation of oil reservoirs imposed common usage of shooting methods in both academic and industrial communities for these problems since they eliminate the presence of state variables in the optimisation algorithm and consequently reduce the size of the problem which is very big mainly due to spacial and time discretisation. In this thesis we focus on applying, still unexplored by oil communities and competent to shooting methods, direct collocation approach in order to optimise oil production under water-flooding in a natural subsurface oil reservoir in secondary recovery phase.

We used an Interior Point Optimiser (Ipopt), which is a software package targeted for large scale non-linear optimisation that was set up in Visual Studio integrated development environment (IDE) and C++ object oriented programming language (OOPL). Before tackling oil problem, we tested simultaneous method in Ipopt on a well known non-linear problem of Van Der Pol oscillator. Then we implemented a two-phase (water-oil) immiscible flow model for an isothermal reservoir with isotropic permeability properties based on mass conservation principle for the each phase. We discretised the model in space by using finite volume method (FVM) and used the two point flux approximation (TPFA) and the single point upstream (SPU) scheme for flux computation. We discretised a new model formulation in time by using implicit scheme backward Euler method. Next we presented that obtained reformulation preserves the mass conservation properties used for model derivation. We implemented a numerical approach with the aim of the future usage by non-linear model predictive control (NMPC) framework and smart-well technology in order to maximise the Net Present Value (NPV) of oil production, which is a function of controllable inputs such as injection rates at the injection wells and bottom whole pressures (BHPs) at the production wells. The optimisation is based on interior point algorithm in the line search framework with the BFGS quasi-Newton method that computes an approximation of the inverse of the Hessian given the first order gradient. The Jacobian of the constraints in the sparse format is obtained analytically by utilising the open structure supported by simultaneous method and direct access to the first order derivatives.

The results from the simultaneous method investigated in this work, present that it has the clear and merit potential for upstream production optimisation of oil reservoirs.

Dansk Resumé

Mange fagområder anvender optimering med bibetingelser til at løse tekniske problemer der involverer justering af input-parametre (kontrolvariable) for at opnå optimal output fra systemet. Opstillingen af en matematisk model, der kan repræsentere den virkelige opførsel af systemet, er udfordrende. I de fleste tilfælde består modellen af et sæt af algebraiske ligninger samt differentialligninger (DAE), der relaterer tilstanden af systemet til en række kontrolvariable. Det er meget vigtigt, at modellen bevarer den fysiske opførsel af systemet, som forekommer i den virkelige verden, fx bevarelse af masse, momentum, energi samtidig med at modellen ikke må være for kompliceret og tidskrævende. Kompleksiteten kan reduceres af shooting metoder fordi de eliminerer tilstandsvariable og reducerer antal af variable i et diskretiseret optimering problem. En alternativ måde kunne være gennem direkte kollokation hvor alle tilstandsvariable beholdes og hvor man gør brug af åbne strukturer i det diskretiserede problem.
I denne afhandling fokuserer vi på at anvende en direkte kollokationstilgang for at optimere olieproduktionen under vand-oversvømmelse i et naturligt underjordisk olie felt i sekundær recovery fase.

Vi brugte et indre punkt Optimiser (Ipopt), som er en softwarepakke målrettet til store non-lineær optimering, der blev oprettet i Visual Studio integreret udviklingsmiljø (IDE) og C + + objektorienteret programmeringssprog (OOPL). Før tackle olie problem, testede vi samtidig metode i Ipopt på et velkendt lineær problem med Van Der Pol oscillator. Så implementerede vi en to-fase (vand-olie) blandbar flow model for en isoterm reservoir med isotrope permeabilitetsegenskaber baseret på massebevarelse princip for hver fase. Vi har diskretiseret modellen i rummet ved hjælp af finite volume metoden (FVM) og brugt de to point flux tilnærmelse (TPFA) og den enkelt punkt opstrøms (SPU) skema for flux beregning. Vi har diskretiseret en ny model formulering i tid ved at bruge implicit baglæns Euler-metoden. Næste præsenterede vi , der opnås omformulering bevarer de massebevarelse ejendomme, der anvendes til model afledning. Vi implementerede en numerisk fremgangsmåde med henblik på den fremtidige anvendelse af ikke-lineær model prædiktiv kontrol (NMPC) ramme og smart-brøndteknologi med henblik på at maksimere Net Present Value (NPV) af olieproduktionen, som er en funktion af styrbare input såsom injektionsatser på injektionsbrøndene og bundhulstryk (BHPs) på produktionsbrøndene. Optimeringen er baseret på indre punkt algoritme med linjesøgning og BFGS (Broyden Fletcher Goldfarb Shanno) quasi-Newton metode, der beregner en tilnærmelse af den inverse Hessian givet den første ordre gradient. En analytisk jacobian af bibetingelserne i sparse format opnås ved at udnytte den åbne struktur og understøttes af den simultane løsningsmetode og direkte adgang til de første ordre derivater. Dette arbejdes resultater med brug af den simultane løsningsmetode, viser at den har klart potentiale for optimering af opstrøms produktion af oliereservoirer.

Preface

This thesis was prepared at the Department of Informatics and Mathematical Modelling at Technical University of Denmark (DTU Informatics) and cross-department unit Centre for Energy Resources Engineering (CERE) in the partial fulfilment of receiving masters degree in Computer Science and Engineering. The work done in this thesis was supervised by Associate Professor John Bagterp Jørgensen and supported by other contributions mentioned in the acknowledgement section. The project was carried out from April 2012 to September 2012. The work done in this project has been submitted and successfully accepted in many conferences and scientific workshops focusing on such topics as optimisation, control and automation, energy resources engineering, and petroleum engineering, see appendix D. The results have been presented in form of either a poster or an oral presentation which resulted in not only feedback from specialists in fields mentioned but also contributed to the improvement of the content of this thesis.

Danmark, København, Kongens Lyngby,
September 2012,
Dariusz Michal Lerch

Papers Included in the Thesis

[A] Dariusz Michal Lerch, Andrea Capolei, Carsten Völcker, John Bagterp Jørgensen, Erling Halfdan Stenby, *Production Optimisation of the Two Phase Flow Reservoir in the Secondary Recovery Phase Using Simultaneous Method and Interior Point Optimiser,* Department of Informatics and Mathematical Modelling (DTU IMM), Group of Scientific Computing, Centre for Energy Resources Engineering (CERE), 2012.

[B] Dariusz Michal Lerch, *Ipopt in MSVS 2008 A Tutorial for Setting up Ipopt Solver in Visual Studio 2008 IDE,* DTU Informatics, Group of Scientific Computing, 2011.

Acknowledgements

I would like to thank my Danish supervisor and mentor John for first of all, introducing me to the topic of constrained numerical optimisation and reservoir engineering as I have to say that before starting working on this thesis these two fields were quite unexplored to me. Second of all, I would like to thank him for being helpful and prudent by scheduling meetings with me on a very regular basis throughout the whole project.

What is more, I should say thank you to Bjørn Maribo-Mogensen, who is a PhD student and software developer at Centre for Energy Resources Engineering (CERE) at Technical University of Denmark (DTU). Since we are work colleagues and share office together Bjørn was always around willing to answer some of my questions concerning implementation of optimisation problems in C++ as he has already a 9-year expertise in programing although he is a 26 year old newly started PhD student.

Next, I would like to thank Carsten Völcker, who has been working on the subsurface two-phase flow simulation and production optimisation for the past 3 years as a PhD student at Scientific Computing Group at DTU Informatics, for sharing with me his papers and posters which I used as an inspiration for presenting my own work.

Another PhD student that I am grateful to is Andrea Capolei. I had the pleasure to work with Andrea in the last weeks of my project and have access to his Matlab implementation of the oil problem based on which I developed my own C++ reservoir simulator.

Furthermore, I am very grateful to PhD student at Centre for Energy Resources Engineering at Chemical Engineering Department, Hao Yuan who works on stochastic modelling polydisperse transport of particles in porous media and was willing to answer all my questions as well as suggest some interesting literature concerning such aspects as two point flux approximation, concept of linear velocity in the porous media or computation of well indexes.

I would also like to thank professor Alexander Shapiro which I have been working for in the past 3 years as a software developer of software for thermodynamic equilibria SPECS in Centre for Energy Resources Engineering. Beyond working with thermodynamic models Professor Shapiro specialises also in advanced waterflooding and modelling of many phenomena taking place in porous media and helped me a lot in understanding the water-oil transport model and verifying its implementation in C++.

Another professor from Chemical Engineering Department whom I would like to say thank you to, although he was not officially involved in my project, is Wei Yan whom I consulted many times about scaling the reservoir parameters used in the two phase flow model.

Special thanks goes to the Centre for Energy Resources Engineering (CERE) and Society of Petroleum Engineering Copenhagen Section (SPE Copenhagen). Both CERE and SPE Copenhagen supported me in not only financial but also other ways, covering my travel expenses to the conferences regarding petroleum engineering topics as well as providing me with very satisfactory work facilities throughout my project.

Finally, I would like to thank my granny, mum and sister for getting me motivated and nagging me to keep on working on my thesis.

Contents

List of Figures

CHAPTER 1

Introduction

Natural petroleum reservoirs known also as oil fields are characterised by the multiphase flow of water and hydrocarbons in the porous media (e.g. rocks). The presence of each phase is dependent on the structure and pressure in the reservoir. A couple of decades ago oil fields were very easy to find and exploit. Initially most of them were discovered onshore however nowadays due to the need for fossil fuel energy offshore areas are becoming more popular then onshore ones taking oil production even to the Arctic region. The development of a conventional oil field starts with placing and drilling the wells into the reservoir rock. Before this is done very often one needs to perform the optimisation in order to find the most efficient well setting with respect to the permeability field, oil saturation and other parameters obtained from the geological measurements. This involves the usage of commercial simulation packages such as Eclipse [1] which analyse different production scenarios based on the well placement which then gives a support to an experienced reservoir engineer that is responsible for choosing the best scenario. Consequently, optimisation techniques are used in several different stages of the oil field development even before the actual production has started and sometimes might involve human decisions. Once the wells are drilled they get connected to the surface facilities from which oil is transported to the refineries for further processing; see fig. 1.1. The production stage of field development of a conventional oil normally can consist of three different phases which are categorised based on the methods applied to extract hydrocarbons from the subsurface. In the primary phase conventional methods which utilise high initial pressure obtained from natural drive are used. Those techniques however, leave more than 70 percent of oil in the reservoir.

A promising decrease of these remained resources can be provided in the secondary recovery phase, where water is injected at the injection well to sustain pressure level and push the oil towards the production well; see fig. 1.2. In some cases a third, tertiary phase can be approved by economic indicators based on the estimated reservoir potential. The tertiary phase, known also as enhanced oil recovery includes technologies such as in situ combustion, CO_2 , polymer or surfactant flooding and is targeted at recovering oil using chemical and thermal effects that reduce the adhesion and surface tension between oil and rocks as well as its viscosity[2]. Since injection of those substances is much more expensive than waters this phase requires complex estimations of the economic value of reservoir resources and occurs more seldom than water-flooding [3]. In this work we are focused on the oil production in the secondary recovery phase with water-flooding technique supported by the smart wells. As mentioned above the basic idea of this method is to sustain already dropped pressure level due to produced oil in the primary phase. If the pressure level in the reservoir is higher than bubble point pressure, secondary recovery occurs by two-phase immiscible flow, where one phase is water and the other is oil. The word immiscible means that two phases do not form a uniform solution but remain separated and what is more, they do not exchange mass with each other. In case of pressure being lower than the bubble point one, oil phase is spilt into vapour and liquid being in equilibrium state. Again there is no mass transfer between water phase and other phases but the hydrocarbon liquid and vapour phases exchange mass in such a manner that they stay in a thermodynamic equilibrium. In this work we focus on the case when bubble pressure is higher than bubble point pressure and consider two-phase immiscible flow of oil and water in the reservoir.

There are many factors contributing to the poor conventional secondary recovery methods e.g. strong surface tension, trapping oil in small pores, heterogeneity of the porous rock structure leading to change of permeability with position in the reservoir or high oil viscosity. Thanks to optimal control, it is possible to adjust injection valves in the individual segments of the well (see fig. 1.3) to control the two-phase immiscible flow in the reservoir and navigate effectively oil to the production wells, so it is swept from the reservoir and does not remain in the porous media.

1.1 Motivation and Main Objectives

The current demand for energy and decreasing number of newly found oil fields gives a clear motivation for exploiting already existing reservoirs to a most possible and satisfactory level. Oil has played a crucial role in the development of human civilisation, and its price, like prices of other commodities, experienced wide swings throughout the last decades in times of shortage or oversupply. What is more, there have also been many political, economic and cultural factors which rapidly affected the oil price and led to its sudden growth or drop throughout the last 40 years; see fig. 1.4. Nevertheless, from the global perspective one can observe a distinct gradual increase of the oil

Figure 1.1: A cross-section view of an offshore field equipped in a smart well system driving subsea production [4]

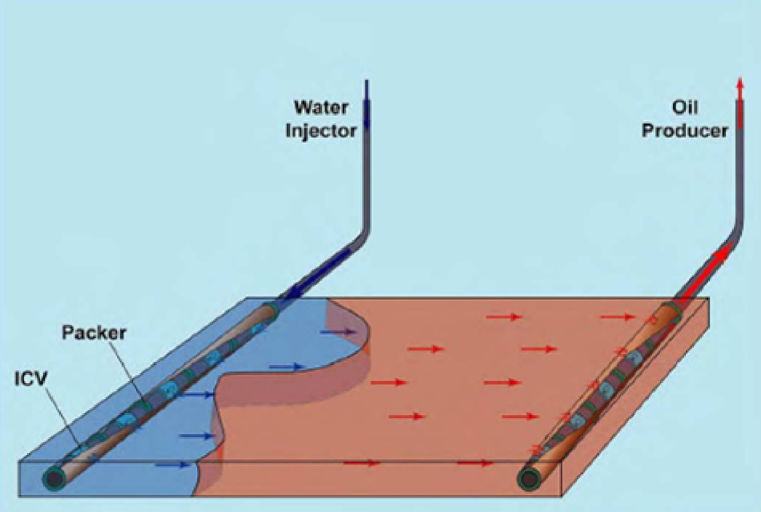


Figure 1.2: Schematic view of horizontal smart well [5]

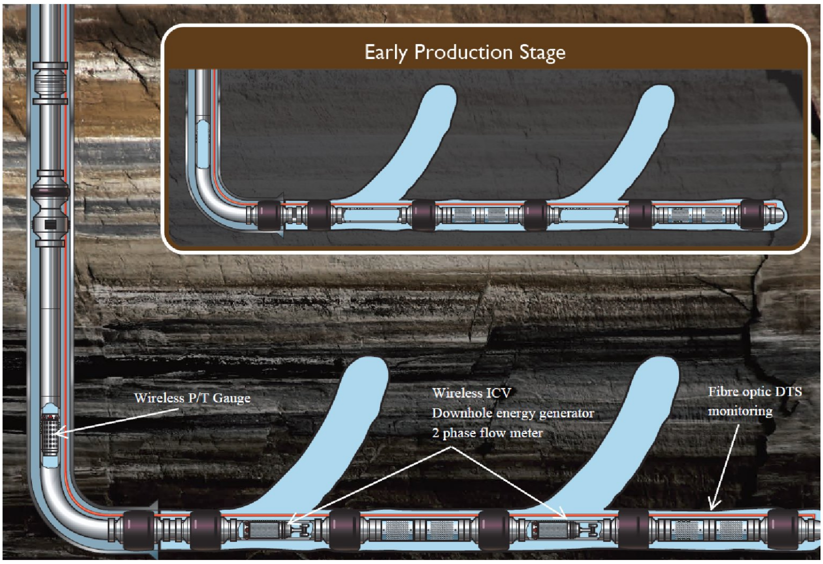


Figure 1.3: A smart-well with individually controllable segments equipped in wireless injection control valves (ICVs) [6]

prices throughout the last decades; see fig. 1.5 , and primary reasons for that are simple supply and demand, driven by the rapidly expanding countries of the developing world, principally China and India. Even financial speculators claim that the world is approaching a fundamental shift in energy prices that will reach a radically new level, expecting oil price reaching even 250 dollars for barrel in the foreseeable future. All those factors resulted in a necessity for a very efficient oil production and led to an interest for reservoir simulation studies as well as exploration of optimisation methods, that are used in the reservoir engineering combined with data assimilation algorithms for improvement of the secondary recovery phase under water-flooding technique.

The optimisation of oil production, which is the main objective of this work, is stated as an optimal control problem in following Bolza form:

$$\min_{\{u(t),x(t)\}_{t_0}^{t_f}} \int_{t_0}^{t_f} J(x(t),u(t))dt \tag{1.1.1a}$$

$$s.t. \quad x(t_0) = x_0 \tag{1.1.1b}$$

$$\frac{dg}{dt}(x(t)) = f(x(t),u(t) \quad t \in [t_0,t_f] \tag{1.1.1c}$$

$$u_{min}^{\Delta} \leq \frac{du}{dt}(t) \leq u_{max}^{\Delta} \tag{1.1.1d}$$

$$u_{\min} \leq u(t) \leq u_{\max} \tag{1.1.1e}$$

Where the eq. (1.1.1a) represents the objective cost function aimed to be minimised with respect to net present value or any other economic objective; eq. (1.1.1b) is an intial condition imposed on the state variables, and eq. (1.1.1c) is a mathematical model designating the nonlinear path constraints, which for oil problem becomes the system of differential equations describing the flow through the porous media, the right hand side of the eq. (1.1.1c) , g(x) are the properties conserved, x(t) are the system states, u(t) are the manipulated variables, while the right-hand side f(x(t), u(t)) has the usual interpretation. Equation (1.1.1d) represents the inequality movement constraints on the control variables, and equation (1.1.1e) states the upper and lower bounds on the control variables. t is an independent variable representing time and t_0 and t_f are initial and terminal times respectively. There exist different methods for solving optimal control problem and they are categorized based on the way the discretise the continuous time problem. Basically, one can distinguish between single-shooting, simultaneous method and a hybrid approach of those two offering a trade-off between of the traits of the two extremes, which is called multiple shooting [7]. So far, the most of attention from academic and industrial communities was given to the single-shooting method, which has been tried out in many works, e.g. in [8], [9], for production optimisation of oil reservoirs. One of the main reasons for common usage of single-shooting (or sequential method as optimisation is executed sequentially to numerical simulation for gradient computation) is because after reformulation it uses only manipulated variables (controls) as optimisation variables [7], [10], which reduces the variable space in the algorithm. Size reduction is a very attractive feature especially for reservoir simulation problems since they have the tendency to be very big in the first place (up to millions of variables) due to reformulating the model by discretising it in time ans space. It is very convenient to eliminate the states from the optimisation algorithm and solve the smaller reformulation using sequential quadratic programming (SQP). What is more, single-shooting is used with the high order ESDIRK methods equiped in step size control and error estimation which results not only in lower number of discretisation points, but also ensures that the model equations are integrated properly. There exist however, some drawbacks coming along with using single shooting. First of all, as a sequential approach single shooting needs model solution at each iteration as it progresses towards the solution of optimisation problem between solving the model and reduced gradient

problem. Hence, from computational point of view single shooting might be costly if function evaluation is costly [10], e.g. if the implicit discretisation scheme has to be applied, which is the case of production optimisation of oil reservoirs.
In this thesis we will test the simultaneous method (fig. 1.6), which has not been explored much by academic and industrial communities for optimising upstream oil production [11]. Simultaneous method, contrary to single-shooting, uses also the discretised future process model variables as optimisation variables. As a result, the newly transcribed discrete nonlinear program is much larger than by single-shooting. Nevertheless, it is often the case that after direct transcription the problem is very sparse and structured, so it is possible to define the sparsity pattern in an algorithmic manner. Of course implementation of the sparsity pattern can be sometimes very time consuming, but it offers a great trade-off when it comes to the reduction of the problem size and other computational aspects. Simultaneous methods do not solve the model at each iteration. Alternatively, as the name says for itself, a simultaneous search for a model solution and optimal point is carried out. Hence, from computational point of view, they can be less costly than single shooting methods. Other features supported by simultaneous methods are: many degrees of freedom, periodic boundary conditions and the full advantage of an open structure after reformulation such as direct access to first and second order derivatives. This enables to obtain derivatives directly from the problem formulation in an analytical way, whereas in case of single-shooting the first order derivatives can not be accessed directly due to problem reduction and are computed by other strategies such as adjoint-based methods, which are more costly than analytical derivative calculations [12]. Examples of single-shooting methods combined with adjoint-based approaches for gradient computation for optimisation of oil production can be found in [9, 8, 13, 12, 14] .

The simultaneous method investigated in this thesis is called direct collocation and fully discretises the continuous optimal control problem by approximating the controls and states as piecewise polynomial functions on finite elements by applying implicit first order Runge Kutta method (Implicit Euler). This enables to represent the problem as a nonlinear program (NLP). The big instance of the reformulation is then solved by Interior Point Optimiser (Ipopt), a large scale non-linear programming software applying interior point algorithm in the line-search framework. The method directly couples the solution of the reservoir model with the optimisation problem. This can ensure that the differential algebraic equation (DAE) system is solved only once provided that an infeasible path algorithm is used (Sequential Quadratic Programming or Interior Point Method). The direct collocation method has been widely and successfully used in process engineering also for solving downstream problems, and that is the reason why we are motivated in this thesis for exploring its potential for optimising upstream oil production.

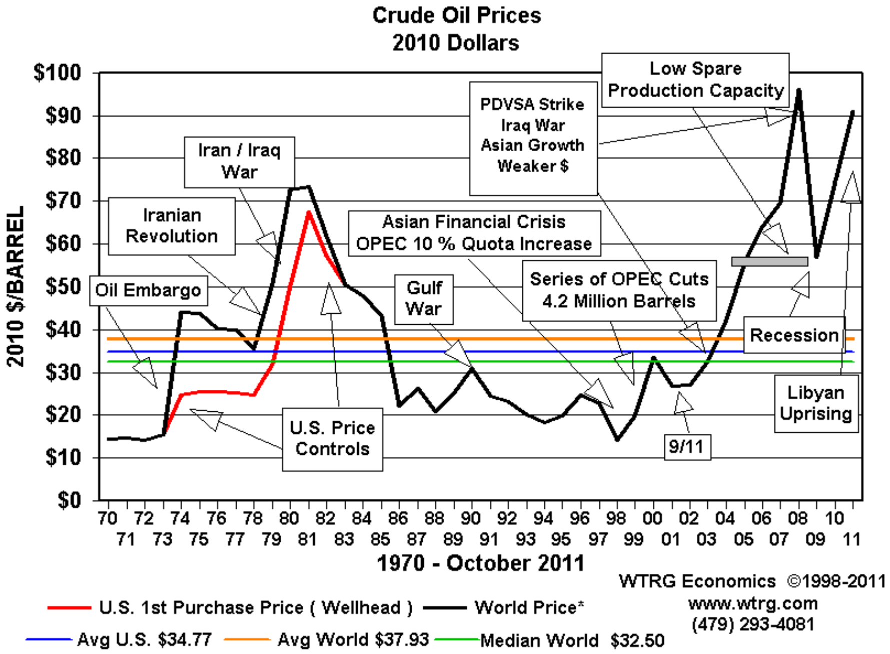
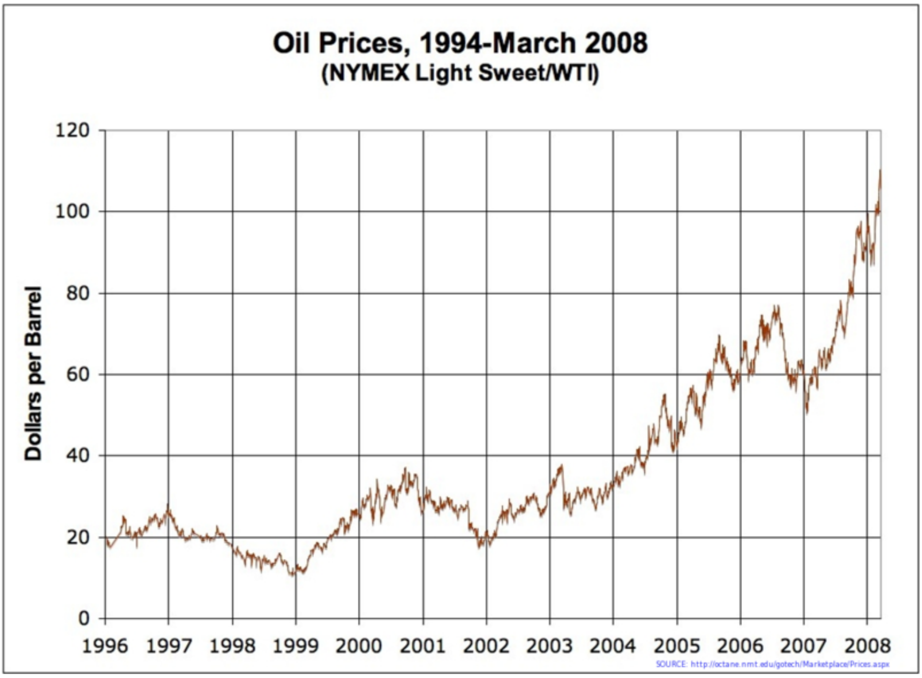


Figure 1.4: Crude Oil Price in American Dollars vs Time since 1970[15]


Oil Prices, 1994-March 2008
(NYMEX Light Sweet/WTI)

SOURCE: http://octane.nmt.edu/gotech/Marketplace/Prices.aspx

Figure 1.5: Crude Oil Price in American Dollars vs Time since 1994 [16]

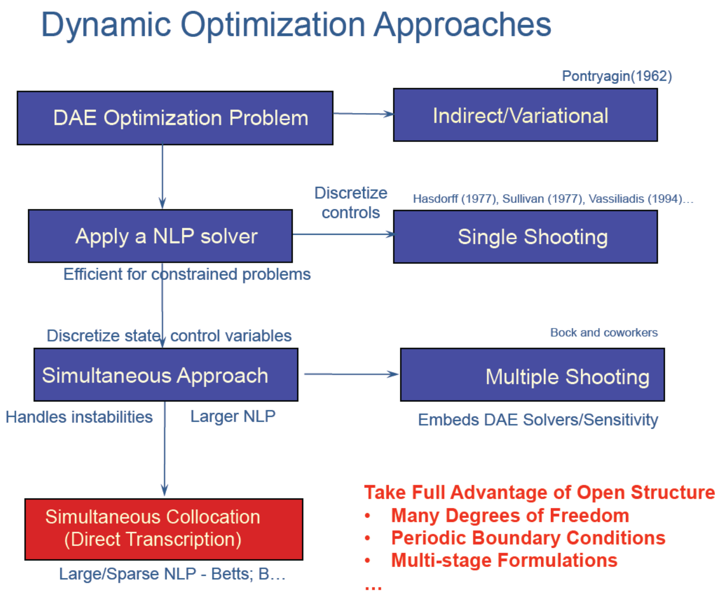

Dynamic Optimization Approaches
Pontryagin(1962)
DAE Optimization Problem
Indirect/Variational
Discretize controls
Hasdorff (1977), Sullivan (1977), Vassiliadis (1994)…
Apply a NLP solver
Single Shooting
Efficient for constrained problems
Discretize state, control variables
Bock and coworkers
Simultaneous Approach
Multiple Shooting
Handles instabilities
Larger NLP
Embeds DAE Solvers/Sensitivity
Simultaneous Collocation (Direct Transcription)
Large/Sparse NLP - Betts; B…
Take Full Advantage of Open Structure
• Many Degrees of Freedom
• Periodic Boundary Conditions
• Multi-stage Formulations
…

Figure 1.6: Dynamic Optimisation Approaches [17]

1.2 Reservoir Management

In order to maximise reservoir performance in terms of oil recovery or another economic objective, reservoir management process is carried out throughout the life cycle of the reservoir, which can be in order of years to decades. An exemplary scheme of this process is presented in the figure 1.7. In many works this scheme might be presented in the slightly different way as the reservoir management process can be enriched or missing some elements depending on the management strategy e.g. in case of open loop reservoir management system models are not updated with data from the sensors through data assimilation algorithms, and whole optimisation is preformed offline. Consequently, this element would not be in the diagram, in case of open loop strategy. What is more, some strategies distinguish between low order and high order system models, which are responsible for uncertainty quantification.

The top element represents the physical system constituting reservoir and well facilities. The central element refers to system models, which consist of static (geological), dynamic(reservoir flow) and well bore flow models. The reason why multiple models are used is because each of them has some uncertain parameters which allow to determine uncertainty about the subsurface. On the right side of the figure we have sensors, which are responsible for keeping the track of the processes that occur in the system. Sensors can be interpreted as physical devices taking measurements of the reservoir

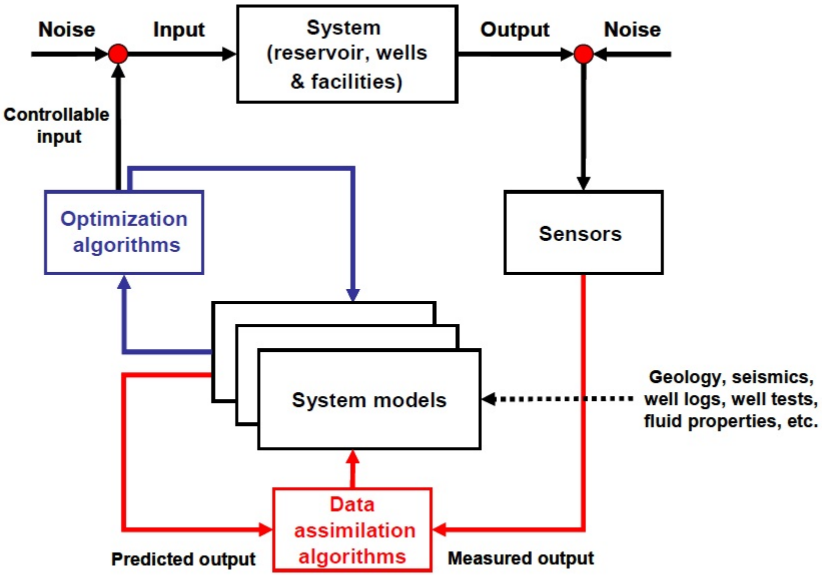

Noise
Input
System (reservoir, wells & facilities)
Output
Noise
Controllable input
Optimization algorithms
Sensors
System models
Geology, seismics, well logs, well tests, fluid properties, etc.
Data assimilation algorithms
Predicted output
Measured output

Figure 1.7: Reservoir Management Process [5]

parameters, such as water or oil saturations and pressures, but they can also be considered in more abstract manner as sources of information about the system variables, e.g. interpreted well tests, time lapse seismics. On the right-hand side of the figure one can find optimisation algorithms, which try to minimise the objective cost based on the set of the constraints obtained from reservoir models. Very important element of the closed-loop reservoir management process are data assimilation algorithms, which obtain the data about the real world from the sensors and then update less realistic models with the more correct information. Data assimilation and model update is performed more frequently than off-line reservoir optimisation as models can easily get off the right track during simulation. As a result, most of reservoir management processes are understood as closed loop ones, and their crucial elements are model based optimisation, decision making and model updating through the data assimilation techniques. Consequently, one can realise himself that model based optimisation, which is the main area of focus in this work, is a very significant element of the reservoir management process.

1.3 Multiple Model Update and Data Assimilation

Data assimilation or computer assisted history matching is a key component of the closed loop reservoir management. The underlying concept of it, is that the mathematical model is not realistic enough to perform the reservoir simulation of the two-phase flow completely independently for many reasons and thus a feedback from the real world is necessary. In general, the system boundaries can be specified accurately for the wells and system facilities but are much more uncertain for the reservoir as its geometry is deduced from seismics that gives limited resolution[18]. Moreover, the parameters of the system are also only known to some certain extent; e.g. the fluid properties can be determined with quite high precision, but the reservoir properties are only really certain at the wells. Another reason why uncertainties of the model might be really significant is because the subsurface is very heterogeneous and the parameters relevant to flow are correlated at different length scales and over distances smaller than inter-well spacing [18]. In order to cope with these physical limitations, multiple subsurface models are constructed to simulate two-phase flow of different geological realisations. In addition, since there are so many uncertainties and unscaled parameters, regular measurements are performed at the top of the wells and in the facilities in oil production process, which give an indication of the pressures and phase rates, e.g. oil and water flow rates. These measurements had been performed monthly or quarterly with limited accuracy. However, recently measurement techniques have been improved by installing sets of sensors that can give almost continuous information about pressures and phase rates not only at the surface but also down-hole [19] [20]. Furthermore, other techniques have emerged, communicating about the changes in reservoir pressure and fluid saturations in between the wells. Consequently, thanks to all these various measurements, reservoir flooding optimisation, based on numerical simulation models can be performed in combination with regular and frequent model updating with the data coming from the sensors. In other words, by combining the measured response of the advanced sensors and the simulated response of the system it is possible to judge to what extent mathematical models represent reality. With the use of data assimilation, it is then possible to adjust the parameters of the individual grid blocks of the numerical models such that the simulated response fits better the measured data. As a result, simulator updated in this way will give more accurate predictions of the future system response. One could argue, that in this manner optimisation combined with data assimilation is a form of non-linear model predictive control with gradually shrinking horizon, however the distinct difference is that data assimilation approach is not aimed at following predefined trajectory as it is in NMPC but rather finding this trajectory to start with. Following this logic, closed loop reservoir management could be classified as a form of real time optimisation (RTO). However, due to the slow dynamics of oil production process and necessity of relatively low frequency of the model updates, it can be performed off-line, contrary to typical RTO [18]. It has to be emphasised however, that enriching reservoir management process by data assimilation and making it closed loop process, results in much frequent updates than conventional reservoir management. Very often the history matching problem itself is formulated as an optimisation problem with an objective function determining the mismatch between measured and simulated output data [21]. The reservoir management as a combination of model-based optimisation and data assimilation is often referred as real-time reservoir management, smart reservoir management as well as closed loop management[22]. Some remarkable works of model-based optimisation of the oil production combined with data assimilation implementing **Kalman filter** can be found in [23, 5, 24].

1.4 Interior Point Methods (IMPs)

This section explains the idea behind the interior point methods, which is utilised by the optimisation tool used in this work.

Since their discovery interior point methods (IPMs) have enjoyed well-deserved interest and have been subjected to intensive study and investigation which led to a development of complete theory and a thorough understanding of their implementation [25]. Interior point or barrier methods are targeted at large scale non-linear optimisation problems. Their recent development was mainly caused by the growing interest of implementation of efficient optimisation algorithms and large scale non-linear programming. This also resulted in better understanding of the convergence properties of interior point methods and development of algorithms with desirable local and global convergence properties. Interior point methods easily generalise from linear, through quadratic to nonlinear programming and for all these types of problems offer efficient algorithms[25]. The main ingredients employed by interior point methods are: backtracking line search, a Newton method for equality constrained minimisation and a barrier function [26]. In addition, these methods offer an attractive alternative to active set strategies. In most cases those methods are implemented in either trust region or line search frameworks (algorithms) and use exact penalty merit functions to iterate towards solution. The mechanism of interior-point method is presented in the figure 1.8. As it can be seen this method always moves within the feasible set which means that it maintains feasibility of decision variables. The only drawback of it is that it might be the case when algorithm converges to the boundary at the limit, i.e. at the point where algorithm terminates. One of the most popular and widely distributed implementations of interior point methods are KNITRO and Ipopt software packages. Ipopt in particular implements primal-dual interior point algorithm with a linesearch filer, whereas KNITRO is based on trust region framework. Both packages give comparable results when tested for large scale optimisation problem; for the results please see[26]. Ipopt and KNITRO have also been used in the reservoir engineering community for oil production optimisation by Suwartadi and Krogstad in [13] where interior point algorithms were combined with adjoint method for gradient computation. In Suwartadi and Krogstad[13] state constraints were removed and added as a barrier term in the objective function. (The main difference between barrier term and penalty term is that

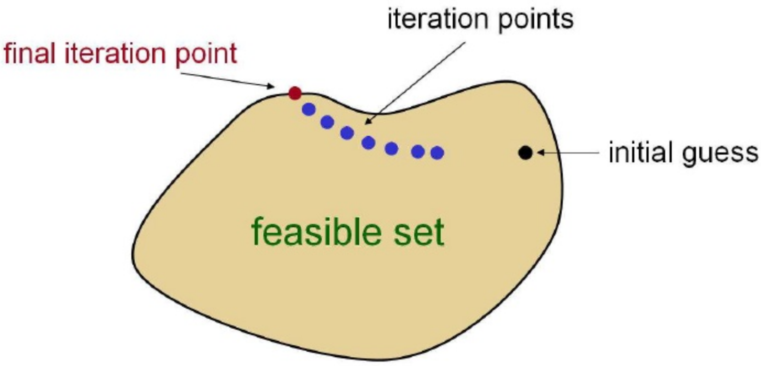


Figure 1.8: Mechanism of Interior-Point Method,taken from [13]

barrier function must use strictly feasible initial guess. This results in a requirement for control input to lie within the feasible set determined by all the constraints of the optimisation problem.) The detailed development of a primal-dual interior point algorithm with a line search filter is presented by Andreas Wächter in [26]. Furthermore, a global convergence of this algorithm is analysed by the same author in [27]. For further information about the algorithm as well as its applications please see [28], [29], [30]

1.5 Multiscale (On-line and Off-line) Optimisation and Operational Aspects

From physical point of view, processes involved in oil production can be classified into upstream and downstream ones; see fig1.9. Downstream processes refer to e.g. pipelines and export facilities, whereas upstream processes are the ones happening in the reservoir e.g. subsurface flows [31]. Those two types of processes differ from each other very distinctively when it comes to their timescales [32]. In the upstream processes the velocity of the fluid can be very slow mainly due to some physical properties of the reservoir such as low permeability value or its size, which can be up to two tens of kilometres. Hence, it can take up to decades to navigate oil by injecting water towards production wells. In case of downstream parts of production, timescales are much lower and can be in order of minutes or even seconds. In this work we focus on optimisation of upstream production, where we model the two phase flow and run so called reservoir simulation. The simulation is based on mathematical models governed by partial differential equations (PDEs, governing equations) and is performed for a long time horizon, even up to decades of years. Consequently the optimisation

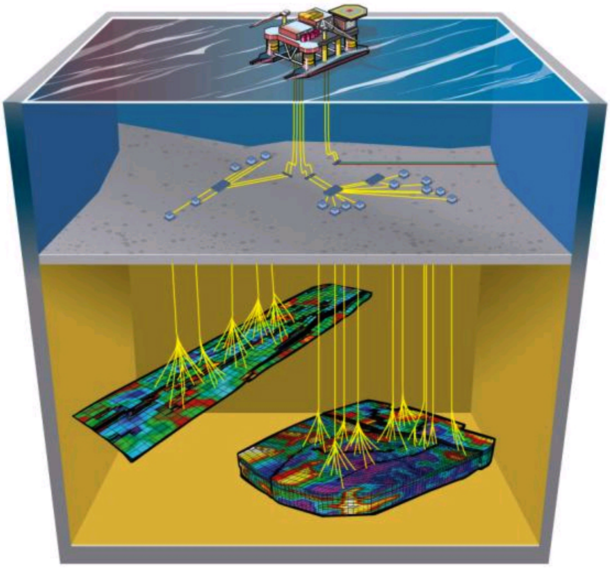


Figure 1.9: Oil Field with Surface and Subsurface Facilities [31]

of upstream part of oil production is run off-line, whereas downstream part is mostly performed on-line. One of the most challenging aspects in closed loop reservoir engineering involves the combination of short-term production optimisation and long-term reservoir management. An open question is, what is the best way of implementing the found, optimal trajectory that was computed off-line into the daily performance of an oil field. Technically, daily valve setting are selected such that they result in instantaneous maximisation of oil production limited by constraints on the processing capacities of gas and water co-produced with the oil. Such settings are mostly determined with heuristics operating protocols, sometimes supported with off-line model based optimisation using sequential or quadratic programming to maximise instantaneous reservoir performance. What is more, a simple, frequent on-line feedback control is used for stabilising the flow rates and pressures in the processing facilities to separate oil, water and gas streams from the wells. It can be seen that there are a few control and optimisation processes going in parallel at different time scales. This kind of strategy involves a layer control structure where longer-term optimisation results provide set points and constraints for the instantaneous, short term optimisation, which then navigates and provides set points for field controllers. This modular approach, also known as multi-scale optimisation, has been widely used in the process industry and was proposed for reservoir management in[32] and [23].

1.6 Linear and Non-linear Model Predictive Control

Model Predictive Control (MPC) is an advanced control strategy that relies on dynamic models of the process for forcasting and optimisaing the system output over some future horizon [33]. The esence of MPC is to use the current plant measurement , and mathematical models to optimise forecasts of the future changes in the dependent variables representing the behaviour of the system. These changes are then used to keep the dependent variables close to target, satisfying constraints on both dependent and independent variables. Model predictive control is defined as receding horizon strategy and is based on iterative finite horizon optimisation of a plant model. The idea behind it, is that only the first step of the control strategy is implemented, then the plant state is sampled again and simulation is started from the new current state. Consequently, the prediction horizon keeps being shifted, yielding a new control and predicted state path [34]. Since the models are not perfect and can get off the track easily, some errors are overcome with a regular feedback and model update. Model Predictive Control originated in 1980s in the chemical process industry but then has been broaden to very wide area of control technology being also attractive to both downstream and upstream processes in oil industry.
Nonlinear Model Predictive Control, or NMPC, is a variant of model predictive control (MPC) and is characterised by non-linear models used directly in the control application. Whereas, in case of regular model predictive control models are linear and even if they are not, then they are linearized over a small operating range to derive Kalman Filter or specify a model for linear MPC [35]. Non-linear Model Predictive Control requires the iterative solution of optimal control problems on a finite prediction horizon. While these problems are convex in linear MPC, in nonlinear MPC they are not convex anymore. The numerical solution of the NMPC optimal control problems is typically obtained by employing direct optimal control methods using Newton-type optimization schemes, in one of the variants: direct single shooting, direct multiple shooting methods, or direct collocation[35]. As a result, the direct collocation approach to optimal control problem of oil production can be utilised as numerical solution in the Non-linear Model Predictive Control framework, with oil reservoir being treated as a plant; fig. 1.7

Thanks to that model predictive control allows practitioners to address trade-offs that should be considered when putting control technology into practice.

1.7 Organisation of this Thesis(Outline)

This thesis consists of 6 chapters and 4 appendices and is organised in the following way:

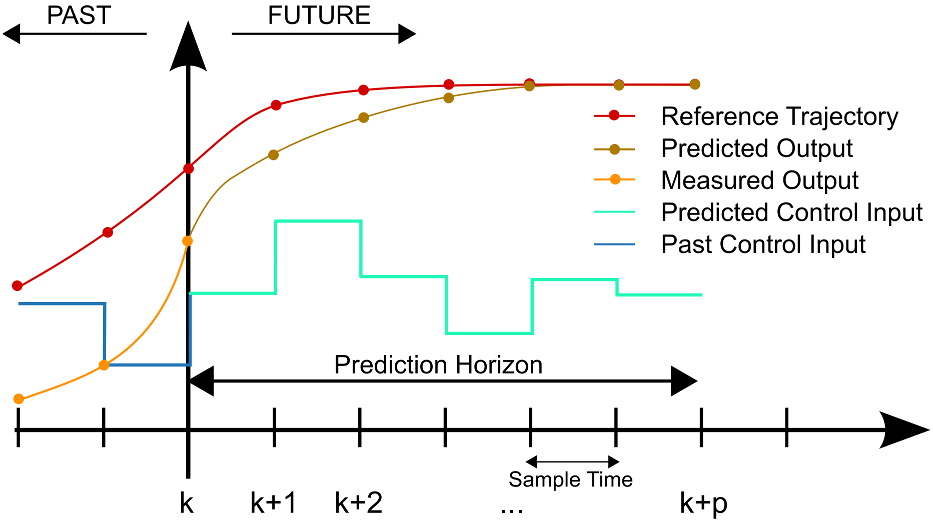


Figure 1.10: Model Predictive Control Mechanism [35]

- In the first chapter we explain the general concept of the reservoir management and give some explanations to key elements of it with empahsis on the optimisation algorithms. What is more, a background of this work is given as well as motivation and main objectives of this research

- In chapter 2 we introduce the tool for large scale nonlinear optimisation Ipopt (interior-point optimiser) by solving a simple non-linear program with its use. Since the implementation of Ipopt is in C++ object oriented programming language we code our exemplary program in Microsoft Visual Studio software development kit.

- In the third chapter we introduce the continuous time Bolza's optimal control problem and present how to solve it by applying direct collocation method with the interior point optimiser on an exemplary problem of Van Der Pol oscillator.

- Chapter four is devoted to the reservoir model, known also as black oil reservoir simulator. In the very beginning, we derive the partial differential equations constituting the continuous mathematical model of the subsurface two phase flow and dynamic constraints on the state variables. Next we present the well models and state transformation from water and oil concentrations to oil pressures and water saturations.

- In chapter five we discretise our model in space by using Gauss theorem and finite volume method and come up with two dimensional model. Then the model is discreitsed in time by using first order implicit scheme (backward Euler method). Finally, necessary information for navigating the optimisation algorithm in the iteration process such as Jacobian of the constraints is retrieved from the reformulation of the model.

- In the chapter six we present the numerical experiment of production optimisation and corresponding results on a particular simulation scenario.
- Chapter seven contains conclusions made throughout the work on this thesis and suggestions for some future work that could be done in the area of simulating two phase flow and oil production optimisation with the simultaneous method.
- Appendix A contains an article written for a journal magazine called Young Petro. The information in the article overlaps to some extent on to what is presented in this thesis. It should be however, treated as a crucial, complementary document that contains the essence and sums up the most important parts of this thesis.
- Paper in appendix B is the manual about setting up Ipopt and loading it as dll (dynamic-link library) into a Visual Studio as a precompiled 3rd party code.
- Appendix C describes how to compute first order derivatives of the properties conserved and flux representation which construct the entries of the Jacobian matrix.
- Appendix D is the list of conferences and scientific workshops to which results from this project were submitted and presented in form of a poster or oral presentation.
- Appendix E is the contract, written after the case study in the field related was done. The contract defines the workload that should be done in order to accomplish this project and fulfil requirements to obtain a degree of master of science.

Chapter 2

Solving Simple Nonlinear Problem Using Interior Point Optimiser

In this chapter we will describe the software package that will be used for tackling oil problem as we proceed further in our research. In order to do this, we will present a simple, exemplary non-linear program and instruct how to solve it using the proposed ptimisation tool. The investigated solver is called Ipopt and is a software package implementing interior point line search filter methods that find a local solution of general nonlinear program of the following form:

$$\min_{x \in R^n} f(x) \tag{2.0.1a}$$

$$s.t. \quad g^L \leq g(x) \leq g^U \tag{2.0.1b}$$

$$x^L \leq x \leq x^U \tag{2.0.1c}$$

where:

- $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the objective function.
- $x \in \mathbb{R}^n$ are the optimisation variables.
- $x^L \in (\mathbb{R} \cup (-\infty))^n$ are the lower bounds on optimisation variables.

- $x^U \in (\mathbb{R} \cup (+\infty))^n$ are upper bounds on optimisation variables.
- g(x) is a constraint function.
- $g^L \in (\mathbb{R} \cup (-\infty))^n$ are lower bounds on constraints g(x)
- $g^U \in (\mathbb{R} \cup (+\infty))^n$ are upper bounds on constraints g(x)

Remarks:
The objective function $f(x)$ and general constraints $g(x)$ can be linear or non-linear and convex or non-convex. They should be however, two times differentiable and satisfy KKT conditions. In case of modelling constraint equality conditions, one should set $g_i^L = g_i^U = g_i$. We put a special emphasis on equations (2.0.1) since they represent a problem format readable for an Ipopt. As a result, to solve any kind of a problem, no matter on its format, one has to use a method which will transcribe it so that it follows those three equations. Ipopt package is an open source one and is available at COIN-OR initiative (www.coin-or.org). It is maintained in C++ and Fortran and can be loaded as a precompiled dynamic link library (DLL) into many different technologies such us Matlab, Python, C and C++. In our case, Ipopt dll distribution was configured and loaded into Microsoft Visual Studio 2008 project. In order to get information on obtaining the Ipopt package and setting it up in Visual Studio, please go to tutorial about Ipopt in Visual Studio in appendix B section.

2.1 Simple Non-linear Program (NLP)

Once Ipopt is set up in Visual Studio , one can represent his own nlp and interact with a solver through a very straightforward C++ interface. To demonstrate how to do this we introduce a simple non-linear program of the following form:

$$\min_{x \in R^n} f(x) = -(x_2 - 2)^2 \tag{2.1.1a}$$

$$s.t. \quad 0 = x_1^2 + x_2 - 1 \tag{2.1.1b}$$

$$-1 \leq x_1 \leq 1 \tag{2.1.1c}$$

Ipopt however, needs a bit more information about the problem than its standard formulation, e.g. Jacobian of the constraints or gradient of the objective. Below additional problem information, required by Ipopt to solve it, is listed.

1. Dimensions of the Problem
   - number of variables
   - number of constraints

2. Bounds of the Problem
    - variable bounds
    - constraint bounds
3. Initial Starting Point
    - Initial values for the primal x variables
    - Initial values for the multipliers
4. Problem Structure
    - number of the non-zeros of the Jacobian of the constraints
    - number of the non-zeros of the Hessian of the Lagrangian function
    - sparsity structure of the Jacobian of the constraints
    - sparsity structure of the Hessian of the Lagrangian function
5. Evaluation of Problem Functions
    - Objective function $f(x)$
    - Gradient of the objective $\triangledown f(x)$
    - Constraint function values $g(x)$
    - Jacobian of the constraints $\triangledown g(x)^T$
    - Hessian of the Lagrangian function $\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$

As we can see, Ipopt needs a bit more information than straight problem specification. At the first glance, one can think, that it is Ipopt disadvantage over some other optimisation packages which are able to find the solution after specifying only the objective function and the feasible set. Ipopt is however, designed for large scale problems and consequently, passing all this additional information pays off as it speeds up calculation time and saves up memory resources.

We will now retrieve from our exemplary nlp this additional information, and pass it to Ipopt through the standardised C++ interface. While some of the data can be red directly from problem formulation, e.g. number of variables or variable bounds, let us concentrate on the more complex equations defined in point 5. We start with the gradient of the objective function, which is:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \tag{2.1.2a}$$

$$\nabla f(x) = \begin{bmatrix} 0 \\ -2 \cdot (x^2 - 2) \end{bmatrix} \tag{2.1.2b}$$

and the Jacobian of the constraints $g(x)$ is:

$$\nabla g(x)^T = \begin{bmatrix} \frac{\partial g}{\partial x_1} & \frac{\partial g}{\partial x_2} \end{bmatrix} \tag{2.1.3a}$$

$$\nabla g(x)^T = \begin{bmatrix} -2x_1 & -1 \end{bmatrix} \tag{2.1.3b}$$

and the Hessian of the Lagrangian is the following:

$$\begin{aligned} &\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x) = \\ &= \sigma_f \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial^2 x_2} \end{bmatrix} + \lambda_1 \begin{bmatrix} \frac{\partial^2 g}{\partial^2 x_1} & \frac{\partial^2 g}{\partial x_1 \partial x_2} \\ \frac{\partial^2 g}{\partial x_1 \partial x_2} & \frac{\partial^2 g}{\partial^2 x_2} \end{bmatrix} \end{aligned} \tag{2.1.4}$$

After calculating second partial derivatives of the objective function and equality constraint one can get:

$$\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x) = \sigma_f \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix} + \lambda_1 \begin{bmatrix} -2 & 0 \\ 0 & 0 \end{bmatrix} \tag{2.1.5}$$

Where the first term comes from the objective function, and the second term is from the Hessian of the constraints(λ variable in front of the Hessian term is a constraint multiplier). Our exemplary nlp is constrained only by one equality constraint, hence we have only one Hessian of the constraints matrix. Our later examples will be constrained in a more complex way which will result in more λ terms, coming from the constraints, in the Hessian of the Lagrangian.Please note that in general the Lagrangian for the nlp is given by:$f(x) + g(x)^T \lambda$ and the Hessian of the Lagrangian is $\nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$. Ipopt format however, introduces additional term in front of portion coming from objective function σ_f so that it can ask for Hessian of the objective or the constraints independently, whenever these are needed.

While Ipopt supports many different matrix formats, triple format was chosen for coding matrices (Hessian of the Lagrangian and Jacobian of the constraints) of this nlp. In the triple format user should specify only non-zero entries of the matrices and their corresponding sparsity structures. Since those non-zero structures should remain constant throughout the whole solving process, user should make sure to include entries for any element that might ever be non-zero, not only those that are non-zero at the starting point (in other words only elements which are always zero should be omitted). For further information about triple matrix format please see [36].

Having retrieved all the required information about our nlp, we can go forward to the next step, which is coding the problem in C++ and passing it to Ipopt.

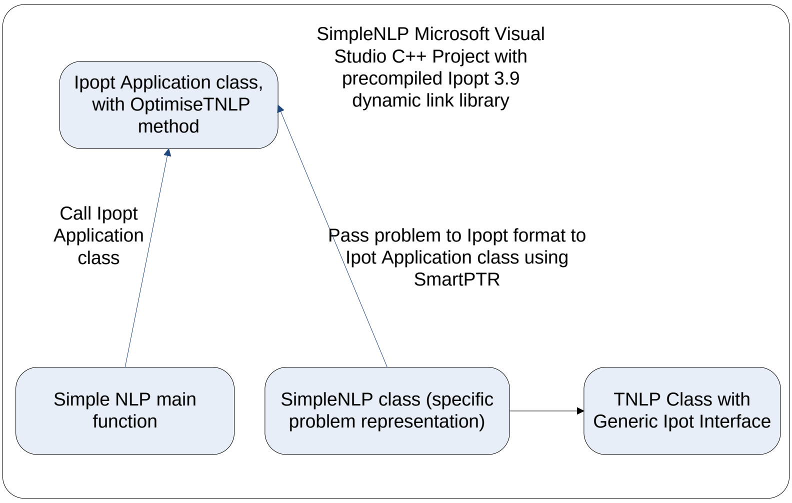

SimpleNLP Microsoft Visual Studio C++ Project with precompiled Ipopt 3.9 dynamic link library
Ipopt Application class, with OptimiseTNLP method
Call Ipopt Application class
Pass problem to Ipopt format to Ipot Application class using SmartPTR
Simple NLP main function
SimpleNLP class (specific problem representation)
TNLP Class with Generic Ipot Interface

Figure 2.1: Simple NLP Visual Studio Project Structure

2.2 Ipopt C++ Interface

Implementation of the optimisation problem in Ipopt uses object oriented features of C++ language and can be divided into two following stages:

- Coding the problem representation in the class inheriting from the abstract TNLP class

- Coding the executable main function and calling Ipopt through the IpoptApplication class.

For getting familiar with the project structure in the Visual Studio 2008, please see figure 2.1.

2.2.1 Coding the Problem Representation

Ipopt is equipped with generic interface consisting of prototyped methods defined in abstract TNLP class. In order to tell Ipopt about given optimisation problem, one is required to create a class inheriting from TNLP class and implement its interface by overriding all the abstract methods. The Ipopt-TNLP interface is presented below. As it can be seen, all the methods have been declared as pure virtual ones by being marked with ”= 0” notation. This means that they do not have their bodies and programer has to define them himself in the child class(in our case SimpleNLP class). Now we will explain the functionality of each virtual method and present its definition in the context of the SimpleNLP implementation (SimpleNLP.cpp implementation file).

```
virtual bool get_nlp_info (Index& n, Index& m, Index& nnz_jac_g,
                           Index& nnz_h_lag,
                           IndexStyleEnum& index_style)=0;

/** Method to return the bounds for my problem */
virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                             Index m, Number* g_l, Number* g_u)=0;

/** Method to return the starting point for the algorithm */
virtual bool get_starting_point(Index n, bool init_x,  Number* x,
                                bool init_z, Number* z_L, Number* z_U,
                                Index m, bool init_lambda,
                                Number* lambda)=0;

/** Method to return the objective value */
virtual bool eval_f(Index n, const Number* x, bool new_x,
                    Number& obj_value)=0;

/** Method to return the gradient of the objective */
virtual bool eval_grad_f(Index n, const Number* x,
                         bool new_x, Number* grad_f)=0;

/** Method to return the constraint residuals */
virtual bool eval_g(Index n, const Number* x, bool new_x,
                    Index m, Number* g)=0;

/** Method to return:
* 1) The structure of the jacobian (if "values" is NULL)
* 2) The values of the jacobian (if "values" is not NULL)
*/
virtual bool eval_jac_g(Index n, const Number* x, bool new_x,
                        Index m, Index nele_jac, Index* iRow,
                        Index *jCol, Number* values)=0;

virtual bool eval_h(Index n, const Number* x, bool new_x,
                    Number obj_factor, Index m, const Number* lambda,
                    bool new_lambda, Index nele_hess, Index* iRow,
                    Index* jCol, Number* values)=0;

/** Solution Methods */
/** This method is called when the algorithm is complete so the TNLP can
store/write the solution */
virtual void finalize_solution(SolverReturn status, Index n,
                               const Number* x, const Number* z_L,
                               const Number* z_U, Index m,
                               const Number* g,
                               const Number* lambda, Number obj_value,
                               const IpoptData* ip_data,
                               IpoptCalculatedQuantities* ip_cq)=0;
```

Method get_nlp_info with a prototype:

```
bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                  Index& nnz_h_lag, IndexStyleEnum& index_style)
```

Returns to Ipopt information about the size of the problem, so that it knows how much memory it should allocate.

- n:(out), the dimension of the x optimisation variable
- m:(out), the dimension of the vector constraint function g(x)
- nnz_jac_g:(out), the number of the non-zero entries in the Jacobian matrix
- index_style: (out), the indexing style for row and column entries of the matrices in the sparse format. It can be set to C_STYLE or FORTRAN_STYLE, which means start indexing from 0 or 1 respectively.
- nnz_h_lag:(out), the number of the non-zero entries of the Hessian of the Lagrangian matrix.

```
bool SimpleNLP::get_nlp_info(Index& n, Index& m,
                             Index& nnz_jac_g, Index& nnz_h_lag,
                             IndexStyleEnum& index_style)
{
  // The problem described in MyNLP.hpp has 2 variables,
  //x1, & x2,
  n = 2;

  // one equality constraint,
 m = 1;

  //in this case Jacobian has 2 nonzero entries
  //(one for x1, and one for x2),
  nnz_jac_g = 2;

  // the hessian of the Lagrangian also has two non-zero entries
  //one in the Hessian of the objective for x2
  //and one in the Hessian of the constraints for x1
  nnz_h_lag = 2;

  // We use the standard C style for row/col entries
  index_style = C_STYLE;

  return true;
}
```

Method get_bounds_info with a prototype:

```
bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                     Index m, Number* g_l, Number* g_u);
```

Returns to the Ipopt information about the bounds on the optimisation variables and constraints.

- n:(in), the dimension of x variables vector.
- x_l:(out), array with lower bounds on x variable vector .
- x_u:(out), array with upper bounds on x variable vector.
- m:(in), the dimension of the constraint function g(x) of the problem.
- g_l:(out), array with lower bounds on the constraint function g(x).
- g_u:(out), array with upper bounds on the constraint function g(x).

The m and n variables in the get_bounds_info method are passed back so that one can check the correctness of their values by using ”assert” macro. Since variable x_1 has lower bound of -1 and upper bound of 1, we set x_l[0] and x_u[0] elements to -1 and 1 respectively. Variable x_2 is not bounded in anyway and consequently, we set x_l[1] and x_u[1] elements to numbers which are interpreted by Ipopt as positive and negative infinities by the default. It is also possible for the user to specify which values Ipopt should understand as lack of bound. In order to do it , one should set the nlp_upper_bound_inf option to a given value. Then any value which is equal or greater than nlp_upper_bound_inf will be taken by Ipopt as no-upper bound. The same thing can be done for nlp_lower_bound_inf. However this time any lower or equal value will be understood as no-lower bound. For more detailed information on setting default values for positive and negative infinities please see [36].

Since nlp_lower_bound_inf and nlp_upper_bound_inf are set by default to -10^{19} and 10^{19} respectively, we also assigned these values to x_l[1] and x_u[1] in the get_bounds_info method. When it comes to constraints g(x) in our exemplary problem, we have only one equality constraints, so we set bounds on this constraint to be equal and zero. That is the way Ipopt recognises equality constraints format and as a result, does not treat it as two inequalities.

```
bool SimpleNLP::get_bounds_info(Index n, Number* x_l,
                                Number* x_u, Index m,
                                Number* g_l, Number* g_u)
{
  // here, the n and m we gave IPOPT in get_nlp_info
  //are passed back to us.  If desired, we could assert
  //to make sure they are what we think they are.
  assert(n == 2);
  assert(m == 1);

  //x1 has a lower bound of -1 and an upper bound of 1
  x_l[0] = -1.0;
  x_u[0] = 1.0;

  // x2 has no upper or lower bound, so we set them to
  //a large negative and a large positive number.
  //The value that is interpretted as -/+infinity can be
  //set in the options, but it defaults to -/+1e19
  x_l[1] = -1.0e19;
  x_u[1] = +1.0e19;

  //we have one equality constraint, so we set the bounds on
  //this constraint to be equal (and zero).
  g_l[0] = g_u[0] = 0.0;

  return true;
}
```

Method get_starting_point with a prototype:

```
bool get_starting_point(Index n, bool init_x, Number* x,
                        bool init_z, Number* z_L, Number* z_U,
                        Index m, bool init_lambda,
                        Number* lambda);
```

Returns the Ipot starting point at which, the solver starts iterating.

- n:(in), the dimension of x variable vector.
- init_x:(in), boolean flag saying whether initial values of the bound multipliers z^L and z^U are specified or not. If it is set to true then user has to specify the intial z^L and z^U.

- z_L:(out), array with initial values of lower bound multipliers.
- z_U:(out), array with initial values of upper bound multipliers.
- m:(in), the dimension of the vector constraint function g(x) of the problem.
- init_lambda:(in), boolean flag saying whether initial values of the constraint multipliers lambda are specified or not. If it is set to true, then user has to specify the initial λ multipliers .
- lambda: (out), array with initial values for the constraint λ multipliers .

```
bool SimpleNLP::get_starting_point(Index n, bool init_x,
                                   Number* x, bool init_z,
                                   Number* z_L, Number* z_U,
                                   Index m, bool init_lambda,
                                   Number* lambda)
{
  // Here, we assume we only have starting values for x,
  //if you code your own NLP, you can provide starting
  //values for the others if you wish.
  assert(init_x == true);
  assert(init_z == false);
  assert(init_lambda == false);

  // we initialize x in bounds, in the upper right quadrant
  x[0] = 0.5;
  x[1] = 1.5;

  return true;
}
```

In the first two lines, we make again sure that we passed the right values of the numbers of the constraints and variables. Then we set init_x flag to true and init_z and init_lambda flags to false. Finally, we assign starting point for x-variable vector in the upper right quadrant.

Method eval_f with a prototype:

```
bool eval_f(Index n, const Number* x, bool new_x,
            Number& obj_value)
```

Returns the value of the objective function at the point x.

- n:(in), the dimension of x-variable vector.

- x:(n),array with x-variables for which objective function is evaluated.
- new_x:(in), flag which is passed from Ipopt to eval_f method. new_*x* is false if any evaluation method was previously called for the same values in x array. This flag can be used for speeding up the implementation which calculates many outputs at one time. Since in this problem single output is evaluated , new_x is not used.
- obj_value:(out), the value of the objective function at point x.

```
bool SimpleNLP::eval_f(Index n, const Number* x, bool new_x,
                       Number& obj_value)
{
  // return the value of the objective function
  Number x2 = x[1];
  obj_value = -(x2 - 2.0) * (x2 - 2.0);
  return true;
}
```

Method eval_grad_f with a prototype:

```
bool eval_grad_f(Index n, const Number* x, bool new_x,
                 Number* grad_f)
```

Returns the gradient of the cost function at point x.

- n:(in), the dimension of x-variable vector.
- x:(n),array of x-variables for which gradient of the objective function evaluated.
- new_x:(in), flag which is passed from Ipopt to eval_f method. new_*x* is false if any evaluation method was previously called for the same values in x array. This flag can be used for speeding up the implementation which calculates many outputs at one time. Since we evaluate single output in our problem, new_x is not used.
- grad_f:(out), array with values of the gradient of the objective cost function. For this particular problem, gradient was determined in equation (2.1.2)

The gradient array should be ordered in the same way as single variables in the x vector, e.g. gradient with respect to x[1] should be placed in grad_f[1] element.

```
bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x,
                        Number* grad_f)
{
  // return the gradient of the objective function grad_{x} f(x)

  // grad_{x1} f(x): x1 is not in the objective
  grad_f[0] = 0.0;

  // grad_{x2} f(x):
  Number x2 = x[1];
  grad_f[1] = -2.0*(x2 - 2.0);

  return true;
}
```

Method eval_g with a prototype:

```
bool eval_g(Index n, const Number* x, bool new_x, Index m,
            Number* g)
```

Returns the array with values of the constraint function $g(x)$ at point x.

- n:(in), the dimension of x-variable vector.
- x:(n),array of x-variables for which gradient of the objective function is evaluated.
- new_x:(in), flag which is passed from Ipopt to eval_f method. new_x is false if any evaluation method was previously called for the same values in x array.
- m:(in), the dimension of the constraint function $g(x)$ of the problem.
- g:(out), the array with values of the constraints function $g(x)$.

As our exemplary problem is constrained by only one equality constraint, we express it in terms of x-vector variable and assign it to g[0] in the body of eval_g function.

```
bool SimpleNLP::eval_g(Index n, const Number* x, bool new_x,
                       Index m, Number* g)
{
  // return the value of the constraints: g(x)
  Number x1 = x[0];
  Number x2 = x[1];

  g[0] = -(x1*x1 + x2 - 1.0);

  return true;
}
```

Method eval_jac_g with prototype

```
bool eval_jac_g(Index n, const Number* x, bool new_x,Index m,
                Index nele_jac, Index* iRow, Index *jCol,
                Number* values)
```

Returns either the values at point x, or the sparsity structure of the Jacobian of the constraints. Both values and sparsity structure refer only to the non-zero elements.

- n:(in), the dimension of x-variable vector.
- x:(n),array of x-variables for which gradient of the objective function is evaluated.
- new_x:(in), flag which is passed from Ipopt to eval_f method. new_x is false if any evaluation method was previously called for the same values in x array. This flag can be used for speeding up the implementation which calculates many outputs at one time. Since we evaluate single output in our problem, new_x is not used.
- m:(in), the dimension of the vector constraint function $g(x)$ of the problem.
- n_ele_jac: (in), number of non-zero elements of the Jacobian (this variable should be equal to the dimension of irow, jcol and values arrays)
- iRow:(out), row indices of the non-zero entries in the Jacobian of the constraints.
- jCol:(out), column indices of the non-zero entries in the Jacobian of the constraints.
- values:(out), the values of the non-zero entries in the Jacobian of the constraints.

The function eval_jac_g should always be implemented according to the specific rules. As it was mentioned before, eval_jac_g returns either sparsity structure or the values of the Jacobian. The decision, which parameter is be evaluated and returned should be done by eval_jac_g based on the value of values array, which is passed to the function from Ipopt. If values array is NULL, then Ipopt expects eval_jac_g to return the sparsity structure of the Jacobian stored in the iRow and jCol arrays. If values array is not NULL, then the actual values of the Jacobian at point x should be assigned into values array. The functionality of m, n, and new_x in-parameters is the same as for previous methods, and their usage is completely optional.

```
bool SimpleNLP::eval_jac_g(Index n, const Number* x, bool new_x,
                           Index m, Index nele_jac, Index* iRow,
                           Index *jCol, Number* values)
{
  if (values == NULL) {
    // return the structure of the jacobian of the constraints

    // element at 1,1: grad_{x1} g_{1}(x)
    iRow[0] = 1;
    jCol[0] = 1;

    // element at 1,2: grad_{x2} g_{1}(x)
    iRow[1] = 1;
    jCol[1] = 2;
  }
  else {
    // return the values of the jacobian of the constraints
    Number x1 = x[0];

    // element at 1,1: grad_{x1} g_{1}(x)
    values[0] = -2.0 * x1;

    // element at 1,2: grad_{x1} g_{1}(x)
    values[1] = -1.0;
  }

  return true;
}
```

Method finalize_solution with prototype:

```
void finalize_solution(SolverReturn status,Index n,
                       const Number* x, const Number* z_L,
                       const Number* z_U, Index m,
                       const Number* g, const Number* lambda,
                       Number obj_value,
                       const IpoptData* ip_data,
                       IpoptCalculatedQuantities* ip_cq)
```

- status(int), informs about the return status of the algorithm. Below we present some possible values of this enumeration: -SUCCESS - algorithm terminated successfully at the local minimiser, which satisfies the convergence tolerances.
  -LOCAL_INFEASIBILITY - algorithm converged to a point which is locally infeasible and whole problem might be infeasible.
  -RESTORATION_FAILURE - algorithm does not know how to proceed, which very often suggests that the problem is implemented in a wrong way (C++ representation is not consistent with actual problem).

- n:(in), the dimension of x-variable vector.

- x:(n),array of x-variables storing final optimised values (local minimiser).
- z_L:(in), the final values for the lower bound multipliers z_*^L.
- z_U:(in), the final values for the upper bound multipliers z_*^U.
- m:(in), the dimension of the constraint function $g(x)$ of the problem.
- g:(in), the final value of the constraint function $g(x)$ at point x_*.
- lambda: (in), the final values of the constraint multipliers λ.
- obj_value:(in), the final value of the objective function f at the point x_*

For more information about return status variable please see section 3.3.1 at [36]

```
void MyNLP::finalize_solution(SolverReturn status, Index n,
                              const Number* x,const Number* z_L,
                              const Number* z_U, Index m,
                              const Number* g,
                              const Number* lambda,
                              Number obj_value,
                              const IpoptData* ip_data,
                              IpoptCalculatedQuantities* ip_cq)
{
  // here is where we would store the solution to variables,
  //or write to a file, etc  so we could use the solution. Since
  //the solution is displayed to the console, we currently do
  //nothing here.
}
```

Please note that all the parameters in finalize_sulution method are ”in - ones”. It is so, because this method is called by Ipopt at the final stage when solution has been found. As a result, Ipopt is passing the return status and the values of the variables , the objective and the constraints, when algorithm excites. Implementation of this method is very flexible and depends on how the user wants to present solution of the problem, e.g. it is possible to use cout class or printf function to display the final values of the x vector variable and constraints, or write this data into a text file with the use of ostream class. Because the solution is already displayed in the main function, it is not necessary to use finalize_solution method in this particular example. It still ,however, has to be implemented since this method was declared as pure virtual one in the parent class, and it would not be possible to compile the program without its definition.
So far we have described all the pure virtual methods of the TNLP interface that user has to implement in the child class to pass the optimisation problem to Ipopt. Last method that we would like to focus on is eval_h and its usage is completely optional.

It is recommended however, to implement it, as eval_h evaluates second derivatives, which speeds up the operation and improves robustness of the algorithm, e.g. it can converge faster.

Method eval_h with prototype:

```
bool eval_h(Index n, const Number* x, bool new_x,
            Number obj_factor, Index m, const Number* lambda,
            bool new_lambda, Index nele_hess, Index* iRow,
            Index* jCol, Number* values);
```

Returns either the sparsity structure or the values of the Hessian of the Lagrangian.

- n:(in), the dimension of x-variable vector.
- x:(n),array of x-variable vector for which values of the Hessian of the Lagrangian are evaluated.
- new_x:(in), flag which is passed from Ipopt to eval_h method. new_x is false if any evaluation method was previously called for the same values in x array. This flag can be used for speeding up the implementation which calculates many outputs at one time. Since we evaluate single output in our problem, new_x is not used.
- obj_factor:(in), factor σ in front of the term coming from objective function; please see eq. (2.1.4)
- m:(in), the dimension of the constraint function $g(x)$ of the problem.
- lambda:(in), array with the values of the constraint multipliers λ for which Hessian of the Lagrangian should be evaluated.
- new_lambda:(in),flag which is passed from Ipopt to eval_h method. new_lambda is false if any evaluation method was previously called for the same values in lambda array.
- nele_hess:(in), the number of non-zero elements in the Hessian of the Lagrangian (nele_hess should be equal to the dimension of irow, jcol and values arrays).
- iRow:(out), row indices of the non-zero entries in the Hessian of the Lagrangian.
- jCol:(out), column indices of the non-zero entries in the Hessian of the Lagrangian.
- values:(out), the values of the non-zero entries in the Hessian of the Lagrangian.

Eval_h should be implemented following the same logic as eval_jac_g, which means that the function has to decide whether to return the values or the sparsity structure of the Hessian of the Lagrangian, depending on the value of values array. If values array is equal to NULL and iRow and jCol are not NULL at the same time, then Ipopt expects to obtain the sparsity structure of the Hessian of the Lagranian. If values array is not NULL, then iRow and jCol are NULL and Ipopt awaits for the values of the Hessian of the Lagrangian evaluated in terms of x, obj_factor and lambda. obj_factor and lambda should be used in the same order as shown in the equation (2.1.5), which means that when evaluating the values of the matrix, the elements coming from objective function term should be multiplied by obj_factor and elements coming from the Hessian of the constraints should be multiplied by corresponding element of lambda array(in our case , there is only one constraint so all the non-zero elements of the Hessian of the constraints $\sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$ are multiplied by first element of lambda array - lambda[0].

For representing the sparsity structure, a sparse symmetric matrix format is used, which basically means that only non-zero elements of the lower left corner are specified as Hessian of the Lagrangian is symmetric. For information and examples on representing matrices in sparse triple format please see [36].

```
bool MyNLP::eval_h(Index n, const Number* x, bool new_x,
                   Number obj_factor, Index m,
                   const Number* lambda, bool new_lambda,
                   Index nele_hess, Index* iRow,
                   Index* jCol, Number* values)
{
   if (values == NULL)
   {
    // return the structure. This is a symmetric matrix,
    //fill the lower left
    // triangle only.

    // element at 1,1: grad^2_{x1,x1} L(x,lambda)
    iRow[0] = 1;
    jCol[0] = 1;

    // element at 2,2: grad^2_{x2,x2} L(x,lambda)
    iRow[1] = 2;
    jCol[1] = 2;

    // Note: off-diagonal elements are zero for this problem
   }
  else
  {
    // return the values

    // element at 1,1: grad^2_{x1,x1} L(x,lambda)
    values[0] = -2.0 * lambda[0];
    // element at 2,2: grad^2_{x2,x2} L(x,lambda)
    values[1] = -2.0 * obj_factor;

    // Note: off-diagonal elements are zero
    //for this problem
  }

  return true;
}
```

2.2.2 Coding the Executable Main Function

In the second step of solving exemplary problem, we will create main function in which we instantiate our SimpleNLP class and store the address of the SimpleNLP object in a smart pointer. Then we instatiate an IpoptApplication class by calling IpoptApplicationFactoryMethod (SmartPtr<IpoptApplication> app = IpoptApplicationFactory();). Using IpoptFactoryMethod is necessary for the Visual Studio implementation as it enables the user to compile the program with Ipopt Windows dll. Now, we have object of the class IpotApplication, which we can utilise by accessing its methods via smart pointer with arrow operator (app->method). Firstly, we call Initialize() method by typing app->Initialize(), and then after checking that initialization was performed successfully (if (status != Solve_Succeeded)), we call OptimizeTNLP(TNLP*) method and pass address of SimpleNLP object to it (OptimizeTNLP(mynlp)). OptimizeTNLP (TNLP*) method returns status variable, whose value we check to make sure, that Ipopt was successful in finding solution. Final stage is displaying results on the console window. To obtain results from Ipopt, we call the following methods in IpoptApplication class:

- IpoptApplication->Statistics()->IterationCount() - returns number of iterations in which solution was found.
- IpoptApplication->Statistics()->FinalObjective() - returns the final value of the objective function when local solution was found.

Of course, it is possible to obtain many other parameters from Ipopt about the solution, e.g. bound multipliers, values of the constraints at the optimal point or final values of the primal variables x_*, by either calling different methods in IpoptApplication class or implementing finalize_solution method.

```
int main(int argv, char* argc[])
{
  // Create an instance of SimpleNLP
  SmartPtr<TNLP> ptr_nlp = new SimpleNLP();

  // Create an instance of the IpoptApplication
  // using Ipopt ApplicationFactory() method
  SmartPtr<IpoptApplication> app = IpoptApplicationFactory();

  // Initialize the IpoptApplication and process the options
  ApplicationReturnStatus status;
  status = app->Initialize();
  if (status != Solve_Succeeded) {
    printf("Error during initialization!");
    return (int) status;
  }

  status = app->OptimizeTNLP(ptr_nlp);

  if (status == Solve_Succeeded) {
    // Retrieve some statistics about the solve
    Index iter_count = app->Statistics()->IterationCount();
    printf
    ("\n*****************************************");
    printf
    ("\n The problem solved in %d iterations!\n", iter_count);

    Number final_obj = app->Statistics()->FinalObjective();
    printf
    ("\nThe value of the objective function is%e.\n",final_obj);
  }
  return (int) status;
}
```

2.2.3 Results and Ipopt Output

After implementing the main function, one can compile and run the program. Since we are using Visual Studio 2008, we do not have to use make files and tell the linker about Ipopt library as this was already set up. In order to build the project and run the program one needs to press key f5 in the Visual Studio SDK. This command runs the program by default and asks the user for building the project if it is out of date. If compilation proceeded successfully, one should get the following output in the console window after running SimpleNLP project.

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Common Public License (CPL).
         For more information visit http://projects.coin-or.org/Ipopt
```

```
******************************************************************************

NOTE: You are using Ipopt by default with the MUMPS linear solver.
      Other linear solvers might be more efficient (see Ipopt documentation).


This is Ipopt version 3.9.1, running with linear solver mumps.

Number of nonzeros in equality constraint Jacobian...:        2
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:        2

Total number of variables............................:        2
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        1
                     variables with only upper bounds:        0
Total number of equality constraints.................:        1
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0 -2.5000000e-001 1.48e+000 7.97e-001  -1.0 0.00e+000    -  0.00e+000 0.00e+000   0
   1 -3.7436285e+000 5.22e-004 1.68e-001  -1.0 1.43e+000    -  1.00e+000 1.00e+000f  1
   2 -3.9841853e+000 9.84e-004 7.68e-003  -1.7 6.12e-002    -  1.00e+000 1.00e+000f  1
   3 -3.9998793e+000 2.17e-006 2.32e-005  -3.8 3.93e-003    -  1.00e+000 1.00e+000h  1
   4 -3.9999982e+000 1.90e-010 1.64e-009  -5.7 2.97e-005    -  1.00e+000 1.00e+000h  1
   5 -4.0000001e+000 5.28e-014 4.23e-013  -8.6 4.60e-007    -  1.00e+000 1.00e+000h  1

Number of Iterations....: 5

                                   (scaled)                 (unscaled)
Objective...............:  -4.0000000774945192e+000   -4.0000000774945192e+000
Dual infeasibility......:   4.2277292777725961e-013    4.2277292777725961e-013
Constraint violation....:   5.2846615972157451e-014    5.2846615972157451e-014
Complementarity.........:   2.5056918053500437e-009    2.5056918053500437e-009
Overall NLP error.......:   2.5056918053500437e-009    2.5056918053500437e-009


Number of objective function evaluations             = 6
Number of objective gradient evaluations             = 6
Number of equality constraint evaluations            = 6
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 6
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 5
Total CPU secs in IPOPT (w/o function evaluations)   =      0.358
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.

******************************************
 The problem solved in 5 iterations!
*** The final value of the objective function is -4.000000e+000.
```

Most of the content of the output is automatically generated by Ipopt. At the beginning, the user is informed against which solver the library was compiled (in the SimpleNLP example it was MUMPS linear solver). Then, initial information about the size of the problem ,passed from the class SimpleNLP, is displayed, e.g. number of non-zeros in the Jacobian or number of the constraints. Then, after initialisation, Ipopt starts finding the solution, and for every iteration it generates one line of output consisting of the following parameters:

- iter: The current iteration count
- objective: The current value of the objective function for the searched values of the optimisation x variables.
- inf_pr: The primal infeasibility at the current point.

- inf_du: The scaled dual infeasibility at the current point.
- lg(mu): Logarithm with base of 10 of the barrier parameter mu.
- ||d||: The infinity norm (max) of the primal step.
- lg(rg): Logarithm with base of 10 of the regularisation term for the Hessian of the Lagrangian in the augmented system.
- alpha_du: Step size for the dual variables.
- alpha_pr: Step size of the primal variables.
- ls: Number of backtracking line search steps.

After finishing iterating Ipopt passes to console window following parameters corresponding to the obtained solution:

- Objective
- Dual infeasibility
- Complementarity
- CPU secs in Ipopt
- CPU secs in NLP function evaluation

and returns with particular exit status (EXIT: Optimal Solution Found), which means that application left the solver. There are different values of the exit status, depending on the result of solving optimal problem, e.g. ”Solved To Acceptable Level” or ”Feasible Point Found”. Last two lines of the output are generated by our implementation of the main function.

Alternatively it would be also possible to implement finalize_solution method of the TNLP interface, which is called automatically by Ipopt when solution is found and receives information about the solution, so that user can output it, depending on the preferences. Below we present alternative output of the exemplary nlp problem with the finalize_solution method implemented in a more advanced way(since first part of the output is automatically generated by Ipopt we show only the bit produced by finalize_solution method and some printing routines in the main function after Ipopt returned with exit status):

```
EXIT: Optimal Solution Found.


Solution of the primal variables, x
x[0] = 1.000000e+000
x[1] = -1.937363e-008
```

```
Solution of the bound multipliers, z_L and z_U
z_L[0] = 1.252846e-009
z_L[1] = 0.000000e+000
z_U[0] = 8.000000e+000
z_U[1] = 0.000000e+000


Objective value
f(x*) = -4.000000e+000

Final value of the constraints:
g(0) = -5.284662e-014

******************************************
 The problem solved in 5 iterations!


*** The final value of the objective function is -4.000000e+000.
```

The implementation of the finalize_solution method is the following:

```
void SimpleNLP::finalize_solution(SolverReturn status, Index n,
                    const Number* x, const Number* z_L,
                    const Number* z_U,Index m,
                    const Number* g, const Number* lambda,
                    Number obj_value, const IpoptData* ip_data,
                    IpoptCalculatedQuantities* ip_cq)
{
  // the purpose of this method is to manage the solution,
  //e.g output it to the
  // console or write it to a text file

  // In this case solution is written to the console
  printf("\n\nSolution of the primal variables, x\n");
  for (Index i=0; i<n; i++)
  {
    printf("x[%d] = %e\n", i, x[i]);
  }

  printf
  ("\n\nSolution of the bound multipliers, z_L and z_U\n");
  for (Index i=0; i<n; i++)
  {
    printf("z_L[%d] = %e\n", i, z_L[i]);
  }

  for (Index i=0; i<n; i++)
  {
    printf("z_U[%d] = %e\n", i, z_U[i]);
  }

  printf("\n\nObjective value\n");
  printf("f(x*) = %e\n", obj_value);
  printf("\nFinal value of the constraints:\n");
```

```
   for (Index i=0; i<m ;i++)
   {
      printf("g(%d) = %e\n", i, g[i]);
   }
}
```

In finalize_solution method, we use n variable as upper limit in the for loops. This variable is passed back to the method to keep the track of the size of the problem. Then, in different for loops, we display by using printf function respectively: bound multipliers, values of primary variables x at the point x_*, optimised value of the objective function and finally values of the constraints at the point x_* The information in this chapter about how to implement simple nlp program in C++ language with Ipopt is based on [36].

CHAPTER 3

Optimal Control Problem

In the chapter 2 it is shown how solve and model a simple non-linear program using Ipopt. However, the final goal of this thesis is to optimise oil production in smart well system, which is a problem of a different and more complex type. Basically, oil problem is defined as optimal control and this chapter is devoted to this kind of problems.

In the very beginning, a brief introduction to optimal control problems will be given. Then, we will show a generic procedure of applying the simultaneous method in order to solve this kind of problem. Finally, we will present the discussed approach on a well-known example of non-linear Van Der Pol oscilator.

3.1 Optimal Control Problem

An optimal control problem is described by set of differential equations and objective function, which is also called a cost function. A typical feature of this kind of problems is that an objective function is a function of state and control variables whereas the mentioned set of differential equations is a mathematical model that defines the paths of the control variables to minimise the objective cost function and determines first order dynamic constraints on the variables. A standard continuous optimal control problem can be defined in the following way:

$$\min_{\{u(t),x(t)\}} \quad J = \int_{t_0}^{t_f} g(x(t), u(t))dt \tag{3.1.1a}$$

$$s.t. \quad x(t_0) = x_0 \tag{3.1.1b}$$

$$\dot{x} = f(x(t), u(t) \quad t \in [t_0, t_f] \tag{3.1.1c}$$

$$c(x(t), u(t)) \geq 0 \quad t \in [t_0, t_f] \tag{3.1.1d}$$

$$x_{\min} \leq x(t) \leq x_{\max} \tag{3.1.1e}$$

$$u_{\min} \leq u(t) \leq u_{\max} \tag{3.1.1f}$$

where:

1. (3.1.1a) represents the cost function.

2. (3.1.1b) is an initial condition.

3. (3.1.1c) determines mathematical model of the first order dynamic constraints.

4. (3.1.1d) determines algebraic path inequality constraints.

5. (3.1.1e), (3.1.1f) determine boundary conditions of state and control variables respectively.

6. x(t) is the state variable vector.

7. u(t) is the control variable vector.

8. t is an independent variable representing time and t_0 and t_f are initial and terminal times respectively.

Please note that some of the conditions to which cost function is subjected can simply be not active. For example, the path inequality constraints can be equal to zero. What is more, x_{min} and x_{max} can be equal to plus and minus infinity respectively, which means that there are no boundary conditions on state variables. The optimal control problem specified in such a way as above can have multiple solutions. In other words, the solution might not be unique, and in such a case, any solution (x(t),u(t),t) to the optimal control problem is so called local minimiser.

3.2 Numerical Methods to Solve Optimal Control Problem

In most of the cases optimal control problems are non-linear and can not be solved analytically like for example some special types of optimal control problems(e.q. linear-quadratic optimal control problems). Consequently, in order to solve these problems, one has to apply numerical methods. In general, one distinguishes between two types of them which are direct and indirect ones. When it comes to indirect methods they were developed in the 1950's with the start of usage of computers. Those methods are based mostly on calculus variation and cannot deal with dynamic path constraints. For the case investigated in this thesis, one of the direct methods will be applied since one of its features is that it translates the problem to the format which can be passed to Ipot (nlp format) for obtaining solution. Direct methods were introduced in the early 80's and have been developed since that time. Basically, in a direct method state, control variables and cost function are approximated by using one of function approximation techniques, for example piecewise constant parametrisation or polynomial approximation. Those procedures enable to discretise and translate the continuous time problem to the finite dimensional non-linear problem . The reason why those methods are so useful is because there already exist standard and robust algorithms for solving non-linear programs. Please also note that the size of translated nonlinear program depends to a significant extent on the type of the direct method for discretisation. It might occur that the nlp might be of size of tens of thousands of variables(e.g. in case of direct collocation, simultaneous method) but even though it is relatively easier to solve rather than boundary-value problem. This is because nonlinear programs can be specified as a sparse ones (containing many zero elements in the matrices which can be simply neglected), which speeds up calculation and saves a lot of computation resources such as memory space. In chapter 2 it was demonstrated that Ipopt asks only for non-zero elements of the matrices). Therefore it is very important that user defines his problem for the solver in a sparse format if it is possible of course.

Another important issue concerning translation of continuous time problem to discrete one is selection of the approximation method, which should provide conditions under which solutions to a series of increasingly accurate discretised problem converge to the solutions of original continuous one. For example, in case of using a variable step-size routine, it might be that the gradient does not converge to zero when integrating numerically dynamic constraints of the model.

3.3 Simultaneous Method as a Solution to an Optimal Control Problem

Simultaneous method is classified as one of the direct methods, and its main concept is that it fully discretises the optimal control problem. As a result, the newly transcibed nlp is very large because the method uses both controls and states as optimisation variables after reformulation. When it comes to model equations, they are not solved at each etarion, as it is done in case of sequential approaches, e.g. single shooting, but a simultaneous search for both model solution and optimal point is carried out. This reduces the number of necessary model simulations and can contribute to the lower computation time if the model evaluation is costly. Since simultaneous method directly couples model equations with the objective cost function and employs optimisation algorithm for finding simultaneously the solution to the model and minimizer to the objective cost function, it offers a full advantage of an open structure after transcription, such as direct access to first and second order derivatives, many degrees of freedomg and periodic boundary conditions.

3.3.1 Retriving Information from Continuous Time Problem

Given a continuous time optimal control problem of a Bolza's form in eq. 3.1.1, one can transcribe it to a discrete and numerically traceable one by employing different discretisation methods. Before doing that, however, we will introduce some important notations for retriving information from continuous time problem which will be used later in solving the dicrete non-linear program because of its periodic and structured boundary conditions. Assuming that the objective cost function g used in eq. (3.1.1a) is continuous and twicely differentiable, we introduce its gradient with respect to state and control variables of the following form:

$$\nabla g(x,u) = \begin{bmatrix} \frac{\partial g(x,u)}{\partial x} \\ \frac{\partial g(x,u)}{\partial u} \end{bmatrix} \tag{3.3.1}$$

and in particular:

$$\frac{\partial g(x,u)}{\partial x} = \begin{bmatrix} \frac{\partial g(x,u)}{\partial x_1} \\ \vdots \\ \frac{\partial g(x,u)}{\partial x_n} \end{bmatrix} \tag{3.3.2}$$

$$\frac{\partial g(x,u)}{\partial u} = \begin{bmatrix} \frac{\partial g(x,u)}{\partial u_1} \\ \vdots \\ \frac{\partial g(x,u)}{\partial u_m} \end{bmatrix} \tag{3.3.3}$$

where n and m define the sizes of the state and control variable vectors respectively. Next we introduce the Jacobians of the left hand side of the model f(x) with respect to states x:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{3.3.4}$$

and controls u:

$$\frac{\partial f}{\partial u} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \cdots & \frac{\partial f_2}{\partial u_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_m} \end{bmatrix} \tag{3.3.5}$$

The Hessian H(g(x,u)), which is a square matrix of partial second order derivatives of the continuous g function with respect to states and controls, is defined as following:

$$H(g(x,u)) = \nabla^2 g(x,u) \tag{3.3.6}$$

$$\nabla^2 g(x,u) = \begin{bmatrix} \frac{\partial g}{\partial x_1 \partial x_1} & \frac{\partial g}{\partial x_1 \partial x_2} & \cdots & \frac{\partial g}{\partial x_1 \partial x_n} & \frac{\partial g}{\partial x_1 \partial u_1} & \frac{\partial g}{\partial x_1 \partial u_2} & \cdots & \frac{\partial g}{\partial x_1 \partial u_m} \\ \frac{\partial g}{\partial x_2 \partial x_1} & \frac{\partial g}{\partial x_2 \partial x_2} & \cdots & \frac{\partial g}{\partial x_2 \partial x_n} & \frac{\partial g}{\partial x_2 \partial u_1} & \frac{\partial g}{\partial x_2 \partial u_2} & \cdots & \frac{\partial g}{\partial x_2 \partial u_m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g}{\partial x_n \partial x_1} & \frac{\partial g}{\partial x_n \partial x_2} & \cdots & \frac{\partial g}{\partial x_n \partial x_n} & \frac{\partial g}{\partial x_n \partial u_1} & \frac{\partial g}{\partial x_n \partial u_2} & \cdots & \frac{\partial g}{\partial x_n \partial u_m} \\ \frac{\partial g}{\partial x_1 \partial u_1} & \frac{\partial g}{\partial u_1 \partial x_2} & \cdots & \frac{\partial g}{\partial u_1 \partial x_n} & \frac{\partial g}{\partial u_1 \partial u_1} & \frac{\partial g}{\partial u_1 \partial u_2} & \cdots & \frac{\partial g}{\partial u_1 \partial u_m} \\ \frac{\partial g}{\partial u_2 \partial x_1} & \frac{\partial g}{\partial u_2 \partial x_2} & \cdots & \frac{\partial g}{\partial u_2 \partial x_n} & \frac{\partial g}{\partial u_2 \partial u_1} & \frac{\partial g}{\partial u_2 \partial u_2} & \cdots & \frac{\partial g}{\partial u_2 \partial u_m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g}{\partial u_m \partial x_1} & \frac{\partial g}{\partial u_m \partial x_2} & \cdots & \frac{\partial g}{\partial u_m \partial x_n} & \frac{\partial g}{\partial u_m \partial u_1} & \frac{\partial g}{\partial u_m \partial u_2} & \cdots & \frac{\partial g}{\partial u_m \partial u_m} \end{bmatrix} \tag{3.3.7}$$

We have already shown in chapter 2 that in order to robustly solve the nonlinear program with the use of Ipopt, one needs to pass the Hessian of the Lagrangian to the solver. This matrix is constructed out of two terms, where one is the Hessian of the objective cost function and the other is the Hessian of the constraints. The constraint term is obtained by summing the Hessians of every single constraint multiplied by the correspondig lambda multiplier. The hessian of the single constraint having index i in the f vector function is defined the following:

$$\nabla^2 f_i(x,u) = \begin{bmatrix} \frac{\partial f_j}{\partial x_1 \partial x_1} & \frac{\partial f_j}{\partial x_1 \partial x_2} & \cdots & \frac{\partial f_i}{\partial x_1 \partial x_n} & \frac{\partial f_i}{\partial x_1 \partial u_1} & \frac{\partial f_i}{\partial x_1 \partial u_2} & \cdots & \frac{\partial f_i}{\partial x_1 \partial u_m} \\ \frac{\partial f_i}{\partial x_2 \partial x_1} & \frac{\partial f_i}{\partial x_2 \partial x_2} & \cdots & \frac{\partial f_i}{\partial x_2 \partial x_n} & \frac{\partial f_i}{\partial x_2 \partial u_1} & \frac{\partial f_i}{\partial x_2 \partial u_2} & \cdots & \frac{\partial f_i}{\partial x_2 \partial u_m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_i}{\partial x_n \partial x_1} & \frac{\partial f_i}{\partial x_n \partial x_2} & \cdots & \frac{\partial f_i}{\partial x_n \partial x_n} & \frac{\partial g}{\partial x_n \partial u_1} & \frac{\partial f_i}{\partial x_n \partial u_2} & \cdots & \frac{\partial f_i}{\partial x_n \partial u_m} \\ \frac{\partial f_i}{\partial x_1 \partial u_1} & \frac{\partial f_i}{\partial u_1 \partial x_2} & \cdots & \frac{\partial g}{\partial u_1 \partial x_n} & \frac{\partial f_i}{\partial u_1 \partial u_1} & \frac{\partial f_i}{\partial u_1 \partial u_2} & \cdots & \frac{\partial f_i}{\partial u_1 \partial u_m} \\ \frac{\partial f_i}{\partial u_2 \partial x_1} & \frac{\partial f_i}{\partial u_2 \partial x_2} & \cdots & \frac{\partial f_i}{\partial u_2 \partial x_n} & \frac{\partial f_i}{\partial u_2 \partial u_1} & \frac{\partial f_i}{\partial u_2 \partial u_2} & \cdots & \frac{\partial f_i}{\partial u_2 \partial u_m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_i}{\partial u_m \partial x_1} & \frac{\partial f_i}{\partial u_m \partial x_2} & \cdots & \frac{\partial f_i}{\partial u_m \partial x_n} & \frac{\partial f_i}{\partial u_m \partial u_1} & \frac{\partial f_i}{\partial u_m \partial u_2} & \cdots & \frac{\partial f_i}{\partial u_m \partial u_m} \end{bmatrix} \tag{3.3.8}$$

3.3.2 Translating a Continuous Time Problem into a Discrete Nonlinear Program (Temporal Discretisation)

In order to translate the continuous time optimal control problem into a numerically traceable discrete one, we divide the temporal domain $[t_0, t_f]$ into N equally distributed control steps for the integration of the differential model equations. Every control step will be coupled with a set of corresponding state and control variables. Since the simultaneous method treats both state and control variables as optimisation variables, we introduce a new variable vector z which is obtained by mapping x and u variables on their time steps.

$$z = \begin{bmatrix} x_0 \\ u_0 \\ x_1 \\ u_1 \\ \vdots \\ x_{N-1} \\ u_{N-1} \\ x_N \end{bmatrix} \tag{3.3.9}$$

The discretisation the differential model equations is done by empoying a first order implicit Runge Kuta method, known also as Euler method. This approach is comonly used for the numerical integration of ordinary differential equations and implements the following approximation:

$$\frac{x_t - x_{t-h}}{h} = f(x_t) \tag{3.3.10}$$

where:

1. h is the value of the time step.

2. f is a right hand side of a integrated differential equation.

by multiplying both sides by the time step h, one can get the formula of the Euler method:

$$x_t = hf(x_t) + x_{t-h} \tag{3.3.11}$$

The reason why this method is called an implicit one is because solution x_t is given as an implicit function of x_t and one has to solve an equation to find x_t.

By applying backward Euler method to mathematical model we discretised the dynamic first order constraints into residual algebraic ones. The set of residual constraints corresponding the single time step can be defined by R_k, where k denotes the given time step, which yields:

$$R_{k+1}(x_{k+1}, x_k, u_k) = x_{k+1} - hf(x_{k+1}, u_k) - x_k = 0 \;\; for\; k = 0, 1, ...N - 1 \tag{3.3.12}$$

and the initial condition constraint is:

$$R_0(x_0) = x_0 - a = 0 \tag{3.3.13}$$

where: a is an initial condition parameter and x_k,u_k are new vectors of parameters obtained from approximation for the k^{th} time step. These parameters are going to be treated as optimisation variables in the transcribed non-linear program.
Quite similar procedure is performed on the objective cost function which enables to represent the integral in the objective cost function as a sum of small portions in which the g function is assumed to be constant. The only difference is that we use an explicit Euler method, known also as forward method. Forward Euler method is one of the simplest explicit Runge-Kutta methods and is given by the formula below:

$$y_{(t+h)} = y_t + hf(t, y_t) \tag{3.3.14}$$

The reason why this method is called an explicit one (contrary to the backward Euler method, being described above), is because the new searched value is defined by terms that are already known, e.g. y_t. After applying forward Euler method to the objective function, it is possible to represent it in the following form:

$$J = \sum_{k=0}^{N-1} hg(x_k, u_k) \tag{3.3.15}$$

Now it is possible to represent the continuouos time optimal control problem defined in eq. 3.1.1 as a discrete-time non-linear program of the following form:

$$\min_{\{u_k,x_{k+1}\}} \quad J = \sum_{k=0}^{N-1} g(x_{k,}u_k) \tag{3.3.16a}$$

$$s.t. \quad R_0(x_0) = 0 \tag{3.3.16b}$$

$$R_{k+1}(x_{k+1}, x_k, u_k) = 0 \quad k = 0, 1, ., N-1 \tag{3.3.16c}$$

$$u_{\min} \leq u_k \leq u_{\max} \qquad k = 0, 1, ..., N-1 \tag{3.3.16d}$$

$$x_{\min} \leq x_k \leq x_{\max} \qquad k = 0, 1, ..., N \tag{3.3.16e}$$

Substituting (3.3.13) into (3.3.16b) and (3.3.12) into (3.3.16c), it is possible formulate the discrete problem in the iterative way:

$$\min_{\{u_k,x_{k+1}\}} \quad J = \sum_{k=0}^{N-1} hg(x_k, u_k) \tag{3.3.17a}$$

$$s.t. \quad x_0 = a \tag{3.3.17b}$$

$$x_{k+1} = hf(x_{k+1}, u_k, t_{k+1}) + x_k \quad k = 0, 1, ..., N-1 \tag{3.3.17c}$$

$$u_{\min} \leq u_k \leq u_{\max} \qquad k = 0, 1, ..., N-1 \tag{3.3.17d}$$

$$x_{\min} \leq x_k \leq x_{\max} \qquad k = 0, 1, ..., N \tag{3.3.17e}$$

$$\tag{3.3.17f}$$

3.3.3 First and Second Order Derivatives of the Discrete NLP

One can think at the first glance that modelling of the discretised non-linear program can be difficult and very time consuming, when it comes to the size of the problem with respect to number of variables and algebraic constraints. In general, non-linear program obtained from direct collocation method have the tendency to be very large since for every time step approximation the problem is enriched in new set of x and u variables. Consequently, process of representation and implementation of this kind of nlps should be automated, so that the code itself generates all information of the discretised problem, whereas the user should only specify parameters strictly specific for the given continuous problem. This approach will definitely speed up the problem representation process and what is more, can be implemented and reused for any other discretisation of continuous optimal control problem.

In this section we will show this kind of modular approach for defining first and second order derivatives of the discrete nlp. This is very crucial as the solver requires a problem of a nlp format and information which corresponds to it. Consequently, we will show how to use and call already defined information in the section (3.3.1) for the continuous time problem in order to construct required information of the discrete problem such as gradient of the discrete cost function, jacobian of the residaul constraints and hessian of the lagrangian.

The gradient of the discrete objective cost function J with respect to z is the following

$$\frac{\partial J(z)}{\partial z} = \begin{bmatrix} \frac{\partial J(z)}{\partial x_0} \\ \frac{\partial J(z)}{\partial u_0} \\ \frac{\partial J(z)}{\partial x_1} \\ \frac{\partial J(z)}{\partial u_1} \\ \vdots \\ \frac{\partial J(z)}{\partial x_{N-1}} \\ \frac{\partial J(z)}{\partial u_{N-1}} \\ \frac{\partial J(z)}{\partial x_N} \end{bmatrix} = \begin{bmatrix} h\frac{\partial g(x_0,u_0)}{\partial x_0} \\ h\frac{\partial g(x_0,u_0)}{\partial u_0} \\ h\frac{\partial g(x_1,u_1)}{\partial x_1} \\ h\frac{\partial g(x_1,u_1)}{\partial u_1} \\ \vdots \\ h\frac{\partial g(x_{N-1},u_{N-1})}{\partial x_{N-1}} \\ h\frac{\partial g(x_{N-1},u_{N-1})}{\partial u_{N-1}} \\ 0 \end{bmatrix} \tag{3.3.18}$$

In order to determine the jacobian of the residual constraints we introduce the vector function R(z) which stores all residual algebraic constraints with respect to z.

$$R(z) = \begin{bmatrix} R_0(x_0) \\ R_1(x_1, x_0, u_0) \\ R_2(x_2, x_1, u_1) \\ \vdots \\ R_N(x_N, x_{N-1}, u_{N-1}) \end{bmatrix} = \begin{bmatrix} x_0 - a \\ x_1 - hf(x_1, u_0) - x_0 \\ x_2 - hf(x_2, u_1) - x_1 \\ \vdots \\ x_N - hf(x_N, u_{N-1}) - x_N \end{bmatrix} \tag{3.3.19}$$

Then the Jacobian of the redisudal constraints will look the following:

$$\frac{\partial R}{\partial z} = \begin{bmatrix} \frac{\partial R_0}{\partial x_0} & \frac{\partial R_0}{\partial u_0} & \frac{\partial R_0}{\partial x_1} & \frac{\partial R_0}{\partial u_1} & \cdots & \frac{\partial R_0}{\partial x_{N-1}} & \frac{\partial R_0}{\partial u_{N-1}} & \frac{\partial R_0}{\partial x_N} \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial x_{N-1}} & \frac{\partial R_1}{\partial u_{N-1}} & \frac{\partial R_1}{\partial x_N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \frac{\partial R_N}{\partial x_0} & \frac{\partial R_N}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_N}{\partial u_1} & \cdots & \frac{\partial R_N}{\partial x_{N-1}} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N} \end{bmatrix} \tag{3.3.20}$$

By looking at the structure of the Jacobian of the constraints one can quickly picture himself its sparsity pattern, e.g. only the first element of the first row of the jacobian matrix will be non-zero as initial condition residual constraint is a function of x_0 only. Sparsity is a typical feature of nonlinear problems obtained by discretising a continous time optimal control problems. Now eq. (3.4.25) can be rewritten, distinguishing between zero and non zero elements.

$$\frac{\partial R}{\partial z} = \begin{bmatrix} \frac{\partial R_0}{\partial x_0} & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{\partial R_N}{\partial x_{N-1}} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N} \end{bmatrix} \tag{3.3.21}$$

Where the non-zero entrances can be computed by calling f and g function of the continuous problem with the corresponding x and u optimisation parameters.

$$\frac{\partial R_0}{\partial x_0} = I \tag{3.3.22a}$$

$$\frac{\partial R_1}{\partial x_0} = -I \tag{3.3.22b}$$

$$\frac{\partial R_1}{\partial u_0} = -h\frac{\partial f}{\partial u}(x_1, u_0) \tag{3.3.22c}$$

$$\frac{\partial R_1}{\partial x_1} = I - h\frac{\partial f}{\partial x}(x_1, u_0) \tag{3.3.22d}$$

$$\frac{\partial R_N}{\partial x_{N-1}} = -I \tag{3.3.22e}$$

$$\frac{\partial R_N}{\partial u_{N-1}} = -h\frac{\partial f}{\partial u}(x_N, u_{N-1}) \tag{3.3.22f}$$

$$\frac{\partial R_N}{\partial x_{N-1}} = I - h\frac{\partial f}{\partial x}(x_N, u_{N-1}) \tag{3.3.22g}$$

The Lagrangian of the discretised nonlinear program is defined as

$$\phi(z) + R(z)^T\lambda \tag{3.3.23}$$

and the Hessian of the Lagrangian in symbolic form is:

$$\nabla^2 I(z) + \sum_{i=0}^{N}\sum_{j=1}^{n}\lambda_{2k+j}\nabla^2 R_{j_i}(z) \tag{3.3.24}$$

where i is a time step iterator, j is a signle residual constraint iterator and R_{j_i} is a single residual constraint with index j at time step i.
The first term in eq. (3.3.24) comes from the Hessian of the discretised objective function and the second sum term comes from the Hessian of the constraints. Please note that in order to follow Ipopt format, contributions of Hessian matrices coming from every single algebraic constraint for every time step are individually multiplied by corresponding lambda multiplier and then summed up. The first term of the Hessian of Lagrangian coming form the objective cost function is defined as following:

$$\nabla^2 I(z) = \begin{bmatrix} h\nabla^2 g(x_0, u_0) & 0 & \cdots & 0 & 0 \\ 0 & h\nabla^2 g(x_1, u_1) & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & h\nabla^2 g(x_{N-1}, u_{N-1}) & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \tag{3.3.25}$$

The hessian of the single algebraic constraint with index j for the time step i - R_{j_i}, can be computed by calling the hessian of the f_j function with the set of x and u optimisation parameters coupled with time step i:

$$\nabla^2 R_{j_i}(z) = \begin{bmatrix} 0 & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & h\nabla^2 f_j(x_i, u_i) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \end{bmatrix} \tag{3.3.26}$$

3.4 Exemplary Optimal Control Problem of Van Der Pol Oscillator

So far in this chapter we explained the concept of a simultaneous method and have shown the generic approach of applying it to solve the continuous time optimal control problem. This section will complement the discussed theory by applying the already described approach on a particular example of optimal control called Van Der Pol Oscillator.

3.4.1 Problem Representation

A Van Der Pol oscillator is a non-conservative oscillator with non-linear damping, which evolves in time according to the second order differential equation but can be also transcribed into two-dimensional form of first order differential equations. The problem is specified in the following Bolza form:

$$\min_{\{u(t), x(t)\}} \quad J = \int\limits_{t_0}^{t_f} g(x, u)dt \tag{3.4.1a}$$

$$s.t. \qquad x(t_0) = a \tag{3.4.1b}$$

$$\dot{x} = f(t, x, u) \tag{3.4.1c}$$

$$x_{\min} < x < x_{\max} \tag{3.4.1d}$$

$$u_{\min} < u < u_{\max} \tag{3.4.1e}$$

where:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{3.4.2a}$$

$$g(x, u) = x_1^2 + x_2^2 + u^2 \tag{3.4.2b}$$

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ (1 - x_1^2)x_2 - x_1 + u \end{bmatrix} = \begin{bmatrix} f_1(x,u) \\ f_2(x,u) \end{bmatrix} = f(x,u) \tag{3.4.3}$$

which makes the Van Der Pol problem look the following:

$$\min_{\{u(t),x(t)\}} \quad J = \int_{t_0}^{t_f} x_1^2 + x_2^2 + u^2 dt \tag{3.4.4a}$$

$$s.t. \quad x_1(t_0) = 1 \tag{3.4.4b}$$

$$x_2(t_0) = 0 \tag{3.4.4c}$$

$$\dot{x}_1 = x_2 \tag{3.4.4d}$$

$$\dot{x}_2 = (1 - x_1^2)x_2 - x_1 + u \tag{3.4.4e}$$

$$u < u_{\max} \tag{3.4.4f}$$

Please note, not all the constraint are active for this optimal control problem. For example there are no boundary condition on state variables, as well as algebraic inequality constraints are equal to zero.The gradient of the continuous objective cost function ∇ g(x,u) is the following:

$$\nabla g(x,u) = \begin{bmatrix} \frac{\partial g(x,u)}{\partial x} \\ \frac{\partial g(x,u)}{\partial u} \end{bmatrix} \tag{3.4.5}$$

where:

$$\frac{\partial g(x,u)}{\partial x} = \begin{bmatrix} \frac{\partial g(x,u)}{\partial x_1} \\ \frac{\partial g(x,u)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix} \tag{3.4.6}$$

$$\frac{\partial g(x,u)}{\partial u} = 2u \tag{3.4.7}$$

The jacobians of the left hand side function f of the model with respect to x and u variables are the following:

$$\frac{\partial f(x,u)}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2x_1x_2 - 1 & 1 - x_1^2 \end{bmatrix} \tag{3.4.8}$$

$$\frac{\partial f(x,u)}{\partial u} = \begin{bmatrix} \frac{\partial f_1}{\partial u} \\ \frac{\partial f_2}{\partial u} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.4.9}$$

The Hessian of the continuous g function H(g(x,u) for this problem becomes:

$$H(g(x,u) = \nabla^2 g(x,u) \tag{3.4.10}$$

$$\nabla^2 g(x,u) = \begin{bmatrix} \frac{\partial g}{\partial x_1 \partial x_1} & \frac{\partial g}{\partial x_1 \partial x_2} & \frac{\partial g}{\partial x_1 \partial u} \\ \frac{\partial g}{\partial x_2 \partial x_2} & \frac{\partial g}{\partial x_2 \partial x_1} & \frac{\partial g}{\partial x_2 \partial u} \\ \frac{\partial g}{\partial u \partial x_1} & \frac{\partial g}{\partial u \partial x_2} & \frac{\partial g}{\partial u \partial u} \end{bmatrix} \tag{3.4.11}$$

and after calculating partial derivatives one can get

$$\nabla^2 g(x,u) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \tag{3.4.12}$$

Since f is two-dimensional vector function, we have two terms for the Hessian of Lagrangian coming from the constraints, which are the following

$$\nabla^2 f_1(x,u) = 0 \tag{3.4.13}$$

$$\nabla^2 f_2(x,u) = \begin{bmatrix} -2x_2 & -2x_1 & 0 \\ -2x_1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.4.14}$$

3.4.2 Temporal Discretisation of Van Der Pol Problem

After retrieving all the necessary information from the continuous problem, we are going to discretise it by using implicit and explicit first order methods described in section 3.3.2. The Van Der Pol mathematical model is solved numerically by using Implicit Euler method which transcibres it to the following residual algebrac form:

$$R_{k+1}(x_{k+1}, x_k, u_k) = x_{k+1} - hf(x_{k+1}, u_k) - x_k = 0 \;\; for\; k = 0, 1, ...N-1 \tag{3.4.15}$$

Since the model consists of 2 dynamic constraints on x_1 and x_2 state variables, every constraint R_k is a two dimensional vector function which can be resolved into the following matrix equation by substituting eq. (3.4.3) into eq. (3.4.15) which yields:

$$R_{k+1}(x_{k+1}, x_k, u_k) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_k - h \begin{bmatrix} x_2 \\ (1-x_1^2)x_2 - x_1 + u \end{bmatrix}_k - \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{k-1} = 0 \tag{3.4.16}$$

The initial condition on the states imposed by the constraint R_0 is the following:

$$R_0(x_0) = x_0 - a = 0 \tag{3.4.17}$$

where

$$a = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3.4.18}$$

and yields:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3.4.19}$$

The objective cost function is discretised by explicit euler method in exactly the same manner as it was shown in section (3.3.2). Consequently, it is possible to represent Van Der Pol oscilaltor optimal control problem as discrete non-linear program:

$$
\begin{aligned}
&\min_{\{u_k,x_{k+1}\}} && J = \sum_{k=0}^{N-1} g(x_{k,}u_k) && (3.4.20a)\\
&s.t. && R_0(x_0) = 0 && (3.4.20b)\\
& && R_{k+1}(x_{k+1}, x_k, u_k) = 0 \quad k = 0, 1, ., N-1 && (3.4.20c)\\
& && u_k \leq u_{\max} \qquad\qquad\quad k = 0, 1, ..., N-1 && (3.4.20d)
\end{aligned}
$$

3.4.3 First and Second Order Derivatives of Van Der Pol Problem

In this section we will apply the general procedure of defining first and second order derivatices for the discrete nlp on the particular example of Van Der Pol problem. In order to do that, we set the number of time step N equal to 3 which makes the problem look the following:

$$
\begin{aligned}
&\min_{\{u_k,x_{k+1}\}} && J = \sum_{k=0}^{2} g(x_{k,}u_k) && (3.4.21a)\\
& && R_0 = x_0 - a = 0 && (3.4.21b)\\
& && R_1(x_1, x_0, u_0) = \; x_1 - hf(x_1, u_0, t_1) - x_0 = 0 && (3.4.21c)\\
& && R_2(x_2, x_1, u_1) = \; x_2 - hf(x_2, u_1, t_2) - x_1 = 0 && (3.4.21d)\\
& && R_3(x_3, x_2, u_2) = x_3 - hf(x_3, u_2, t_3) - x_2 = 0 && (3.4.21e)\\
& && u_0 \leq u_{\max} && (3.4.21f)\\
& && u_1 \leq u_{\max} && (3.4.21g)\\
& && u_2 \leq u_{\max} && (3.4.21h)
\end{aligned}
$$

And the z vector introduced in eq. (3.3.9) for this particular problem becomes:

$$
z = \begin{bmatrix} x_0 \\ u_0 \\ x_1 \\ u_1 \\ x_2 \\ u_2 \\ x_3 \end{bmatrix} \tag{3.4.22}
$$

The gradient of the discrete cost function with respect to z variable vector is the following:

$$\frac{\partial J(z)}{\partial z} = \begin{bmatrix} \frac{\partial J(z)}{\partial x_0} \\ \frac{\partial J(z)}{\partial u_0} \\ \frac{\partial J(z)}{\partial x_1} \\ \frac{\partial J(z)}{\partial u_1} \\ \frac{\partial J(z)}{\partial x_2} \\ \frac{\partial J(z)}{\partial u_2} \\ \frac{\partial J(z)}{\partial x_3} \end{bmatrix} = \begin{bmatrix} h\frac{\partial g(x_0,u_0)}{\partial x_0} \\ h\frac{\partial g(x_0,u_0)}{\partial u_0} \\ h\frac{\partial g(x_1,u_1)}{\partial x_1} \\ h\frac{\partial g(x_1,u_1)}{\partial u_1} \\ h\frac{\partial g(x_2,u_2)}{\partial x_2} \\ h\frac{\partial g(x_2,u_2)}{\partial u_2} \\ 0 \end{bmatrix} \tag{3.4.23}$$

The reason why, partial derivative with respect to x_3 is zero, is because this set of variables does not appear in the objective cost function as portions of g function were summed from k = 0 , to k = 2. Next we focus on the derivatives of the constraints. The term R(z) introduced in eq. (3.4.24) for this instance of the problem is the becomes:

$$R(z) = \begin{bmatrix} R_0(x_0) \\ R_1(x_1, x_0, u_0) \\ R_2(x_2, x_1, u_1) \\ R_3(x_3, x_2, u_2) \end{bmatrix} = \begin{bmatrix} x_0 - a \\ x_1 - hf(x_1, u_0) - x_0 \\ x_2 - hf(x_2, u_1) - x_1 \\ x_3 - hf(x_3, u_2) - x_2 \end{bmatrix} \tag{3.4.24}$$

and consequently the jacobian $\frac{\partial R}{\partial z}$ is the following:

$$\frac{\partial R}{\partial z} = \begin{bmatrix} \frac{\partial R_0}{\partial x_0} & \frac{\partial R_0}{\partial u_0} & \frac{\partial R_0}{\partial x_1} & \frac{\partial R_0}{\partial u_1} & \frac{\partial R_0}{\partial x_2} & \frac{\partial R_0}{\partial u_2} & \frac{\partial R_0}{\partial x_3} \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial u_1} & \frac{\partial R_1}{\partial x_2} & \frac{\partial R_1}{\partial u_2} & \frac{\partial R_1}{\partial x_3} \\ \frac{\partial R_2}{\partial x_0} & \frac{\partial R_2}{\partial u_0} & \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial u_1} & \frac{\partial R_2}{\partial x_2} & \frac{\partial R_2}{\partial u_2} & \frac{\partial R_2}{\partial x_3} \\ \frac{\partial R_3}{\partial x_0} & \frac{\partial R_3}{\partial u_0} & \frac{\partial R_3}{\partial x_1} & \frac{\partial R_3}{\partial u_1} & \frac{\partial R_3}{\partial x_2} & \frac{\partial R_3}{\partial u_2} & \frac{\partial R_3}{\partial x_3} \end{bmatrix} \tag{3.4.25}$$

Next, equation (3.4.25) can be rewritten, distinguishing between zero and non zero elements.

$$\frac{\partial R}{\partial z} = \begin{bmatrix} \frac{\partial R_0}{\partial x_0} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial u_1} & \frac{\partial R_1}{\partial x_2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial R_3}{\partial x_2} & \frac{\partial R_3}{\partial u_2} & \frac{\partial R_3}{\partial x_3} \end{bmatrix} \tag{3.4.26}$$

The Hessian of the Lagrangian in symbolic form for the problem examined is:

$$\nabla^2 J(z) + \sum_{i=0}^{3} \sum_{j=1}^{2} \lambda_{2k+j} \nabla^2 R_{j_i}(z) \tag{3.4.27}$$

which becomes:

$$\nabla^2 J(z) + \lambda_4 \nabla^2 R_{2_1} + \lambda_6 \nabla^2 R_{2_2} + \lambda_8 \nabla^2 R_{2_3} \tag{3.4.28}$$

The first term of the Hessian of Lagrangian coming form the objective cost function is the following:

$$\nabla^2\phi(z) = \begin{bmatrix} h\nabla^2 g(x_0,u_0) & 0 & 0 & 0 \\ 0 & \nabla^2 g(x_1,u_1) & 0 & 0 \\ 0 & 0 & \nabla^2 g(x_2,u_2) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.4.29}$$

and the non-zero contributions of the Hessian of the constraints are:

$$\nabla^2 R_{2_1}(z) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -h\nabla^2 f_2(x_1,u_0) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.4.30}$$

$$\nabla^2 R_{2_2}(z) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -h\nabla^2 f_2(x_2,u_1) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.4.31}$$

$$\nabla^2 R_{2_3}(z) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -h\nabla^2 f_2(x_3,u_2) \end{bmatrix} \tag{3.4.32}$$

After retrieving all necessary information from our non-linear program, it is possible to move to the next step which is implementing the problem in C++ and passing it to Ipopt.

3.5 Van Der Pol Oscilator Ipopt Interface

As it was mentioned before, in order to model discrete nonlinear problem in Ipopt, a modular approach should be taken distinguishing between 2 different steps when representing the problem. First step is to retrieve necessary information from continuous problem such as: dynamic constraints, bounds on state and control variables, Jacobian of the dynamic constraints and Hessian of the Lagrangian. Whereas, the second step is to create a discrete non-linear problem based on this information. It can also be seen that the first step is very problem-specific, while the second step is generic. As mentioned in chapter 2, in order to model optimisation problem in Ipopt, one needs to implement the generic interface by inheriting from TNLP class and override interface methods in the child class, so that they return information about the specific optimisation problem. In order to be consistent with Ipopt structure, we will create an additional class layer. This class (DiscreteNLP) will be responsible for discretisation (step 2) of

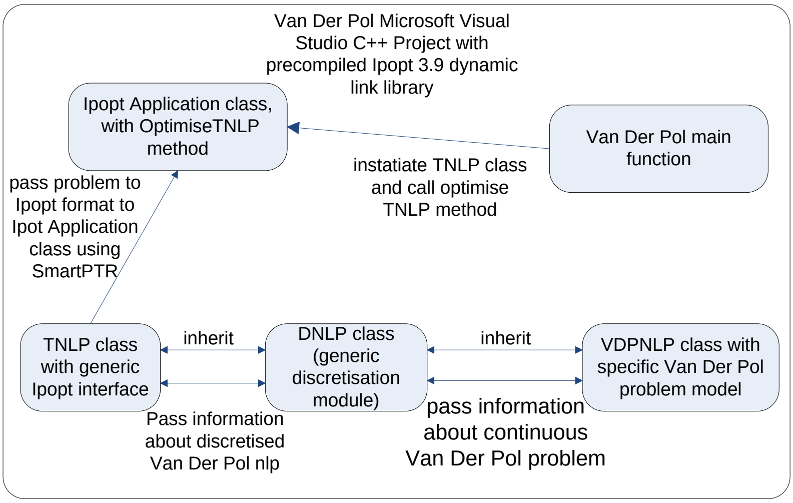

Van Der Pol Microsoft Visual Studio C++ Project with precompiled Ipopt 3.9 dynamic link library
Ipopt Application class, with OptimiseTNLP method
Van Der Pol main function
instatiate TNLP class and call optimise TNLP method
pass problem to Ipopt format to Ipot Application class using SmartPTR
TNLP class with generic Ipopt interface
inherit
DNLP class (generic discretisation module)
inherit
VDPNLP class with specific Van Der Pol problem model
Pass information about discretised Van Der Pol nlp
pass information about continuous Van Der Pol problem

Figure 3.1: Van Der Pol NLP class structure

the continuous time problem. The class will have its own interface that should be implemented in the object of the child class related to the specific problem (step 1). Obviously, all the classes should inherit from TNLP class since, only by overriding methods in this class it is possible to pass the problem to Ipopt. It is also possible to completely hard-code the problem directly by inheriting from TNLP class, as it was done in the chapter 2. This approach however, would not be very efficient since nlps obtained by direct collocation methods are very large and as a result it would be time consuming to model this problem for a large horizon. What is more, thanks to separation between discrete and initial problem it is possible to set up appropriate horizon, and change the value or number of time steps without any ease and modifications of the initial problem. The class structure of our approach is presented in the figure 3.1. An extra attention should be paid to the VDPNLP(Van Der Pol Non-Linear Program) class which functionality is to represent the Van Der Pol problem. This class inherits from abstract DNLP class and consequently, has to implement all the pure virtual methods declared in that class. By implementing these methods in the VDPNLP class, one can retrieve information about the Van Der Pol continuous problem. The declaration of the virtual methods in the VDPNLP class header file is the following

```
//virtual methods region
virtual void get_nlp_info(Index& N, Index& m, Index& d_x,
                          Index& d_u, Index& nnz_jac_g,
                          Index& nnz_h_lag_init,
                          Index& nnz_h_lag,
                                   Index& nnz_h_lag_fin,
                                   TNLP::IndexStyleEnum& index_style)

virtual void get_bounds_info(Number* &x_l, Number* &x_u,
                             Number* &u_l,  Number* u_u,
                             Number* &g_l, Number* &g_u)

virtual Number eval_f_d(Number* x, Number* u);

virtual bool eval_grad_f_x(Number* x, Number* u,
                           Number* & grad_f_x);

virtual bool eval_grad_f_u(Number* x, Number* u,
                           Number* & grad_f_u);

virtual bool eval_g_init_d(Number* x, Number* u,
                           Number*& g);

virtual bool eval_g_d(Number* x, Number* u,
                      Number*& g);

virtual bool eval_jac_g_values_x(Number*& values, bool*& flags,
                                 Number* x, Number* u);

virtual bool eval_jac_g_values_u(Number*& values, bool*& flags,
                                 Number* x, Number* u);

virtual bool eval_jac_g_s(Int*& irow, Int*& jcol);

virtual bool eval_h_s(Int*& irow_i, Int*& jcol_i, Int*& i_row,
                      Int*& j_col, Int*& irow_f, Int*& j_col_f);

virtual bool eval_h_v(Number* &valules, Number* x, Number* u,
                      const Number* lambda, Number obj_factor,
                      Int t_s, Int k, Int x_d, Number h);
```

The short description of the functionality and purpose of each virtual method is given below, so it is clear how to implement them in the child class. Please remember that the virtual methods defined in the DNLP class are generic and can be used to represent any continuous time problem that should be translated into discrete nlp by creating an instance of a class derived from DNLP class. Almost all the methods resemble the virtual ones defined in TNLP class when it comes to the format. They might require some additional parameters storing information for discretiation (e.g. number of time steps, value of the current time step or the step size). Consequently, if the user is able to code the problem in an Ipopt format it should not be difficult for him to implement the new methods.

Method virtual void get_nlp_info with prototype:

```
get_nlp_info(Index& N, Index& m, Index& x_d, Index& u_d,
             Index& nnz_jac_g,Index& nnz_h_lag_init,
             Index& nnz_h_lag, Index& nnz_h_lag_fin,
             IndexStyleEnum& index_style)
```

Passes to Ipopt information about the size of continuous and discrete problem

- N - number of discretisation time steps
- m - number of the constraints in the mathematical model
- x_d - the dimension of the x variable vector
- u_d - the dimension of the u variable vector
- nnz_jac_g - number of non zeros of Jacobian of the constraints with respect to x and u variable defined in equation (3.4.6)
- nnz_h_lag_init - number of non zeros of the Hessian of the Lagrangian for initial condition (k = 0)
- nnz_h_lag - number of non zeros of the Hessian of the Lagrangian for regular condition (0 < k < N) - $\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$
- nnz_h_lag_fin - number of non zeros of the Hessian of the Lagrangian for final condition (k = N)
- index_style - the way of the numbering array elements when specifying sparsity structures, it is possible to set it to either C_STYLE- starting from 0, or FORTRAN_STYLE starting from 1.

```
void VDPNLP :: get_nlp_info(Index& N, Index& m, Index& d_x,
                            Index& d_u, Index& nnz_jac_g,
                            Index& nnz_h_lag_init,
                            Index& nnz_h_lag,
                            Index& nnz_h_lag_fin,
                            TNLP::IndexStyleEnum& index_style)

{
  N = 20;
  m = 2;
  d_x = 2;
  d_u = 2;
  nnz_jac_g = 7;
  nnz_h_lag_init = 3;
  nnz_h_lag = 4;
  nnz_h_lag_fin = 2;
  index_style = TNLP::C_STYLE;
}
```

Method get_bounds_info with a prototype:

```
void get_bounds_info(Number* &x_l, Number* &x_u, Number* &u_l,
                     Number* &u_u, Number* &g_l, Number* &g_u);
```

- x_l - lower bounds on x variable
- x_u - upper bounds on x variable
- u_l - lower bounds on u variable
- u_u - upper bounds on u variable
- g_l - lower bounds on constraints g(x)
- g_u - upper bounds on constraints g(x)

```
virtual void get_bounds_info(Number* &x_l, Number* &x_u,
                             Number* &u_l, Number* u_u,
                             Number* &g_l, Number* &g_u)
{
  //since states are not bounded we set upper and lowet
  //bounds to positive and negative infinities which are
  //the default values of the nlp_upper_bound_inf and
  //nlp_lower_bound_inf respectively
  x_l[0] = -1.0e19;
  x_l[1] = -1.0e19;
  x_u[0] = +1.0e19;
  x_u[1] = +1.0e19;

  //setting lower and upper bounds on control variable
  u_l[0] = -0.75;
  u_u[0] = +0.75;

  //because the two dynamic constraints are equality
  //constraints we set upper and lower bounds equal to 0
  g_l[0] = 0;
  g_l[1] = 0;
  g_u[0] = 0;
  g_u[1] = 0;
}
```

Method Number eval_f with a prototype:

```
Number eval_f(Number* x, Number* u)
```

Returns the value of the stage cost function.

- x- x vector at the given time step
- u - u vector at the given time step

```
Number VDPNLP :: eval_f(Number* x, Number* u)
{
  Number f =  x[0] * x[0] + x[1] * x[1] + u[0] * u[0];
  return f;
}
```

Method eval_grad_f_x with a prototype:

```
bool eval_grad_f_x(Number* x, Number* u, Number* & grad_f_x)
```

Evaluates the gradient of the stage cost function with respect to x variable

- x - x variable vector at given time step.
- u - u variable vector at given time step.
- grad_f_x - gradient of the cost function with respect to x variable vector.

```
bool VDPNLP :: eval_grad_f_x(Number* x, Number* u,
                             Number* & grad_f_x)
{
  //gradient with respect to x1
  grad_f_x[0] = 2 * x[0];

  //gradient with respect to x2
  grad_f_x[1] = 2 * x[1];

  return true;
}
```

Method eval_grad_f_u with a prototype:

```
bool eval_grad_f_u(Number* x, Number* u, Number* & grad_f_u)
```

Evaluates the gradient of the stage cost function with respect to u variable vector.

```
bool VDPNLP :: eval_grad_f_u(Number* x, Number* u,
                             Number* & grad_f_u)
{
  //gradient with respect to u
  grad_f_u[0] = 2 * u[0];
  return true;
}
```

Method eval_g_init with a prototype:

```
bool eval_g_init(Number* x, Number* u, Number*& g)
```

Evaluates the value of the constraints for the initial condition(t = 0)

- x - x variable vector at t = 0.
- u - u variable vector at t = 0.
- g - the array of the constraints g(x(t = 0), u(t = 0)) for initial condition.

```
bool VDPNLP :: eval_g_init(Number* x, Number* u, Number*& g)
{
  //evaluate initial dynamic constraints, (t=0)
  g[0] = x[0] - 1;
  g[1] = x[1];

  return true;
}
```

Method eval_g

```
bool eval_g(Number* x, Number* u, Number*& g)
```

Evaluates the value of the constraints at the point (x(t), u(t)) for t > 0

- x - x variable vector at t > 0
- u - u variable vector at t > 0
- g - the array of the constraints g(x(t),u(t)) for t > 0

```
bool VDPNLP ::eval_g(Number* x, Number* u, Number*& g)
{
  //evaluate daynamic constraint, t>0
  g[0] = x[1];
  g[1] = (1 - x[0] * x[0]) * x[1] - x[0] + u[0];

  return true;
}
```

Method eval_jac_g_values_x wit a prototype:

```
 bool eval_jac_g_values_x(Number*& values, bool*& flags,
                          Number* x, Number* u)
```

Evaluates the values of the Jacobian with respect to x vector, eq. (3.3.4)

- values - array with the values of the Jacobian
- flags - array with flags determining weather the given element of a Jacobian is zero or not
- x - x variable vector
- u - u variable vector

```
bool VDPNLP::eval_jac_g_values_x(Number*& values, bool*& flags,
                                  Number* x, Number* u)
{
  //we specify the values of the jacobian of the constraint with
  //respect to x, the zero elements are not taken into account,
  //however we write zero to these elements in order not to
  //have random value in the memory block and we set the
  //corresponding element in the flags array to false, so the
  //discretisation module knows it is a zero element
  values[0] = 0;
  flags[0] = false;
  values[1] = 1;
  flags[1] = true;
  values[2] = - 2 * x[0] * x[1] - 1;
  flags[2] = true;
  values[3] = 1 - x[0] * x[0];
  flags[3] = true;
  return true;
}
```

Method eval_jac_g_values_u with a prototype:

```
bool eval_jac_g_values_u(Number*& values, bool*& flags,
                         Number* x, Number* u)
```

Evaluates the values of the Jacobian of the constraints with respect to u vector specified in equations (3.3.5)

- values - array with the values of the Jacobian of the constraints
- flags - array with flags determining weather the given element is zero or not
- x - x variable vector
- u - u variable vector

```
bool VDPNLP :: eval_jac_g_values_u(Number*& values, bool*& flags,
                                   Number* x, Number* u)
{
  //determine the values of tha jacobian of the constraints with
  //respect to u, coding mechanism is the same as in eval_jac_g
  //values_x method
  values[0] = 0;
  flags[0] = false;
  values[1] = 1;
  flags[1] = true;
  return true;
}
```

Method eval_jac_g_s

```
bool eval_jac_g_s(Int*& irow, Int*& jcol)
```

Determines the sparsity structure of the Jacobian of the constraints.

- irow -(out) the row indices of entries in the Jacobian of the constraints.
- jcol - (out) the column indices of entries in the Jacobian of the constraints.

```
bool VDPNLP :: eval_jac_g_s(Int*& irow, Int*& jcol)
{
  irow[0] = 0;
  jcol[0] = 0;
  irow[1] = 0;
  jcol[1] = 3;
  irow[2] = 0;
  jcol[2] = 4;
  irow[3] = 1;
  jcol[3] = 1;
  irow[4] = 1;
  jcol[4] = 2;
  irow[5] = 1;
  jcol[5] = 3;
  irow[6] = 1;
  jcol[6] = 4;
  return true;
}
```

Method eval_h_s with a prototype:

```
bool eval_h_s(Int*& irow_i, Int*& jcol_i, Int*& irow,
              Int*& jcol, Int*& irow_f, Int*& jcol_f );
```

Determines the sparsity structure of the Hessian of the Lagrangian.

- irow_i - (out) the row indices of the entries in the Hessian of the Lagrangian for initial condition (k = 0)
- jcol_i- (out) the column indices of the entries in the Hessian of the Lagrangian for initial condition (k = 0)
- irow- (out) the row indices of the entries in the Hessian of Lagrangian for regular condition
- jcol - (out) the column indices of the entries in the Hessian of the Lagrangian for regular condition
- irow_f - (out) the row indices of the entries in the Hessian of the Lagrangian for final condition (k = N)
- jcol_f - (out) the column indices of the entries in the Hessian of the Lagrangian for final condition (k = N)

```
bool VDPNLP :: eval_h_s(Int*& irow_i, Int*& jcol_i, Int*& irow,
                        Int*& jcol, Int*& irow_f, Int*& jcol_f)
{
  //evaluate the sparsity structure of hessian of lagrangian
  //for initial condition
  irow_i[0] = 0;
  jcol_i[0] = 0;
  irow_i[1] = 1;
  jcol_i[1] = 1;
  irow_i[2] = 2;
  jcol_i[2] = 2;

  //evaluate the sparsity structure of hessian of lagrangian
  //for regular condition
  irow[0] = 0;
  jcol[0] = 0;
  irow[1] = 1;
  jcol[1] = 0;
  irow[2] = 1;
  jcol[2] = 1;
  irow[3] = 2;
  jcol[3] = 2;

  //evaluate the sparsity structure of hessian of lagrangian
  //for final condition
  irow_f[0] = 0;
  jcol_f[0] = 0;
  irow_f[1] = 1;
  jcol_f[1] = 0;
  return true;
}
```

Method eval_h_v with a prototype:

```
bool eval_h_v(Number*& valules, Number* x, Number* u,
              const Number* lambda, Number obj_factor,
              Int k, Int N, Int x_d, Number h)
```

Evaluates the values of the Hessian of the Lagrangian. Please note that the values of this matrix are coming from both second derivatives of the stage cost function as well as model constraints which are multiplied by corresponding lambda factors. Consequently, an offset variable is introduced which enables to keep the track and call the appropriate lambda factors for the given time step in the eval_h_v function. The value of the offset variable should be equal to the product of the number of constraints (x_d) and time step (k) at which the function is called. Those two variables are passed to the eval_h_v as in-parameters for the user to determine the offset.

- values - (out) the values array of the Hessian of the Lagrangian
- x - (in) x variable vector
- u - (in) u variable vector
- lambda - (in) the values for the constraint multipliers, at which the Hessian is to be evaluated.
- obj_factor - (in) factor in front of the objective term in the Hessian,
- k - (in) the time step at which the eval_h_v is called by discretisation module
- N - (in) the number of the time steps
- x_d - (in) the dimension of the x and constraints vector
- h - (in) the value of the single time step

```
bool VDPNLP:: eval_h_v(Number*& values, Number* x, Number* u,
                       const Number* lambda, Number obj_factor,
                       Int k, Int N, Int x_d, Number h)
{
  Index offset = k * x_d;
  //evaluate values of hessian of lagrangian
  //for initial condition (k = 0)
  if(k = 0)
  {
    for(Index i = 0 ; i <=2; i++)
      values[i] = obj_factor * 2 * h;

    }
  //evaluate values of hessian of lagranfian
  //for final condition (k = N)
  else if(k == N)
  {
    values[0] = lambda[offset + 1] * 2 * h * x[1];
    values[1] = lambda[offset + 1] * 2 * h * x[0];
  }
  //evaluate values of hessian of lagrangian
  //for regular condition
  else
  {
    values[0] = obj_factor * 2 * h + lambda[offset + 1]
    * 2 * h * x[1];
    values[1] = lambda[offset + 1] * 2 *h * x[0];
    values[2] = obj_factor * 2 * h;
    values[3] = obj_factor * 2 * h;
  }
  return true;
}
```

Having implemented all the methods in the VDPNLP class, one should now code the main function. This procedure is exactly the same as when coding the nlp in the chapter 2. The only difference is that this time one has to instantiate VDPNLP class and pass it to application factory by smart pointer (smart pointer should store the address of the VDPNLP object). In order to see more details about coding the main function please go to section 2.2.2.

3.6 Oscillator Simulation and Results

In this section the results of Van Der Pol oscillator problem are presented and discussed. The start time is 0 and the final time is 20s. The value of the time step is set to 0.1 of second, and the number of time steps to 200 (200 * 0.1 s = 20s). After passing this problem to Ipopt, it was possible to obtain the solution after 27 Jacobian and 52 objective function evaluations. The plots of control and state variables versus time are presented in figure 3.2. The Ipopt console output is presented below.

```
                          (scaled)
```

```
Objective...............:   2.4651069210031884e+000
Dual infeasibility......:   7.4138736976623641e-010
Constraint violation....:   1.1657341758564144e-015
Complementarity.........:   2.6453533666883465e-009
Overall NLP error.......:   2.6453533666883465e-009

                                 (unscaled)
Objective...............:  2.9531980913618203e+000
Dual infeasibility......:   8.8818206897995131e-010
Constraint violation....:  1.7652546091539989e-014
Complementarity.........:   3.1691333332926395e-009
Overall NLP error.......:   3.1691333332926395e-009


Number of objective function evaluations             = 28
Number of objective gradient evaluations             = 28
Number of equality constraint evaluations            = 28
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 28
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 27
Total CPU secs in IPOPT (w/o function evaluations)   = 1.030
Total CPU secs in NLP function evaluations           = 0.000

EXIT: Optimal Solution Found.
```

Subplots in the figure 3.2 clearly show the initial conditions we imposed on the state variables. Next states and controls were selected in such a way by the Ipopt, that they could satisfy the oscillator model and reach 0 value within 6 seconds. From that moment all the contributions of the state cost function g from eq. (3.3.17a) summed over the remaining horizon obtained its minimum and were equal to 0.

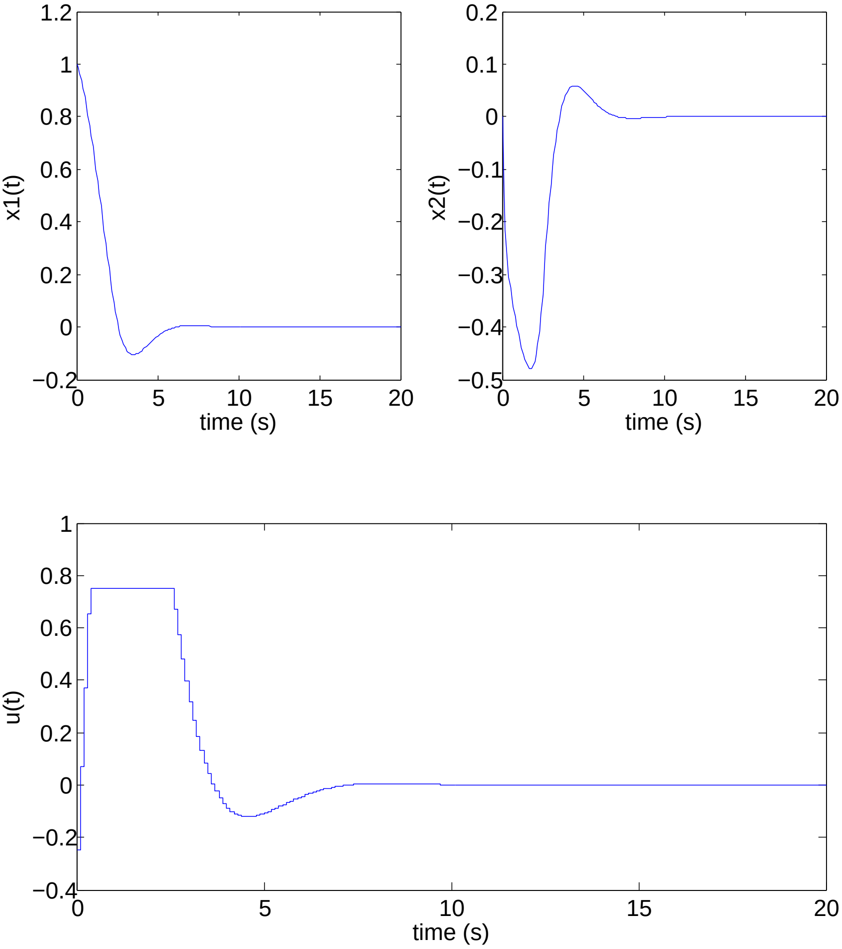


Figure 3.2: Van Der Pol Results

CHAPTER 4

Oil Reservoir Model

So far it was presented how to set up Ipopt software and solve simple non-linear program in the chapter 2. What is more, it was shown how to solve Van Der Pol oscillator optimal control problem with the simultaneous approach in the chapter 3. Consequently, at the current stage the research of the optimisation topic has gone far enough to tackle the oil problem. In this chapter the mathematical model of the two-phase reservoir is derived. The considered model is based on finite volume method (FVM) [37, 38, 39, 40] and conservation of mass for each phase and has already been used in many works in the field of reservoir engineering, e.g. in [2, 41].

4.1 Two-Phase Flow 1-Dimensional Model

In order to derive one-dimensional model, the law of mass conservation is applied, which basically states that the complete mass of an isolated system cannot be changed. Furthermore one dimensional spatial domain is considered as: $\Omega = \{x \in \mathbb{R} : 0 \leq x \leq L\}$, and time domain as: $T = \{t \in \mathbb{R} : t \geq 0\}$ The spatial domain is bounded by: $\delta\Omega = \{x \in \mathbb{R} : x = 0 \wedge x = L\}$, and its interior is: $\Omega^o = \{x \in \mathbb{R} : 0 \leq x \leq L\}$ and the boundaries and interior of time domain are the following: $\delta T = \{T \in \mathbb{R} : t = 0\}$ $T^o = \{t \in \mathbb{R} : t \geq 0\}$. Having defined our spatial and time domains, we will derive partial differential equations of rate of change of oil and water concentrations with

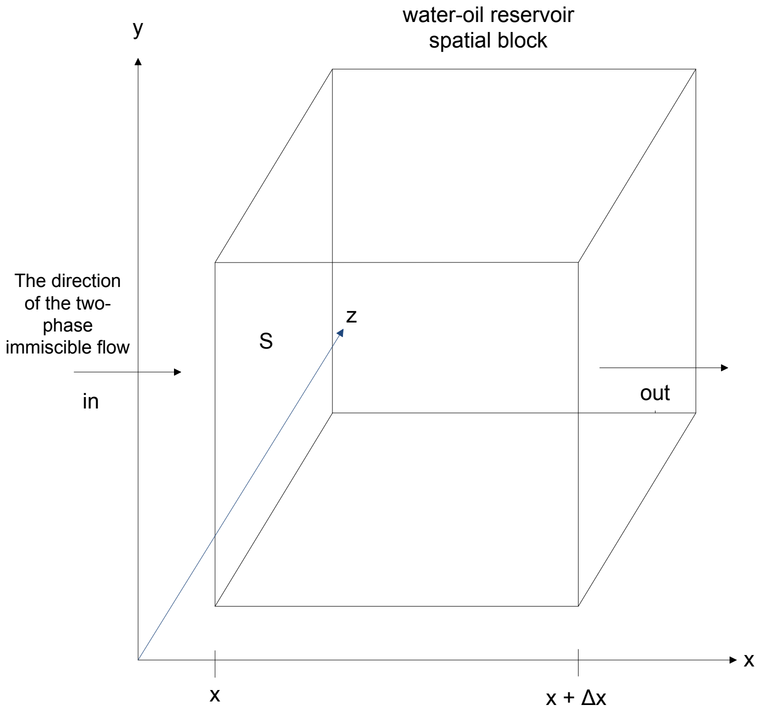


Figure 4.1: Water-Oil Reservoir Spatial Block

respect to time and water, oil fluxes as function of spatial position in the reservoir. Let us introduce a quantity accumulation (Acc in kg), stating how much water or oil was accumulated in the given spatial block of the reservoir in the given time unit, then:

$$Acc = In - Out \tag{4.1.1}$$

Where In and Out are masses of the each phase entering and leaving the given block respectively. Let us for now investigate the flow of the water phase in the reservoir. To do that, we denote the accumulation and concentration term with the subscript w, which will correspond to the water phase. We represent now left-hand side of the equation - accumulation (Acc), as a difference of products of volume of the considered block and water concentrations at start and final time.

$$Acc_w = C_w(t + \Delta t_x, x) \cdot (S \cdot \Delta x) - C_w(t, x) \cdot (S \cdot \Delta x) \tag{4.1.2}$$

where:

- C_w - water concentration in kg/m^3
- $\triangle\ t$ - time in s during which water travels the distance of length of the resvoir block ($\triangle\ x$)
- S - cross-sectional area of the reservoir block in m^2
- $\triangle\ x$ - length of the reservoir block in m

The right-hand side mass terms (In_w and Out_w) of the equation (4.1.1) are represented as products of mass transfer fluxes at the corresponding boundary surfaces, their cross-sectional areas and time which it takes for the fluid to flow through the reservoir.

$$In_w = N_w(x, t) \cdot S \cdot \Delta t \tag{4.1.3a}$$
$$Out_w = N_w(x + \Delta x, t) \cdot S \cdot \Delta t \tag{4.1.3b}$$

where:

- $N_w(x, t)$- water mass transfer flux at the in-boundary surface of the reservoir block, expressed in $kg \cdot m^{-2} \cdot s^{-1}$
- $N_w(x + \Delta x, t)$ - water mass transfer flux at the out-boundary surface

Equations (4.1.2), (4.1.3a), (4.1.3b) are substituted into (4.1.1) which gives the following:

$$(C_w(t + \Delta t, x) - C_w(t, x)) \cdot S \cdot \Delta x = (N_w(t, x) - N_w(t, x + \Delta x)) \cdot S \cdot \Delta t \tag{4.1.4}$$

then both sides of the equation are divided by S, $\vartriangle t$ and $\vartriangle x$, which yields:

$$\frac{C_w(t+\Delta t,x)-C_w(t,x)}{\Delta t}=-\frac{N_w(t,x+\Delta x)-N_w(t,x)}{\Delta x} \tag{4.1.5}$$

From the equation (4.1.5) can be seen that the rate of change of water concentration with respect to time, is equal to the rate of change of mass transfer flux with respect to the distance with minus sign. We consider now the limits of both sides of (4.1.5) for infinitesimally small portions of time $\vartriangle t$ and distance $\vartriangle x$, which gives:

$$\lim_{\Delta t\to 0}\frac{C_w(t+\Delta t,x)-C_w(t,x)}{\Delta t}=-\lim_{\Delta x\to 0}\frac{N_w(t,x+\Delta x)-N_w(t,x)}{\Delta x} \tag{4.1.6}$$

which can also be transcribed into differential equation of the following form:

$$\frac{\partial C_w(t,x)}{\partial t}=-\frac{\partial N_w(t,x)}{\partial x} \tag{4.1.7}$$

The same procedure is performed for the oil phase, which gives the same relation for the rate of change of oil mass concentration and flux. What is more, both equations are enriched by additional source/sink terms which represent contributions coming from injection and production wells respectively (those terms will be zero for all the blocks not directly connected with the production or injection wells), which gives the following set of partial differential equations:

$$\frac{\partial C_w(t,x)}{\partial t}=-\frac{\partial N_w(t,x)}{\partial x}-Q_w \tag{4.1.8a}$$
$$\frac{\partial C_o(t,x)}{\partial t}=-\frac{\partial N_o(t,x)}{\partial x}-Q_o \tag{4.1.8b}$$

where

- Q_w - water sink/source term, since water is both injected at the injection wells and sucked at the production well, this term can become either a sink or a source term depending whether the given grid block is connected with the injection or production well.
- Q_o - oil sink term, since in our problem the only phase injected is water, this term can be only a sink term. In other words $Q_o = 0$ at the injection well.

Please note that at this point of model representation we do not distinguish between sourse or sink nature of the Q term as those equations will refer to al the grid blocks which can be penetrated by either production or injection well. The exact representation of those terms is given in section 4.2. Both fluxes through the porous medium $N_w(x,t), N_o(x,t)$ and mass concentrations $C_w(x,t), C_o(x,t)$ are functions of time t$\in$ T

and position $x \in \Omega$. The initial concentrations of oil and water in the reservoir are given as:

$$C_w(t,x) = C_{w0}(x)\ t \in \delta T, x \in \Omega \tag{4.1.9a}$$

$$C_o(t,x) = C_{o0}(x)\ t \in \delta T, x \in \Omega \tag{4.1.9b}$$

and the fluxes of oil and water through the porous medium at the boundary conditions are:

$$N_w(t,x) = 0\ t \in \delta T, x \in \Omega \tag{4.1.10a}$$

$$N_o(t,x) = 0\ t \in \delta T, x \in \Omega \tag{4.1.10b}$$

Having determined conservation equations and their boundary conditions, one can focus now on constitutive models where Darcy's law will be applied for expressing flow in the porous media. By doing so, it will be possible to introduce in the equations some new physical quantities, describing flows in the reservoir. Firstly water and oil concentrations are represented as products of their porosities, densities and saturations, which gives the following:

$$C_w = \phi\rho_w(P_w)S_w \tag{4.1.11a}$$

$$C_o = \phi\rho_o(P_o)S_o \tag{4.1.11b}$$

where:

- ϕ is the porosity of the reservoir block. Porosity, called also a void fraction, is a measure of the void spaces in the material (in this case rocks) that can be occupied by the fluids (oil and water). Porosity is a fraction of the void volume over the total volume, its values are dimensionless and can range from 0 to 1. In the constitutive model shown in this thesis porosity is assumed to be constant within the reservoir.
- $\rho_o(P_o), \rho_w(P_w)$ - densities of oil and water in kg/m^3. Those densities are functions of oil and water pressures respectively.
- S_o, S_w - saturations of oil and water. Their values can range from 0 to 1 and are equal to fraction of the oil/water volume over the void volume. Since it is assumed that there are no other substances in the void volumes except for oil and water, the void spaces are completely filled with oil and water which gives:

$$S_w + S_o = 1 \tag{4.1.12}$$

Since water and oil travel by convection in the reservoir, their fluxes can be expressed as:

$$N_w = \rho_w(P_w)u_w(P_w, S_w) \tag{4.1.13a}$$

$$N_o = \rho_o(P_o)u_o(P_o, S_o) \tag{4.1.13b}$$

where:

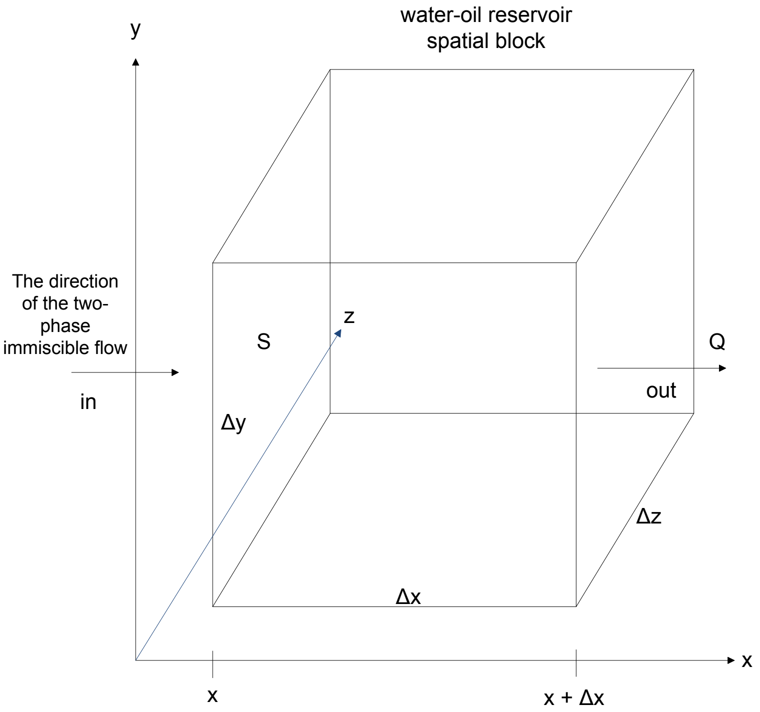


Figure 4.2: Diagram with Definitions and Reference Directions for Darcy's law

- $u_w(P_w, S_w), u_o(P_o, S_o)$ - are the linear velocities of water and oil phase respectively, expressed as functions of pressure and saturation in m/s

The term pressure discharge Q is introduced (measured in units of volume per time, e.g. m^3/s), which according to the Darcy's law, is equal to the product of the permeability of the medium, cross-sectional area of the flow and the pressure drop, all divided by the viscosity and the length over which the pressure is dropping. The concept of Darcy's law has a very wide application in the field of modelling of petroleum reservoirs and is shown on the figure 4.2

$$Q = \frac{-kS(P_{x+\Delta x} - P_x)}{\mu \Delta x} \tag{4.1.14}$$

where:

- k - is the permeability of the medium. Permeability is the measure of a porous media (in this case reservoir rocks) to transmit fluid. The permeability $k = k(x)$ depends only on the spatial position in the reservoir and is a linear function of it. The idea behind permeability is to determine the connectivity and preferred flow direction in the reservoir. The permeability is defined as a tensor of size 3 x 3. Theoretically, k is a full tensor but in many practical applications it can be also assumed that k is a diagonal tensor, that is the one that has non-zero entries only on the diagonal which simply means that all cross terms are equal to zero. One distinguishes between two kinds of permeability fields which are isotropic and unisotropic. In case of an isotropic field, horizontal permeabilities are equal to the vertical permeability ($k_{xx} = k_{yy} = k_{zz}$). In this work, an isotropic permeability field is assumed. The SI unit of permeability is m^2, however, in this thesis the unit millidarcy is used, which is typical for the investigations of petroleum reservoirs.

- S - cross-sectional area through which the flow takes place (unit m^2).

- μ - viscosity of the fluid, known also as thickness or internal friction of the fluid. Viscosity is a measure of the resistance of the fluid to the deformation and its SI unit is $Pa \cdot s$.

- Δx - length over which the pressure drop takes place in m

- $P(x), P(x + \Delta x)$ - initial and final pressures in Pa

Dividing both sides of the (4.1.14) by the area S and applying more general notation gives:

$$q = \frac{-k}{\mu} \Delta P \tag{4.1.15}$$

where:

- q - is the flux (discharge per unit area, with units length per time, e.g m/s). The relation between flux and pressure discharge is the following:

$$q = \frac{Q}{S} \tag{4.1.16}$$

- ΔP - is the pressure gradient vector equal to the rate of change of pressure over the given distance and its SI unit is $Pa{\cdot}m^{-1}$. The relation between pressures at the boundaries and the pressure gradient (known also as a thickness of the medium) is given in the following equation.

$$\Delta P = \frac{P_{x+\Delta x} - P_x}{\Delta x} \tag{4.1.17}$$

Please note that the flux discharge q is not the velocity experienced by the given phase when it is travelling through the porous medium. The pore velocity v of either oil or water phase is related to their corresponding flux discharge by the porosity, which can be written as:

$$v = \frac{q}{n} \tag{4.1.18}$$

The reason why flux discharge, known also as Dary's velocity, is divided by the porosity, is to account for the fact that only a fraction of the total volume is available for flow. Consequently, the pore velocity, considered in some works as the linear velocity, is defined as velocity that a conservative tracer would experience, if taken by the fluid of the given phase through the formation. Multiplying both sides of (4.1.18) by porosity n, yields:

$$vn = q \tag{4.1.19}$$

In this work however, we define the linear velocity as Darcy's velocity q (pressure discharge) introduced in equation (4.1.15), and we do not account for the fact that the medium is porous as in our model we do not have any phenomena influenced by the porosity, such as formation damage or fines migration. This approach has been undertaken in many reservoir simulation works e.g. by Völcker at [2] and yields that the linear velocity u of the given phase used in this model is equal to:

$$u = \frac{-k\Delta P}{\mu} \tag{4.1.20}$$

Applying Darcy's law enabled us to determine linear velocity of the liquid in the reservoir in terms of viscosity, permeability and pressure gradient. Now we rewrite equation (4.1.20) in a differential form and enrich it by relative permeability terms to express linear velocities of each phase:

$$u_w = -k\frac{k_{rw}(S_w)}{\mu_w}\frac{\partial P}{\partial x} \tag{4.1.21a}$$

$$u_o = -k\frac{k_{ro}(S_o)}{\mu_o}\frac{\partial P}{\partial x} \tag{4.1.21b}$$

where:

- k_{rw}, k_{ro}- are the relative permeabilities of water and oil phase respectively. Those permeabilities are non-linear functions of the saturation of each phase and are approximated by the Corey relations:

$$s_w = \frac{S_w - S_{wc}}{1 - S_{wc} - S_{or}} \tag{4.1.22a}$$

$$s_o = \frac{S_o - S_{oc}}{1 - S_{oc} - S_{or}} \tag{4.1.22b}$$

$$k_{rw}(S_w) = \begin{cases} 0 & 0 \le S_w \le S_{wc} \\ k_{rw0} s_w^{n_w} & S_{wc} \le S_w \le 1 - S_{or} \\ k_{rw0} & 1 - S_{or} \le S_w \le 1 \end{cases} \tag{4.1.23a}$$

$$k_{ro}(S_o) = \begin{cases} 0 & 0 \le S_o \le S_{or} \\ k_{ro0} s_o^{n_o} & S_{or} \le S_o \le 1 - S_{wc} \\ k_{ro0} & 1 - S_{wc} \le S_o \le 1 \end{cases} \tag{4.1.23b}$$

where:

- k_{rw0}, k_{ro0} - are permeabilities determined experimentally for each particular porous medium.
- n_w, n_o - are determined experimentally for each particular porous medium.
- S_{wc} - is a critical water saturation.
- S_{or} - is a residual oil saturation.
- s_w, s_o - are reduced saturation of water and oil.

If we assume the compressibilities of oil c_o and water c_w to be constant over the considered pressure range, we can express the densities of the two phases using the following equations of state:

$$\rho_w = \rho_{w0} e^{c_w(P_w - P_{w0})} \tag{4.1.24a}$$

$$\rho_w = \rho_{o0} e^{c_o(P_o - P_{o0})} \tag{4.1.24b}$$

where:

- $\rho_{w0} = \rho_w(P_{w0}), \rho_{o0} = \rho_o(P_{o0})$ are the densities at the reference pressures P_{w0} and P_{o0}.

The pressure difference between wetting fluid (water in our model) and non-wetting fluid is given by the capillary pressure , which is a function of water saturation:

$$P_{cow}(S_w) = P_o - P_w \tag{4.1.25}$$

The pressure in the wetting fluid is less than in the non-wetting one. If the media is highly permeable and porous, then capillary effects are small. In case of dense formations, the capillary pressure introduces an additional diffusive term into the model; please see [2]. The capillary effects can partly be explanation for irreducible saturations of oil and water S_{wc}, S_{or}. Consequently, capillary pressure is taken into account in an implicit way through the relative permeabilities. In our constitutive model we assume zero capillary pressure.

$$P_{cow}(S_w) = 0 \tag{4.1.26}$$

4.2 Well Models

Well modells are the terms realising how much mass of of each phase was pumped or sucked by the well in the given unit of time. Their SI unit is $[kg/s]$. To model injection and production wells, we consider indexes sets I and P for injectors and producers respectively. As a result, the injector locations are at $x_{I,j}$ for $j \in I$ and the producers are at $x_{P,j}$ for $j \in P$. We also introduce Dirac's delta function as:

$$\delta_{I,j} = \delta(x - x_{I,j}) \tag{4.2.1a}$$

$$\delta_{P,j} = \delta(x - x_{P,j}) \tag{4.2.1b}$$

Then the set of injection wells can be modelled as:

$$q^{inj}_{w,j} = q_w(t,x) = \rho_w(P_w) q_j \delta_{I,j} j \in I \tag{4.2.2a}$$

$$q^{inj}_{o,j} = q_o(t,x) = 0 \tag{4.2.2b}$$

where:

- q_j is the volumetric injection rate of water at injection well $j \in I$ in $[m^3/s]$

Since, oil is not injected into the reservoir by the injection wells, $q_o(t,x) = 0$. The set of production wells can be modelled in a similar way, which is:

$$q^{prod}_{w,j} = -\alpha_j w_j \rho_w(P_w) \frac{kk_{rw}(S_w)}{\mu_w}(P_w - P_{well,j})\delta_{P,j} \tag{4.2.3a}$$

$$q^{prod}_{o,j} = -\alpha_j w_j \rho_o(P_o) \frac{kk_{ro}(S_o)}{\mu_o}(P_o - P_{well,j})\delta_{P,j} \tag{4.2.3b}$$

where:

- $P_{well,j}$ - is the pressure at the production well $j \in P$ in Pa
- α_j - is the position of the control valve, and $\alpha_j \in [0,1]$
- w_j - is the index of the production well $j \in P$. Well index is dimensionless quantity directly related to the well describing the geometry of the well [42, 43, 44, 45, 46], e.g. whether it is a central or a corner well. We use Peaceman well indexes[47].

The relation between sink (production) terms introduced in equations (4.1.8a), (4.1.8b) and their corresponding well models is the following:

$$Q_{w,j}^{prod} = \frac{q_{w,j}^{prod}}{V} \tag{4.2.4a}$$

$$Q_{o,j}^{prod} = \frac{q_{o,j}^{prod}}{V} \tag{4.2.4b}$$

Similarly for the source (injection) terms introduced in equations (4.1.8a), (4.1.8b) one can get:

$$Q_{w,j}^{inj} = \frac{q_{w,j}^{inj}}{V} \tag{4.2.5a}$$

$$Q_{o,j}^{inj} = \frac{q_{o,j}^{inj}}{V} = 0 \tag{4.2.5b}$$

Source/sink terms for oil and water phase represent the rate of produced/injected mass of each phase over the grid block volume and their unit is $[kg/(s \cdot m^3)]$

4.3 State Transformation

In our two-phase immiscible flow model we used oil and water concentrations C_o, C_w as state variables. It is possible, however, to transform our model by using equations (4.1.11) (4.1.12) (4.1.13) to compute oil pressures and water saturation, knowing their corresponding C_w and C_o. This will enable us to use oil pressure $P = P(t, x) = P_o(t, x)$ and water saturation $S = S(t, x) = S_w(t, x)$ as new state variables instead of C_w and C_o. Hence, initial conditions of the oil problem expressed in terms of new variables will be the following:

$$S(t, x) = S_0(x) \quad t \in \delta T, x \in \Omega \tag{4.3.1a}$$

$$P(t, x) = P_0(x) \quad t \in \delta T, x \in \Omega \tag{4.3.1b}$$

Chapter 5

Spatial and Time Discretisation

This chapter is devoted to discretisation techniques used for solving numerically governing equations of the two phase flow model. Firstly we discretise the reservoir in space and show that the new model formulation preserves the mass conservation property, which was the initial point for derivation of equations. Next, we discretise the model in time and come up with the residual constraints that will be used in chapter 6 for stating the discrete nonlinear program of oil production

5.1 Spatial Discretisation

Oil problem is more complex than the Van Der Pol Oscilator problem which is mainly due to the differential form of the right hand side of the model, describing the rate of change of oil and water fluxes as functions of positions in the reservoir. In order to get rid of this differantial form and transcribe right-hand side of the equations to the algebraic one, it is necessary to discretise those equations in space. Spatial discretisation is performed based on finite volume method. What is more, Gauss' divergence theorem is employed to express the model in an integral form.

$$\frac{\partial}{\partial t}\int_{\Omega} C_w dV = -\int_{\Omega}(N_w n)dS + \int_{\Omega} Q_w dV \tag{5.1.1a}$$

$$\frac{\partial}{\partial t}\int_{\Omega} C_o dV = -\int_{\Omega}(N_o n)dS + \int_{\Omega} Q_o dV \tag{5.1.1b}$$

Now, it is possible to consider the reservoir not as a continuous space, but a structured grid of finite small 3-dimensional blocks and model it in the following manner:

$$\Omega_{i,j,k} = \{(x,y,z) \in \mathbb{R} : x_{i-} \leq x_{i+}, y_{j-} \leq y_{j+}, z_{k-} \leq z_{k+}\} \tag{5.1.2}$$

where:
$x_{i-} = \sum_{l=1}^{i-1} \Delta_l, x_i = x_{i-} + \frac{1}{2}\Delta$ and $x_{i-} = x_{i-} + \Delta x$, the same rule applies to y and z dimensions, which basically can be understood as every single small reservoir volume $\Omega_{i,j,k}$ has its mid point at (x_i, y_j, z_k).
In this thesis, we assume the constant height (depth) of the reservoir at every point, which enables us to simplify the reservoir model from three to two-dimensional case, where every grid block is of the same height and does not have any adjacent blocks in the z direction. Because of that, we do not consider any water-oil flow in the z direction, and every grid block will have the same index k equal to unity. Consequently, from now on it will be omitted in the further notations. Next, we compute the oil and water concentration in each control volume $\Omega_{i,j}$ by solving equations (5.1.1).

$$\frac{\partial}{\partial t}\int_{\Omega_{i,j}} C dV = \frac{dC_{i,j}}{dt}(t)V_{i,j} \tag{5.1.3}$$

where:

$$C = \begin{bmatrix} C_o \\ C_w \end{bmatrix} \tag{5.1.4}$$

and finite volume is expressed as:

$$V_{i,j} = \Delta x_i \Delta y_j \Delta z_k \tag{5.1.5}$$

The flux terms at the boundary conditions are resolved into rectangular components in x and y dimension (in case of three dimensional model we would also analyse flux terms in z direction) and can be expressed as:

$$\int_{\Omega_{i,j}} (N \cdot n)dS = S_{x,i}(N_x(t, x_{i+}, y_j) - N_x(t, x_{i-}, y_j)) + S_{y,j}(N_y(t, x_{i+}, y_j) - N_y(t, x_i, y_{j-})) \tag{5.1.6}$$

where:
S denotes a boundary surface of the corresponding grid block in the given direction:

$$S_{x,j,k} = \Delta y_j \Delta z_k \tag{5.1.7a}$$

$$S_{y,i,k} = \Delta x_i \Delta z_k \tag{5.1.7b}$$

Then, we introduce a new term called fluid transmisibility $\zeta_{i,j} = \rho(P_{i,j})k_r(S_{i,j})/\mu$ which is a product of density and relative permeability divided by the viscosity. Next, we express every rectangular component of the flux in terms of fluid transmisibility, permeability of the grid block and pressure difference, which yields:

$$N_{x,i+,j} = -k_{x,i+,j}\zeta_{i+,j}\frac{P_{i+1,j} - P_{i,j}}{x_{i+1} - x_i} \tag{5.1.8a}$$

$$N_{x,i-,j} = -k_{x,i-,j}\zeta_{i-,j}\frac{P_{i,j} - P_{i-1,j}}{x_i - x_{i-1}} \tag{5.1.8b}$$

$$\Delta N_{x,i,j} = N_{x,i+,j} - N_{x,i-,j} \tag{5.1.8c}$$

Equation(5.1.8c) expresses flux in the x direction in the given grid block with index i,j, flux contribution in y direction can be expressed following the same logic. To sustain flux continuity on the interfaces of the neighbouring grid blocks, $k_{x,i+,j}$ and $k_{x,i-,j}$ are computed using the harmonic average for the permeabilities in the adjacent grid blocks. Also the fluid transmissibilities $\zeta_{i+,j}$ and $\zeta_{i-,j}$ are obtained by using upstream information. The single point upstream scheme used in this work is given in Völcker at[2]. The flux contributions in y direction are expressed in the same way as for the x direction. In case of three-dimensional model, the flux in z direction incorporates the additional gravity term. The source terms in equation (5.1.1) are expressed as:

$$\int_{\Omega_{i,j}} QdV = Q_{i,j}(t)V_{i,j} \tag{5.1.9}$$

where:

$$Q = \begin{bmatrix} Q_w \\ Q_o \end{bmatrix} \tag{5.1.10}$$

Finally, we can express our model from equation (5.1.1) as a system of ordinary differential equations of the following form:

$$\frac{dC_{i,j}}{dt} = -\left(\frac{\Delta N_{x,i,j}}{\Delta x} + \frac{\Delta N_{y,i,j}}{\Delta y}\right) + Q_{i,j} \tag{5.1.11}$$

5.2 New Model Formulation

We can recall ourselves that the model of the Van Der Pol oscilator problem was of the following form:

$$\frac{d}{dt}x(t) = f(t, x(t)) \tag{5.2.1}$$

which is not the case for the oil problem, as it is denoted as:

$$\frac{d}{dt}g(x(t)) = f(t, x(t)) \tag{5.2.2}$$

This model presented in eq. (5.2.2) was proposed and tried out for the simulation purposes in many works e.g. in [48, 8, 41]. The reason why the model formulation is so important, is to ensure that the new transcribed model still preserves the mass conservation principle as the partially differential equations constituting continuous one-dimensional, two-phase flow model given in (4.1.8a) and (4.1.8b) were derived based on this property. In optimisation problems involving process simulations, reformulating the problem and discretising it in time or space is always a challenge since it is often the case that a new discrete model should preserve such properties as e.g. conservation of mass , energy, or momentum [48]. This is due to the fact that these properties are the initial outlet for the constraints definitions.

In our new model $x(t)$ represents states which are oil pressures and water saturations for every grid block of the reservoir. $g(x(t))$ are the conserved properties which are the oil and water mass concentrations in the given grid block. The general form of the model in (5.2.2) corresponds to our particular reservoir model discretised in space in (5.1.11) where:

$$x(t) = \begin{bmatrix} x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{n,n} \end{bmatrix} = \begin{bmatrix} P_{o1,1} \\ S_{w1,1} \\ P_{o1,2} \\ S_{w,1,2} \\ \vdots \\ P_{on,n} \\ S_{wn,n} \end{bmatrix} \tag{5.2.3}$$

where:

- n is a number of grid blocks in x and y direction. In this thesis we consider an exemplary square oil water reservoir which consists of the same number of grid blocks in x and y direction and that is the reason why the last element of the state vector has index of (n,n).
- $x_{i,j}$ is a two dimensional vector storing water saturation $S_{i,j}$ and oil pressure $P_{i,j}$ in the grid block with index (i,j).

The left-hand side model vector function stores the water and oil concentrations for every grid block of the reservoir and is denoted in the following manner:

$$g(x(t)) = \begin{bmatrix} g_{1,1} \\ g_{1,2} \\ \vdots \\ g_{n,n} \end{bmatrix} = \begin{bmatrix} C_{o1,1} \\ C_{w1,1} \\ C_{o1,2} \\ C_{w1,2} \\ \vdots \\ C_{on,n} \\ C_{wn,n} \end{bmatrix} \tag{5.2.4}$$

where:

- $g_{i,j}$ is a two-dimensional vector function containing water and oil concentrations$C_{oi,j}$, $C_{wi,j}$ in the grid block with the index i,j.

The left hand side of the (5.2.2) is the following:

$$f(x(t)) = \begin{bmatrix} f_{1,1} \\ f_{1,2} \\ \vdots \\ f_{n,n} \end{bmatrix} = \begin{bmatrix} f_{o1,1} \\ f_{w1,1} \\ f_{o1,2} \\ f_{w1,2} \\ \vdots \\ f_{on,n} \\ f_{wn,n} \end{bmatrix} \tag{5.2.5}$$

where:

- $f(i, j)$ is a two dimensional vector function storing fluxes of oil and water in the grid block with index i,j.

We introduce the Jacobians of the f(x(t)) and g(x(t)), with respect to the states, which are the following:

$$\frac{\partial g}{\partial x} = \begin{bmatrix} \frac{\partial g_{1,1}}{\partial x_{1,1}} & \frac{\partial g_{1,1}}{\partial x_{1,2}} & \frac{\partial g_{1,1}}{\partial x_{1,3}} & \dots & \frac{\partial g_{1,1}}{\partial x_{n,n}} \\ \frac{\partial g_{1,2}}{\partial x_{1,1}} & \frac{\partial g_{1,2}}{\partial x_{1,2}} & \frac{\partial g_{1,2}}{\partial x_{1,3}} & \dots & \frac{\partial g_{1,2}}{\partial x_{n,n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_{n,n}}{\partial x_{1,1}} & \frac{\partial g_{n,n}}{\partial x_{1,2}} & \frac{\partial g_{n,n}}{\partial x_{1,3}} & \dots & \frac{\partial g_{n,n}}{\partial x_{n,n}} \end{bmatrix} \tag{5.2.6}$$

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_{1,1}}{\partial x_{1,1}} & \frac{\partial f_{1,1}}{\partial x_{1,2}} & \frac{\partial f_{1,1}}{\partial x_{1,3}} & \dots & \frac{\partial f_{1,1}}{\partial x_{n,n}} \\ \frac{\partial f_{1,2}}{\partial x_{1,1}} & \frac{\partial f_{1,2}}{\partial x_{1,2}} & \frac{\partial f_{1,2}}{\partial x_{1,3}} & \dots & \frac{\partial f_{1,2}}{\partial x_{n,n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n,n}}{\partial x_{1,1}} & \frac{\partial f_{n,n}}{\partial x_{1,2}} & \frac{\partial f_{n,n}}{\partial x_{1,3}} & \dots & \frac{\partial f_{n,n}}{\partial x_{n,n}} \end{bmatrix} \tag{5.2.7}$$

As $x_{i,j}$ and $g_{i,j}$ and $f_{i,j}$ are two dimensional vectors, every element of the presented above g and f jacobian is a 2 by 2 square matrix itself. e.g:

$$\frac{\partial g_{i,j}}{x_{i,j}} = \begin{bmatrix} \frac{\partial g_{oi,j}}{\partial x_{oi,j}} & \frac{\partial g_{oi,j}}{\partial x_{wi,j}} \\ \frac{\partial g_{wi,j}}{\partial x_{oi,j}} & \frac{\partial g_{wi,j}}{\partial x_{wi,j}} \end{bmatrix} = \begin{bmatrix} \frac{\partial C_{oi,j}}{\partial P_{oi,j}} & \frac{\partial C_{oi,j}}{\partial S_{wi,j}} \\ \frac{\partial C_{wi,j}}{\partial P_{oi,j}} & \frac{\partial C_{wi,j}}{\partial S_{wi,j}} \end{bmatrix} \tag{5.2.8}$$

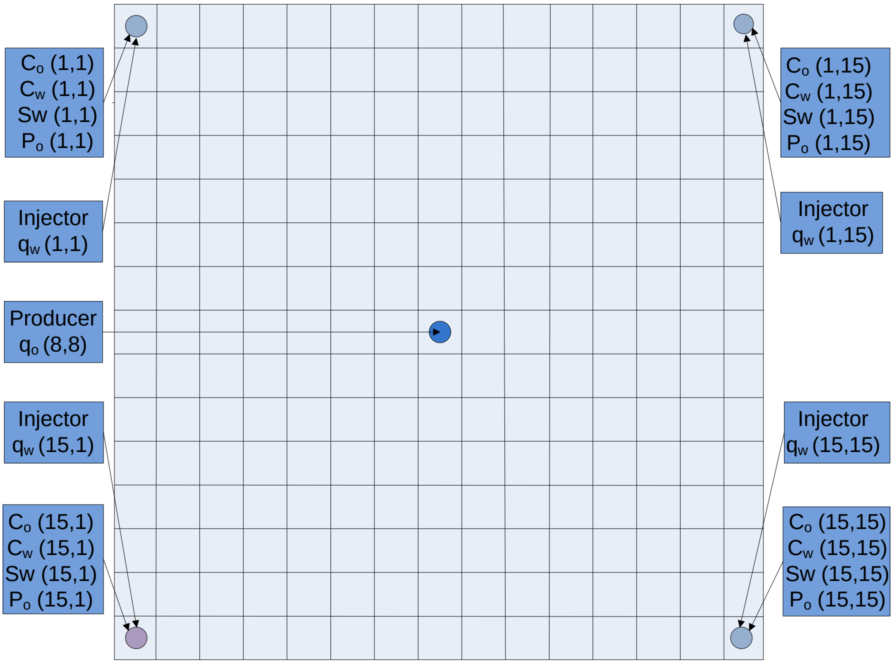


Figure 5.1: Water-Oil Discretised Reservoir with 4 Injectors and One Producer

$$\frac{\partial f_{i,j}}{x_{i,j}} = \begin{bmatrix} \frac{\partial f_{oi,j}}{\partial x_{oi,j}} & \frac{\partial f_{oi,j}}{\partial x_{wi,j}} \\ \frac{\partial f_{wi,j}}{\partial x_{oi,j}} & \frac{\partial f_{wi,j}}{\partial x_{wi,j}} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_{oi,j}}{\partial P_{oi,j}} & \frac{\partial f_{oi,j}}{\partial S_{wi,j}} \\ \frac{\partial f_{wi,j}}{\partial P_{oi,j}} & \frac{\partial f_{wi,j}}{\partial S_{wi,j}} \end{bmatrix} \tag{5.2.9}$$

For the better clarity of the mathematical representation of our model we present a discretised grid of the reservoir with the corresponding concentrations $C_{i,j}$ and state variables. The reservoir is organised in such a way that it consists of 225 grid blocks (15 cubic grid blocks along x and y axis) and it is equipped with 4 water injectors, one at each corner, and one oil producer located in the middle. By looking at the figure and Jacobian matrix one can picture himself the size of the oil problem e.g. in case of the reservoir having 15 grid blocks along x and y direction, one ends up with 225 grid blocks and 450 equality constraints on 450 state variables. Consequently, the Jacobian matrix of the two-phase flow model dg/dx will be 450 by 450 element square matrix. The complexity of the oil problem caused by its size can be, however, resolved by determining the sparsity structure of the Jacobian. If we recall ourselves the equation (5.1.8), which represents the flux of each phase for every single grid block, we can see that it involves only the pressure and saturation values of the of block itself and his neighbours. In other words, only the states of the neighbouring grid blocks or the states of the element itself can influence the particular flux or concentration value. Consequently, all the elements of the Jacobians of f(x(t),u(t)) and g(x(t)), which represent the partial derivatives of the constraints for the given block with respect to states not corresponding to either this block or its neighbours, will be zero. e.g $\frac{\partial g_{1,1}}{x_{1,3}} = 0$ because the grid block 1,3 is not a neighbour of 1,1. which means that $g_{1,1}$ is not a function of $x_{1,3}$

5.3 Discretisation of the Model in Time

We have already discretised the dynamic constraints of the oil problem in space and presented a new form of mathematical model which still preserves the primary properties such as conservation of mass. Now, we are going to completely eliminate the differential form of the constraints by transcribing them into residual ones. In order to perform the temporal discretisation we apply implicit Euler method which yields:

$$R_k(x_k, u_{k-1}, x_{k-1}) = g(x_k) - hf(x_k, u_{k-1}) - g(x_{k-1}) \;\; for \;\; k = 1, 2, 3...N \tag{5.3.1}$$

where:

- x_k - set of all the states at the time step k.
- u_k - set of all the controls at the time step k.
- h - the size of the time step.

The residual constraint of the initial condition is:

$$R_0 = x_0 - a = 0; \tag{5.3.2}$$

where a stores the initial values of pressures and water saturarions at all grid blocks at time step k equal to 0. Following the same logic as when constructing the Jacobian of the residual constraints for the Van Der Pol problem we introduce a vector z storing all the states x and controls u for all the time steps which yields:

$$R(z) = 0 \tag{5.3.3}$$

And the Jacobian $\frac{\partial R}{\partial z}$of the residual constraint function for the N time steps is:

$$\frac{\partial R}{\partial Z}=\begin{bmatrix} \frac{\partial R_0}{\partial x_0} & \frac{\partial R_0}{\partial u_0} & \frac{\partial R_0}{\partial x_1} & \frac{\partial R_0}{\partial u_1} & \cdots & \frac{\partial R_0}{\partial x_{N-1}} & \frac{\partial R_0}{\partial u_{N-1}} & \frac{\partial R_0}{\partial x_N} \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial x_{N-1}} & \frac{\partial R_1}{\partial u_{N-1}} & \frac{\partial R_1}{\partial x_N} \\ \frac{\partial R_2}{\partial x_0} & \frac{\partial R_2}{\partial u_0} & \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial u_1} & \cdots & \frac{\partial R_2}{\partial x_{N-1}} & \frac{\partial R_2}{\partial u_{N-1}} & \frac{\partial R_2}{\partial x_N} \\ \frac{\partial R_3}{\partial x_0} & \frac{\partial R_3}{\partial u_0} & \frac{\partial R_3}{\partial x_1} & \frac{\partial R_3}{\partial u_1} & \cdots & \frac{\partial R_3}{\partial x_{N-1}} & \frac{\partial R_3}{\partial u_{N-1}} & \frac{\partial R_3}{\partial x_N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \frac{\partial R_N}{\partial x_0} & \frac{\partial R_N}{\partial u_0} & \frac{\partial R_N}{\partial x_1} & \frac{\partial R_N}{\partial u_1} & \cdots & \frac{\partial R_N}{\partial x_{N-1}} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N} \end{bmatrix} \tag{5.3.4}$$

What is more, every entry with respect to the vector of the control variables can be computed from the equation below:

$$\frac{\partial R_n}{u_n}=-h\frac{\partial f}{\partial u} \tag{5.3.5}$$

Every entry of the Jacobian of the residual constraints determined in eq (5.5.2) is a matrix itself, which size depends on the number of the finite volumes constituting a reservoir and number of control variables (water injectors and oil producers). We have already tried to picture the scale of the size of the oil problem by analysing the size of the Jacobian $\frac{\partial g}{\partial x}$, which is a 450x450 square matrix for an oil reservoir consisting of 15 blocks along x and y direction. The Jacobian of the residual constrains $\frac{\partial R}{\partial z}$ is however, much bigger since it stores Jacobians $\frac{\partial g}{\partial x}$ of the model for every single time step. As a result, analysing sparsity of this matrix will contribute a lot to the robustness and efficiency of the computation. Below we present the Jacobian of the residual constraints in the sparse format:

$$\frac{\partial R}{\partial Z}=\begin{bmatrix} \frac{\partial R_0}{\partial x_0} & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial u_1} & \frac{\partial R_2}{\partial x_2} & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial R_3}{\partial x_2} & \frac{\partial R_3}{\partial u_2} & \frac{\partial R_3}{\partial x_3} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \frac{\partial R_N}{\partial x_N-1} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N} \end{bmatrix} \tag{5.3.6}$$

where the non zero elements can be camputed by calling the Jacobians of the model defined in (5.2.7) and (5.2.6) e.g.:

$$\frac{\partial R_0}{\partial x_0} = I \tag{5.3.7a}$$

$$\frac{\partial R_1}{x_0} = -\frac{\partial g}{\partial x}(x_0) \tag{5.3.7b}$$

$$\frac{\partial R_1}{\partial u_0} = -h\frac{\partial f}{\partial u}(x_1, u_0) \tag{5.3.7c}$$

$$\frac{\partial R_1}{\partial x_1} = \frac{\partial g}{\partial x}(x_1) - h\frac{\partial f}{\partial x}(x_1, u_0) \tag{5.3.7d}$$

$$\frac{\partial R_N}{\partial x_{N-1}} = -\frac{\partial g}{\partial x}(x_{N-1}) \tag{5.3.7e}$$

$$\frac{\partial R_N}{\partial u_{N-1}} = -h\frac{\partial f}{\partial u}(x_N, u_{N-1}) \tag{5.3.7f}$$

$$\frac{\partial R_N}{\partial x_N} = \frac{\partial g}{\partial x}(x_N) - h\frac{\partial f}{\partial x}(x_N, u_{N-1}) \tag{5.3.7g}$$

Jacobians of the functions g(x) and f(x,u) with respect to x were already determined in the equations (5.2.6), (5.2.7). The main difference between the representation of the Jacobian of the residual constraints for Van Der Pol and oil problem is that, in case of oil problem we have to call $\frac{\partial g}{\partial X}$ whereas in the VanDerPol problem we were calling identity matrix. This is because our new residuals in the oil problem do not contain states in a direct way, but in an implicit one which is as arguments of the g function.

5.4 Inequality Movement Constraints

In order to represent the physical limitations on the controls of water-flooding problem we use movement constraints of the following form:

$$u^{\Delta}_{min} \leq \frac{du}{dt}(t) \leq u^{\Delta}_{max} \tag{5.4.1}$$

Since we use zero-order-hold-parametrisation, after discretising the continuous time optimal control problem, these constraints can be defined in the following manner:

$$-u^{\Delta}_{min} \leq \Delta u_k \leq u^{\Delta}_{max} \tag{5.4.2}$$

where:

$$\Delta u_k = u_k - u_{k-1} \quad for \;\; k = 1..N-1 \tag{5.4.3}$$

Let us denote $q^{inj}_{w,k}$ and BHP_k as vectors storing the values of the manipulated variables - water injection rates at the injectors and bottom whole pressures at the producers at time step k. Let the index n represent the number of injectors and index m the number of producers. We represent the controls in the following vector forms:

$$q_{w,k}^{inj} = \begin{bmatrix} q_{w,1,k}^{inj} \\ q_{w,2,k}^{inj} \\ \vdots \\ q_{w,n,k}^{inj} \end{bmatrix} \tag{5.4.4}$$

$$BHP_k = \begin{bmatrix} BHP_{1,k} \\ BHP_{2,k} \\ \vdots \\ BHP_{m,k} \end{bmatrix} \tag{5.4.5}$$

Then we express the control vector u_k from eq. (5.4.2) in the following form:

$$u_k = \begin{bmatrix} q_{w,k}^{inj} \\ BHP_k \end{bmatrix} \tag{5.4.6}$$

5.5 Jacobian Matrix of the Waterflooding Discrete Non-linear Program

One of the main features of the simultaneous method is that it treats state and control variables as optimisation variables. Consequently, problems transcribed by this method are expressed in terms of newly obtained variable vector e.g. z which stores both control and state variables. The algorithm solving discrete non-linear program obtained from direct transcription not only does not see the difference between state and control variables but also will handle inequality movement constraint in almost the same way as residual equality constraints on the state variables. In this section we will wrap up inequality movement constraints with the algebraic residual constraints into one vector function, in order to follow this uniform algorithmic approach. In order to do that we introduce a new vector function $U_k(u_k, u_{k-1})$ denoting the inequality movement constraints at time time step k, given in eq. 5.4.2 and vector function $C(z)$ storing the residual algebraic constraints and inequality movement constraints for all the time steps, then:

$$C(z) = \begin{bmatrix} R_0(x_0) \\ R_1(x_1, x_0, u_0) \\ U_1(u_1, u_0) \\ R_2(x_2, x_1, u_1) \\ U_2(u_2, u_1) \\ R_3(x_3, x_2, u_2) \\ U_3(u_3, u_2) \\ \vdots \\ R_{N-1}(x_{N-1}, x_{N-2}, u_{N-2}) \\ U_{N-1}(u_{N-1}, u_{N-2}) \\ R_N(x_N, x_{N-1}, u_{N-1} \end{bmatrix} \tag{5.5.1}$$

The Jacobian matrix of C(z) with respect to all the states and controls stored in z will be the following:

$$\frac{\partial C}{\partial Z} = \begin{bmatrix}
\frac{\partial R_0}{\partial x_0} & \frac{\partial R_0}{\partial u_0} & \frac{\partial R_0}{\partial x_1} & \frac{\partial R_0}{\partial u_1} & \cdots & \frac{\partial R_0}{\partial x_{N-1}} & \frac{\partial R_0}{\partial u_{N-1}} & \frac{\partial R_0}{\partial x_N} \\
\frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & \frac{\partial R_1}{\partial u_1} & \cdots & \frac{\partial R_1}{\partial x_{N-1}} & \frac{\partial R_1}{\partial u_{N-1}} & \frac{\partial R_1}{\partial x_N} \\
\frac{\partial \Delta u_1}{\partial x_0} & \frac{\partial \Delta u_1}{\partial u_1} & \frac{\partial \Delta u_1}{\partial x_1} & \frac{\partial \Delta u_1}{\partial u_1} & \cdots & \frac{\partial \Delta u_1}{\partial x_{N-1}} & \frac{\partial \Delta u_1}{\partial u_{N-1}} & \frac{\partial \Delta u_1}{\partial x_N} \\
\frac{\partial R_2}{\partial x_0} & \frac{\partial R_2}{\partial u_0} & \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial u_1} & \cdots & \frac{\partial R_2}{\partial x_{N-1}} & \frac{\partial R_2}{\partial u_{N-1}} & \frac{\partial R_2}{\partial x_N} \\
\frac{\partial \Delta u_2}{\partial x_0} & \frac{\partial \Delta u_2}{\partial u_1} & \frac{\partial \Delta u_2}{\partial x_1} & \frac{\partial \Delta u_2}{\partial u_1} & \cdots & \frac{\partial \Delta u_2}{\partial x_{N-1}} & \frac{\partial \Delta u_2}{\partial u_{N-1}} & \frac{\partial \Delta u_2}{\partial x_N} \\
\frac{\partial R_3}{\partial x_0} & \frac{\partial R_3}{\partial u_0} & \frac{\partial R_3}{\partial x_1} & \frac{\partial R_3}{\partial u_1} & \cdots & \frac{\partial R_3}{\partial x_{N-1}} & \frac{\partial R_3}{\partial u_{N-1}} & \frac{\partial R_3}{\partial x_N} \\
\frac{\partial \Delta u_3}{\partial x_0} & \frac{\partial \Delta u_3}{\partial u_1} & \frac{\partial \Delta u_3}{\partial x_1} & \frac{\partial \Delta u_3}{\partial u_1} & \cdots & \frac{\partial \Delta u_3}{\partial x_{N-1}} & \frac{\partial \Delta u_3}{\partial u_{N-1}} & \frac{\partial \Delta u_3}{\partial x_N} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
\frac{\partial R_{N-1}}{\partial x_0} & \frac{\partial R_{N-1}}{\partial u_0} & \frac{\partial R_{N-1}}{\partial x_1} & \frac{\partial R_{N-1}}{\partial u_1} & \cdots & \frac{\partial R_{N-1}}{\partial x_{N-1}} & \frac{\partial R_{N-1}}{\partial u_{N-1}} & \frac{\partial R_{N-1}}{\partial x_N} \\
\frac{\partial \Delta u_{N-1}}{\partial x_0} & \frac{\partial \Delta u_{N-1}}{\partial u_0} & \frac{\partial \Delta u_{N-1}}{\partial x_1} & \frac{\partial \Delta u_{N-1}}{\partial u_1} & \cdots & \frac{\partial \Delta u_{N-1}}{\partial x_{N-1}} & \frac{\partial \Delta u_{N-1}}{\partial u_{N-1}} & \frac{\partial \Delta u_{N-1}}{\partial x_N} \\
\frac{\partial R_N}{\partial x_0} & \frac{\partial R_N}{\partial u_0} & \frac{\partial R_N}{\partial x_1} & \frac{\partial R_N}{\partial u_1} & \cdots & \frac{\partial R_N}{\partial x_{N-1}} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N}
\end{bmatrix} \tag{5.5.2}$$

where the entries coming from the residual equality constraints were already computed in equations (5.3.7) and the entries coming from the inequality movement constraints can be computed in the following manner:

$$\frac{\partial \Delta u_k}{\partial u_k} = I \tag{5.5.3a}$$

$$\frac{\partial \Delta u_k}{\partial u_{k-1}} = -I \tag{5.5.3b}$$

Consequently the Jacobian $\frac{\partial C}{\partial Z}$ represented in the sparse format looks the following:

$$\frac{\partial R}{\partial Z} = \begin{bmatrix}
\frac{\partial R_0}{\partial x_0} & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
\frac{\partial R_1}{\partial x_0} & \frac{\partial R_1}{\partial u_0} & \frac{\partial R_1}{\partial x_1} & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & -I & 0 & I & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{\partial R_2}{\partial x_1} & \frac{\partial R_2}{\partial u_1} & \frac{\partial R_2}{\partial x_2} & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -I & 0 & I & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{\partial R_3}{\partial x_2} & \frac{\partial R_3}{\partial u_2} & \frac{\partial R_3}{\partial x_3} & \cdots & 0 & 0 & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \frac{\partial R_{N-1}}{\partial x_{N-2}} & \frac{\partial R_{N-1}}{\partial u_{N-2}} & \frac{\partial R_{N-1}}{\partial x_{N-1}} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & -I & 0 & I & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{\partial R_N}{\partial x_{N-1}} & \frac{\partial R_N}{\partial u_{N-1}} & \frac{\partial R_N}{\partial x_N}
\end{bmatrix} \tag{5.5.4}$$

CHAPTER 6

Oil Production Optimisation

In this chapter we present the numerical experiment of testing simultaneous method for optimising oil production. In order to increase the economic value of the reservoir and investigate the behaviour of the fluids in the subsurface, we use the two-phase flow model defined in chapter 4. The model was discretised in space by the finite volume method and Gauss' divergence theorem. The time discretisation was performed by using first order ESDIRK method called implicit Euler. The solution to the constrained optimisation nonlinear problem is found by interior point algorithm in the linesearch framework with BFGS approximation for the second order derivatives. The simultaneous search provided by the direct transcription method couples reservoir model with the optimal point solution reducing the number of simulations in the gradient based iteration process.
This chapter is organised in the following way. Section 6.1 presents the waterflooding optimal control problem. In section 6.2 we describe the economic objective with respect to which optimisation is performed. In the section 6.3 we test simultaneous method on a particular production scenario and discuss the results.

6.1 Oil Optimal Control Problem

We have derived partial differential equations governing the flow of each phase in the chapter 4 and shown that the obtained model preserves the mass property which was the initial assumption for the model formulation. We are going to use the two phase flow model from (5.2.2) to formulate the waterflooding problem as a continuous Bolza problem.

$$\min_{\{u(t),x(t)\}_{t_0}^{t_f}} \int\limits_{t_0}^{t_f} J(x(t),u(t))dt \tag{6.1.1a}$$

$$s.t. \quad x(t_0) = x_0 \tag{6.1.1b}$$

$$\frac{dg}{dt}(x(t)) = f(x(t),u(t)) \quad t \in [t_0, t_f] \tag{6.1.1c}$$

$$u_{min}^{\Delta} \leq \frac{du}{dt}(t) \leq u_{max}^{\Delta} \tag{6.1.1d}$$

$$u_{\min} \leq u(t) \leq u_{\max} \tag{6.1.1e}$$

The eq(6.1.1c) describes the two phase black oil simulator, which represents the physical system of the reservoir as a plant, whereas constraints (6.1.1d) should be understood as movement ones and model the physical limitations on adjusting the controlable inputs. Then in order to transcribe the infinite dimensional problem into numerically traceable one, we use a direct collocation method and fully discretise the optimal control problem by approximating the controls and states as piecewise polynomial functions on finite elements by applying implicit first order Runge Kutta method. This enables to represent optimal control problem as a discrete nonlinear program of the following form:

$$\min_{\{u_k,x_k\}} \phi = \sum_{k=0}^{N-1} J(x_k,u_k) \tag{6.1.2a}$$

$$s.t. \quad R_0(x_0) = 0 \tag{6.1.2b}$$

$$R_{k+1}(x_{k+1}, x_k, u_k) = 0 \tag{6.1.2c}$$

$$u_{min}^{\Delta} \leq \Delta u_k \leq u_{max}^{\Delta} \tag{6.1.2d}$$

$$u_{\min} \leq u_k \leq u_{\max} \tag{6.1.2e}$$

Or alternatively in a simultaneous manner:

$$\min_{\{z\}} \phi(z) \tag{6.1.3a}$$

$$s.t. \quad C(z) \leq 0 \tag{6.1.3b}$$

$$z_{\min} \leq z \leq z_{\max} \tag{6.1.3c}$$

For the notational convenience and consistence with the solution of Van Der Poll problem, the residual function(6.1.2c) representing the simulator was described further in eq. (5.3.1). The upper and lower bounds on the manipulated variables are set in the following manner. The lower bound value $u_{min} = q_{inj,w,min}$ is set to 0 m^3/day whereas the upper bound on the injection rate is computed in such a way that no more that given number of porous volumes is injected throughout the whole production. Following those bound constraints one will implicitly satisfy:

$$0 \leq \sum_{k=0}^{N-1} \sum_{j=1}^{N_{inj}} \int_{t_k}^{t_{k+1}} q_{w,j}^{inj}(t)dt \leq PV_{max} \tag{6.1.4}$$

The lower and upper bounds on bottom whole pressures($u_{min} = BHP_{min}$ and $u_{max} = BHP_{max}$) are set to 150 and 200 bars respectively. Those values are typical in petroleum industry for the production wells. Bascially the upper bound on the bottom whole pressure is set in such a manner that it is lower than the initial pressure in the reservoir. This quarantees the pressure difference and flow from reservoir to the producers.

6.2 Stage Cost Function

The stage cost function(J) of the oil problem is used to maximise the net present value (NPV) of oil production and is related to oil recovery as a function of dynamic valve settings of injection rates at the injection wells and bottom whole pressures at the production wells.

$$J = -\sum_{k=1}^{N} \Delta(t_k) \frac{\sum_{j=1}^{N_{prod}} [r_o q_{o,j}(k) - r_{ws} q_{w,j}(k)] - \sum_{j=1}^{N_{inj}} [r_w q_{w,j}(k)]}{(1+b)^a} \tag{6.2.1}$$

where:

- k - time step counter
- N - total number of time steps in the simulation
- N_{inj} - the total number of injection wells in the reservoir
- N_{prod} - the total number of production wells in the reservoir
- j - the well segment counter
- q_o - oil flow rate. $[m^3/s]$
- q_w - water flow rate $[m^3/s]$

- r_o - oil price 5 [\$ $/m^3$]
- r_w - water price 283 [\$ $/m^3$]
- r_{ws} - water, oil separation price 10 [\$ $/m^3$]
- $\Delta(t_k)$ time step k [s]
- b - discount factor expressed as a fraction per year. The idea behind using a discount factor is to compensate for the fact that money loose its value with time.
- a - number of years passed since the start of production.

In order to estimate NPV, values of produced oil in all the production wells are summed up and then the costs of water injection at the injection wells and water separation are subtracted. The reason why the cost of water separation is included in the objective cost function is to realise that the oil obtained in the production wells is actually in the two-phase oil-water mixture and still some effort has to be put in on the downstream production line to separate those two phases. In order to impose more dynamic behaviour of the optimiser with respect to time the discount factor is used which gives the weight to the profit obtained faster. Thanks to that we are sure that we use the optimal operational strategy for the reservoir management and we inject water not too rapidly and not too slowly at the same time. For example, one could inject the whole porous volume of the reservoir in the beginning of the oil production and fill the whole reservoir with water. This action would definitely increase the oil production but could also result in the higher cost for water injection and water separation as the water saturation in the reservoir would be very high. Consequently, even though oil production was performed very fast and reservoir was fully exploited, the Net Present Value would be relatively low. With this approach, one can conclude when it makes sense to stop injecting water, shut the production wells down and stop the production e.g. injection rates at some point can be very small and the net present value will not be increasing for the next time steps of the simulation.

6.3 Production Scenario

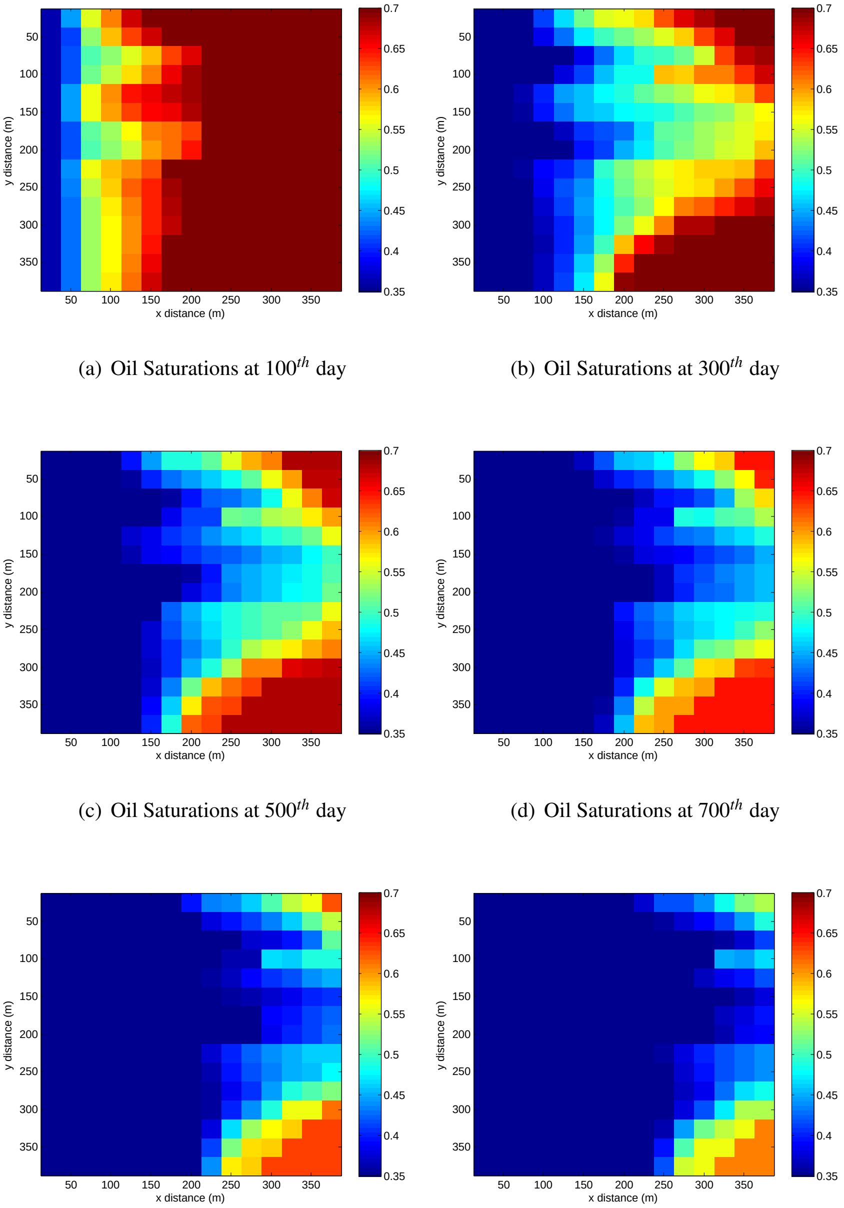
In order to test the simultaneous method for optimisation of the oil field, the numerical experiment with the following production scenario was carried out:
Simulation is performed in the spatially discretised reservoir into 15 x 15 grid blocks. Each of the grid blocks is 25 m wide, 25 m long and 20 m high, which means that the volume of each block is 12500 m^3 and the volume of the whole reservoir is 2812500 m^3. The porosity is 0.2 and constant within the reservoir which gives the total porous

volume (PV) of the reservoir equal to 5631250 m^3. The injection well is located at the left hand side of the reservoir and is divided into 15 segments equipped in one injector each. The permeability field is constructed, based on the standar case with to permeable streaks presented in [49] and [8]. The production well is located on the right hand side of the reservoir and is divided into 15 segments, where each of the segments contains one producer. In the production simulation water is injected by 15 injectors in order to displace the oil towards the producers sucking the mixture of oil and water. Sweeping oil from the reservoir and delivering it to production wells is considered as final stage of the upstream production and further processes are not analysed in the two phase displacement simulator. The discount rate factor related to NPV is set to zero since in many works e.g. in [9] it has been shown that the optimal injection rates are very sensitive to this parameter. The simulation is performed for 1500 days and the width of the time step Δt is 30 days which gives 100 discrete time steps. The decreasing oil saturations in the individual reservoir blocks at different stages of the production are presented in the figure 6.1.

Figures 6.2(a) and 6.2(b) show respectively the values of the bottom whole pressures at the 15 segments of production well and the injection rates of the 15 segments of injection well projected on the whole time horizon.

Figures 6.3(a) and 6.3(b) show respectively the plots of evolution of the net present value and injected porous volumes against time.

In can be clearly seen that the maximum of the net present value is attained at the 990^th day of production. Although the oil saturations are decreasing with the increasing time of production, the net present value does not increase. This means that the injected water front has hit the production well and produced after that period poorly oil-saturated two-phase mixture does not compensate for the prices of further water injection and water-oil separation. The physical sense of this result can be supported by the fact that in the 990^th day 1.08 PV was injected, which means that whole void space of the reservoir was flooded with water. In most cases simulation studies under waterflooding expect the total injection rate to be at least equal or grater than one porous volume of the reservoir. Since after 990^th day one cannot obesrve any increase in the NPV, which is in this case the exponent of the reservoir potential, according the the optimiser the wells are shot down after this day. It also shows the sense behind the studies done in this thesis and how simulation based optimisation techniques help to answer the very open question when to stop the production and how much profit expect from the reservoir.



(a) Oil Saturations at 100^{th} day

(b) Oil Saturations at 300^{th} day

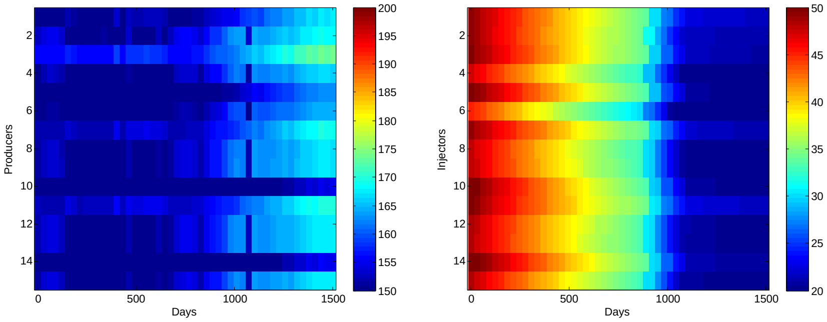
(c) Oil Saturations at 500^{th} day

(d) Oil Saturations at 700^{th} day

(e) Oil Saturations at 800^{th} day

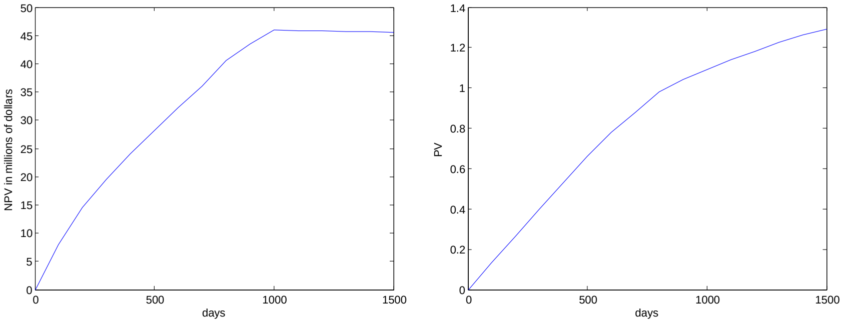
(f) Oil Saturations at 900^{th} day

Figure 6.1: Distribution of Oil Saturation at Different Stages of the Production



(a) BHPs vs Time (b) Water Injection Rates vs Time

Figure 6.2: Control Values vs Time



(a) Net Present Value in Million Dollars vs Days (b) Injected Porous Volumes vs Days

Figure 6.3: Net Present Value and Corresponding Injected Porous Volumes

Chapter 7

Conclusions and Future Work

The main focus in this thesis concerned the optimisation of oil production in the secondary recovery phase by the use of simultaneous method and interior point optimiser (IPOPT). This chapter is the final one and briefly summarises accomplished tasks undertaken in order to achieve the final goal.

- We have set up the open source solver IPOPT used for solving problems in this thesis. The tool was configured with the Microsoft Visual Studio Integrated Development Environment (MS VS IDE) and plugged in as a dynamic link library (dll) into several C++ implementations of optimisation problems investigated in this thesis. The process of setting up IPOPT in MSVS 2008 was thoroughly described in appendix B

- We have taught ourselves and demonstrated how to solve simple nonlinear programs with the use of Ipopt and C++ object oriented language in chapter 2.

- We have tested the simultaneous method by direct collocation, solving an exemplary optimal control problem of Van Der Pol oscilator and presented the results in chapter 3.

- We have implemented the two phase immiscible flow of black oil simulator in an isothermal reservoir with isotropic permeability field. The model is used to simulate the flow in the subsurface reservoir water-flooded in the secondary recovery phase. The differential algebraic equations were derived based on Darcy's law and mass conservation principle. The injection and production well models are based on the Peaceman well index for vertical wells in a non-square Cartesian grids. The control parameters of operation of injection and production wells are water injection rates and bottom whole pressures respectively.

- We have used the Corey relations to model the relative permeabilities of each phase and used the residual water and oil saturation to impose the boundaries on the saturation of each phase. We have neglected the pressure difference between two phases known as capilary pressure [50]. We have discretised the flow governing equations in space by using finite volume method (FVM) and Gauss' divergence theorem. Since this method requires the information about fluxes at the interfaces of the control volumes, we used two point flux approximation (TPFA) and the single-point upstream (SPU) scheme for computing the interface fluxes. The spatially discretised model was solved by applying backward Euler fully implicit (FIM) method.

- We have stated the problem of production optimisation of oil reservoir as a continuous Bolza' problem with water pressures and saturations as state variables and water injection rates and bottom whole pressure as controllable inputs. The optimisation goal was to increase the reservoir performance throughout the given production period in terms of net present value (NPV) as economic objective. In order to solve our continuous time problem we transcribed it into numerically traceable one by approximating states and controls as a piecewise polynomial functions on finite elements and represent it as a discrete nonlinear program.

- We optimised oil production by using simultaneous method in a nonlinear model predictive control framework. The optimisation is based on interior-point method in a linesearch framework with limited memory BFGS approximation for the Hessian of Lagrangian and exact analytical representation for the gradients. The reservoir simulator and routines for representing fully discrete nlp were written in C++ object oriented language in Microsoft Visual Studio Integrated Development Environment.

- The simultaneous method was tested on a particular scenario of oil production. Results clearly show that this method has a clear and merit potential for further investigation and solution of realistic cases.

Throughout accomplishing points listed above a few things have been observed which motivated the following comments on the project and suggestions for the future work.

Simultaneous method leads to a very large and sparse nonlinear problem. Since we performed our optimisation with the simulation scenario on a home PC computer, we were limited to a big extent by the machine memory resources. Consequently, we were trying to trade off between the realistic simulation scenario and guarantee that our equations are integrated properly by selecting reasonably big time steps and control volumes. This was done by realising the fact that the fluid velocities in the subsurface reservoir are of order of magnitude 10^{-5} and the optimisation problem is not smooth, so the information could be lost in case of too big step selection. Figures 6.1(a), 6.1(b), 6.1(c), 6.1(d) show that in some areas of the reservoir the transition between saturations in the neighbouring grid blocks was quite rapid, which might suggest selecting smaller steps in the future work. It doesn't, however, negate the fact that it was possible to solve the model with the simultaneous search and first order implicit scheme instead of using high-order ESDIRK methods with step size control. Another important thing worth paying attention to, is the way the model equation were discretised in space, which is by finite volume method (FVM). In finite volume method the whole physical space is represented as a collection of smaller control volumes filling the entire domain. The elements can have different dimensions as wells as there are no limitations on the whole grid structure, which easily allows the method to represent unstructed grids in higher dimension. The solution, however, is approximated by the cell average at the centre of the element which in our case defines saturations of each phase to be constant within each control volume. There exist other approaches that try to deal with this problem such as streamline based simulations, where one solves the equations for saturations changing along the randomly shaped lines representing the path travelled by the fluid particles in the reservoir, whereas pressures are found still with the FVM approach as they do not change that much with the spatial position in the reservoir [51]. The model implemented for simulation purposes in this work considers two phases, which are oil and water and is also known under term simple black oil simulator. However, many reservoir are build of multiple rocktypes, e.g. cap rock, which results in low pressure level and appearance of hydrocarbons which would simply represent the gas phase. In order to perform a simulation in such conditions the third, gas phase should be considered in the model with assumption that gas can dissolve in oil and oil vaporise in gas.

The simulation performed in this work was done on the two-dimensional grid. If one would like to consider a realistic simulator, he should realise himself that petroleum reservoir should be represented by three-dimensional grid of any shape, not necessarily rectangular, accounting also for flow in the vertical direction with the gravity force contribution.

Appendix A

Production Optimisation of the Two Phase Flow Reservoir in the Secondary Recovery Phase Using Simultaneous Method and Interior Point Optimiser

Production Optimisation of the Two Phase Flow Reservoir in Secondary Recovery Phase Using Simultaneous Method and Interior Point Optimiser.

Dariusz Lerch, Andrea Copolei, Carsten Völcker, Erling Halfdan Stenby, John Bagterp Jørgensen,

**Centre for Energy Resources Engineering, (CERE)
**Department of Informatics and Mathematical Modelling (IMM)
*Department of Chemical Enginering (Kemi)
*Technical University of Denmark (DTU), Kongens Lyngby 2800, Denmark (email: dal@kt.dtu.dk)

Abstract

In the world of increasing demand for energy and simultaneously decreasing number of newly found oil fields one can witness the interest for the simulation studies combined with optimisation methods in order to improve secondary recovery phase under water flooding techniques. Since the optimisation on the realistic reservoirs can be prohibitive when it comes to size and computation time a lot of attention was given to single-shooting methods combined with the use of adjoints for gradient computation which reduces the size of the problem. There exist, however other approaches optimisation of oil reservoirs as multiple-shooting or simultaneous method which have not been investigated that much by industrial and academic communities mainly because they do not eliminate states from the optimisation algorithm resulting in a problem of up to millions of optimisation variables.

In this paper we investigate the simultaneous approach on direct transcription for optimising oil production in the secondary recovery phase under water flooding. Results are encouraging and suggesting a merit potential of this approach for further investigation.

In the first section we explain the idea of smart well technology in the two phase flow reservoir. Then we introduce the process of reservoir management and picture the location of optimisation algorithms in it. Section III describes the two timescales involved in oil production and resulting challenges. Section IV points out the features of different optimisation methods, which have the potential for solving oil problem. In section V we present the mathematical formulation of the reservoir model and then we discretise and use it to formulate the optimal control problem in section VI. Section VII presents the particular instance of a production scenario with the corresponding results. Finally, the conclusions and suggestions for the future work are made in the last section.

Key Words: Optimisation, Reservoir Simulation, Discretisation, Two Phase Flow, Waterflooding

I. Introduction

Natural petroleum reservoirs are characterised by 2-phase flow of oil and water in the porous media (e.g. rocks). Conventional methods of extracting oil from those fields, which utilise high initial pressure obtained from natural drive, leave more than 70 % of oil in the reservoir. A promising decrease of these remained resources can be provided by smart wells applying water injections to sustain satisfactory pressure level in the reservoir throughout the whole process of oil production. Basically, to enhance secondary recovery of the remaining oil after drilling, water is injected at the injection wells of the down-hole pipes (figure 1.). This sustains the pressure in the reservoir and drives oil towards production wells. There are however, many factors contributing to the poor conventional secondary recovery methods e.g. strong surface tension, heterogeneity of the porous rock structure leading to change of permeability with position in the reservoir, or high oil viscosity. Therefore it is desired to take into account all these phenomena by implementing a realistic simulator of the 2-phase flow reservoir, which imposes the set of constraints on the state variables of optimisation problem. Then, thanks to optimal control, it is possible to adjust effectively injection rates, bottom whole pressures or other parameters to control the flow in every grid block of the reservoir and effectively navigate oil to the production wells so it does not remain in the porous media. The use of such a smart technology known also as smart fields, or closed loop optimisation, can be used for optimising the reservoir performance in terms of net present value of oil recovery or another economic objective.

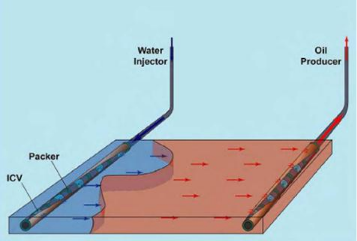


Figure 1 Schematic view of horizontal smart wells, [Jansen et al.,2008]

II. Reservoir Engineering

In order to maximise reservoir performance in terms of oil recovery or another economic objective, reservoir management process is carried out throughout the life cycle of the reservoir, which can be in order of years to decades. Reservoir management was firstly elaborated by Jansen et al. [2004] and its scheme is presented in the figure 2. In some other works this scheme might be presented in the slightly different way as the reservoir management can be enriched or missing some elements depending on the management strategy e.g. in case of an open loop reservoir management system models are not updated with data from the sensors through data assimilation algorithms and whole optimization is performed offline. What is more, some strategies distinguish between low and high order system models, which are responsible for uncertainty quantification. The top element in the fig. 2 represents the physical system constituting reservoir and well facilities. The central element refers to system models which consist of static (geological), dynamic (reservoir flow) and well bore flow models. The reason why multiple models are used is because each of them has some uncertain parameters which allow to determine uncertainty about the subsurface. The updated models through data assimilation and history matching technique with an uncertainty description give the support to the optimizer. On the right side of the figure, we have sensors, which are responsible for keeping the track of the processes that occur in the system. Sensors can be interpreted as physical devices taking measurements of the reservoir parameters, such as water or oil saturations and pressures but they can also be considered in more abstract manner as sources of information about the system variables e.g. interpreted well tests, time lap-seismics. On the left-hand side of the figure, one can find optimisation algorithms, which try to maximize the performance of the reservoir in terms of the given objective (e.g. net present value) based on the set of the constraints obtained from reservoir models. Since it is almost impossible to capture all important issues in the mathematical formulation, the optimizer and estimator elements will always include some human judgment. Very important element of the closed-loop reservoir management process are data assimilation algorithms (bottom of the figure), which obtain the data about the real world from the sensors and then update less realistic models with the more correct information. Data assimilation and model update is performed more frequently than off-line reservoir optimisation as models can easily get off the right track during simulation. As a result, most of reservoir management processes are understood as closed loop ones and their crucial elements are model based optimisation, decision making and model updating through the data assimilation techniques. One can realise himself that model based optimisation which is the main area of focus in this work, is a very significant element of the whole reservoir management process.

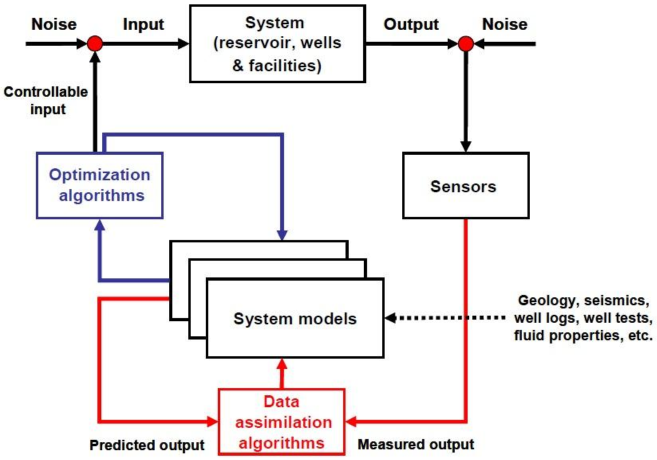

Noise
Input
System (reservoir, wells & facilities)
Output
Noise
Controllable input
Optimization algorithms
Sensors
System models
Geology, seismics, well logs, well tests, fluid properties, etc.
Data assimilation algorithms
Predicted output
Measured output

Figure 2 Reservoir Management Process, [Jansen et al. 2004]

III. Multi-scale (Upstream and Downstream) Optimisation

From physical point of view processes involved in oil production can be classified into upstream and downstream ones. Downstream processes refer to e.g. pipelines and export facilities whereas upstream processes are the ones happening in the reservoir e.g. subsurface flows. Those two types of processes differ from each other very distinctively when it comes to their timescales. In the upstream processes the velocity of the fluid can be very slow mainly due to some physical properties of the reservoir such as low permeability value or its size which can be up to two tens of kilometres. Hence it can take up to decades to navigate oil by injecting water towards production wells. In case of downstream parts of production timescales are much lower and can be in order of minutes or even seconds. In this work we focus on optimisation of upstream production where we model the two phase flow and run so called reservoir simulation. The simulation is based on mathematical models governed by partial differential equations (PDEs, governing equations) and is performed for a long time horizon, even up to decades of years. Consequently the optimisation of upstream part of oil production is run off-line whereas downstream part is mostly performed on-line. One of the most challenging aspects in closed loop reservoir engineering involves the combination of short-term production optimisation and long-term reservoir management. An open question is: what is the best way of implementing the found, optimal trajectory that was computed off-line into the daily performance of an oil field. Technically, daily valve setting are selected such that they result in instantaneous maximisation of oil production limited by constraints on the processing capacities of gas and water co-produced with the oil. Such settings are mostly determined with heuristics operating protocols, sometimes supported with off-line model based optimisation using sequential or quadratic programming to maximise instantaneous reservoir performance. What is more, a simple, frequent online feedback control is used for stabilising the flow rates and pressures in the processing facilities to separate oil, water and gas streams from the wells. It can be seen that there are a few control and optimisation processes going in parallel at different time scales. This kind of strategy involves a layer control structure where longer-term optimisation results provide set points and constraints for the instantaneous, short term optimisation, which then navigates and provides set points for field controllers. This modular approach, also known as multi-scale optimisation, has been widely used in the process industry and was proposed for reservoir management in Jansen et al. [2004] and has also been elaborated in Saputelli [2006].

IV. Optimisation Methods

Optimisation of oil production is stated as an optimal control problem constrained by the 2 phase flow model and boundaries on state and control variables. The model is non-linear and governed by partial differential equations (PDEs) for an each phase. The optimisation is performed in the nonlinear model predictive control framework where constrained dynamic optimisation problem is re-solved and re-implemented on regular sampling intervals; see Biegler et al. [2004]. This supports the advantages coming with combining the numerical optimal control solution with the feedback of the updated model through data assimilation techniques. There exist three main methods (single-shooting, multiple-shooting and simultaneous method) for solving NMPC dynamic optimisation problem and can be categorised based on how they discretise the continuous optimisation problem; see Ringset [2010]. So far, the most of attention from academic and industrial oil communities was given to single-shooting method which has been tried out in many works e.g. in Völcker et al. [2010], Capolei et al. [2011] or Suwartadi [2009] for optimization of oil reservoirs. The main reason for usage of single-shooting method (or sequential as optimisation is executed sequentially to numerical simulation for gradient computation) is because after reformulation it uses only manipulated variables (controls) as optimisation variables which reduces the optimisation space in the algorithm. Size reduction is a very attractive feature especially for oil problems since they have the tendency to be very big in the first place (up to millions of variables) so it is very convenient to eliminate the states from the optimisation algorithm and solve the smaller reformulation sequentially (SQP); see Li [1989] and Di Oiliveira [1995]. What is more, single-shooting is used with high order ESDRIK methods equipped with the error estimator which results not only in lower number of discretisation points but also ensures that the model equations are integrated properly. Contrary to single-shooting approach, the simultaneous method, which implementation for optimization of oil reservoirs is the main interest in this work, uses also a discretisation of the future process model variables as optimization variables. Thanks to that, the method offers the full advantage of an **open structure** after reformulation such as direct access to first and second order derivatives, many degrees of freedom and periodic boundary conditions. The transcribed nonlinear program by this method is however, much larger than by single-shooting. Nevertheless, it is very often the case that after direct transcription the problem is very sparse and structured so it is possible to define the sparsity pattern in an algorithmic way. Of course implementation of the sparsity pattern can be sometimes very time consuming but it offers a great trade-off when it comes to reduction of the problem size and other computational aspects. In simultaneous method the model is not solved at each iteration but a simultaneous search for both model solution and optimal point is carried out. In case of single shooting the model is solved (with an initial value solver) sequentially with reduced size optimisation problem. Consequently single shooting may be costly if evaluation of the problem functions is costly e.g. if implicit discretisation scheme must be applied, which is the case in optimisation of oil production.

V. Reservoir Model

The model for two-phase, completely immiscible flow comprises partial differential equations representing the mass conservation for water and oil phase of the following form:

$$\begin{aligned}\frac{\partial C_w}{\partial t} &= -\frac{\partial N_w}{\partial x} + Q_w \\ \frac{\partial C_o}{\partial t} &= -\frac{\partial N_o}{\partial x} + Q_o\end{aligned} \quad (1)$$

which state that the rate of change of water/oil concentration(C_w / C_o) with respect to time is equal to negative rate of change of flux(N_w / N_o) of each phase with respect to distance, enriched by the fee injection and production terms(Q_w / Q_o). The concentration of each phase is expressed as product of its density, saturation and porosity of the reservoir.

$$\begin{aligned}C_w &= \phi\rho_w(P_w)S_w \\ C_o &= \phi\rho_o(P_o)S_o\end{aligned} \quad (2)$$

The porosity is the fraction of the void space that can be occupied by the fluid and is assumed to be constant with respect to position in the reservoir. Saturations S_w, S_o are defined as the fraction of a volume filled by that phase. Since it is assumed that the petroleum reservoir contains only oil and water and two phases fill the available volume, saturations satisfy the following equation:

$$S_w + S_o = 1 \quad (3)$$

Densities of each phase are pressure dependent and are represented by the following equations of state:

$$\begin{aligned}\rho_w &= \rho_{w0} e^{c_w(P_w - P_{w0})} \\ \rho_o &= \rho_{o0} e^{c_o(P_o - P_{o0})}\end{aligned} \quad (4)$$

c_w , c_o are compressibilities of each fluid assumed to be constant in the given range of interest. $\rho_{w0} = \rho_w(P_{w0})$, $\rho_{o0} = \rho_o(P_{o0})$ are the densities at the reference pressures P_{w0} and P_{o0}

Mass is transported by convection and its velocity is obtained from Darcy's law that formulates the velocity through porous medium. This allows to express the fluxes as:

$$\begin{aligned}N_w &= \rho_w(P_w)u_w(P_w, S_w) \\ N_o &= \rho_o(P_o)u_o(P_o, S_o)\end{aligned} \quad (5)$$

u_w, u_o are linear velocities and are defined as the velocities that a conservative tracer would experience if taken by the fluid of the given phase through the porous formation. The reason why we use linear velocities and do not account for the fact that the medium is porous is because in our model we do not have any phenomenon influenced by the porosity such as formation damage or fines migration. This approach has been undertaken in many reservoir simulation works e.g. Völcker et al. [2009] or Aziz [1971] and yields:

$$\begin{aligned}u_w &= -k\frac{k_{rw}(S_w)}{\mu_w}\frac{\partial P_w}{\partial x} \\ u_o &= -k\frac{k_{ro}(S_o)}{\mu_o}\frac{\partial P_o}{\partial x}\end{aligned} \quad (6)$$

Where $k = k(x)$ denotes the absolute permeability of the porous medium, which is dependent only on the spatial position in the reservoir. $k_{rw} = k_{rw}(S_w)$ and $k_{ro} = k_{ro}(S_o)$ are relative permeabilities of each phase and are modelled by the Corey relations; see Völcker et al. [2009]. We also use residual oil saturation S_{or} and connate water saturation S_{wc} to impose the following boundaries on the saturation of each phase.

$$\begin{aligned}S_{or} &\le S_o \le 1 - S_{wc} \\ S_{wc} &\le S_w \le 1 - S_{or}\end{aligned} \quad (7)$$

Then the reduced saturations can be modelled as:

$$
\begin{aligned}
s_w &= \frac{S_w - S_{wc}}{1 - S_{wc} - S_{or}} \\
s_o &= \frac{S_o - S_{or}}{1 - S_{wc} - S_{or}}
\end{aligned}
\quad (8)
$$

Due to the surface tension and curvature in the interface between two phases the oil pressure tends to be higher than the water. The pressure difference between 2 phases is called the capillary pressure. This effect however, is very low in the highly permeable and porous media and is neglected in this model; see Berenblyum [2003]. The model is discretised in space by using finite volume method (FVM) and Gauss' divergence theorem ; see Völcker et al. [2012], which enables to consider the reservoir as a grid formed by blocks with constant dimensions. Each grid block is given an index i which indentifies its position in the reservoir. The absolute permeabilities k_i are assumed to be isotropic and constant within the grid block. The geological permeabilities at the interfaces between neighbouring grid blocks i and j are calculated as harmonic average of the absolute permeabilities of those blocks. What is more, the relative permeabilities $k_{rw,ij}$, $k_{ro,ij}$ at the interfaces between neighbouring grid blocks i and j are calculated using upstream weighting which requires the use of integer variables and results in solving mixed integer nonlinear program (MINMLP) having a highly combinatorial character which is more complex than a regular NLP.

VI. General Formulation and Time Discretisation

In optimisation problems involving process simulations, reformulating the problem and discretising it in time or space is always a challenge since it is often the case that a new discrete model should preserve such properties as e.g. conservation of mass, energy, or momentum. This is due to the fact that these properties are the initial outlet for the constraints definitions. As proposed by Völcker et al. [2012] mass preserving, spatially discretised reservoir model has the following form:

$$\frac{d}{dt} g(x(t)) = f(t, x(t)) \quad x(t_o) = x_o \, (9)$$

In which $x(t) \in \mathbb{R}^m$ represents the states (pressures and water saturations), $g(x(t)) \in \mathbb{R}^m$ are the properties conserved, whereas the right hand side function $f(t, x(t)) \in \mathbb{R}^m$ has the usual interpretation. Then with the use of eq. 9 we can formulate the water flooding problem as a continuous Bolza problem

$$
\begin{aligned}
&\min_{[x(t),u(t)]_{t_o}^{tf}} \int J(t, x(t), u(t)) dt \\
&s.t. \quad \frac{d}{dt} g(x(t)) = f(t, x(t)), \quad x(t_o) = x_o \\
&\qquad u_{\min} \le u(t) \le u_{\max} \\
&\qquad u_{\min}^{\Delta} \le \frac{d}{dt} u(t) \le u_{\max}^{\Delta}
\end{aligned}
\quad (10)
$$

The constraints $u_{\min}^{\Delta} \le \frac{d}{dt} u(t) \le u_{\max}^{\Delta}$ should be understood as movement ones and model the physical limitations on the controls (water injection rates and bottom whole pressures). In order to transcribe the infinite dimensional problem into numerically traceable one we use direct collocation method and fully discretise the optimal control problem by approximating the controls and states as piecewise polynomial functions on finite elements by applying implicit first order Runge Kutta method (Implicit Euler). This enables to represent to optimal control problem as a nonlinear program (NLP).

VII. Production Scenario

The numerical experiment of optimising production in oil reservoir was performed under following scenario: Simulation is run in the reservoir discretised into 15 x 15 grid blocks . Each of the grid blocks is 25 meters wide, 25 meters long and 15 meters high, the rock porosity is 0.2 and constant within the reservoir which gives the total porous volume equal to 562500 cubic meters. The time horizon was 1500 days was divided into 50 equal time steps of 30 days each. The injection well is located at the left hand side of the reservoir and is divided into 15 segments equipped in one injector each. The production well is located on the right hand side of the reservoir and is divided into 15 segments, where each of the segments contains one producer. In the production simulation water is injected by 15 injectors in order to displace the oil towards the producers sucking the mixture of oil and water. Delivering oil towards production wells is considered as final stage of the upstream production and further processes are not analysed in the 2-phase displacement simulator. The discount rate factor related to NPV is set to zero since in many works it has been shown that the optimal injection rates are very sensitive to this parameter, e.g. Capolei [2011]. The physical model data, as well as fluid properties and economic data that we used for this experiment can be found in Völcker [2012]. The water injection rates are constrained in such a way that no more than 2 porous volumes are injected throughout the total production time which gives minimum and maximum injection rates of single producer Q_{wmin} and Q_{wmax} equal to 0 and 50 cubic meters per day respectively. The lower and upper bounds on production well control parameters (bottom whole pressures) are set to 150 and 200 bars respectively. Such values are commonly used in this kind of simulations by the industrial community. The decreasing oil saturations within the reservoir at different stages of oil production are shown in the figures 3-7:

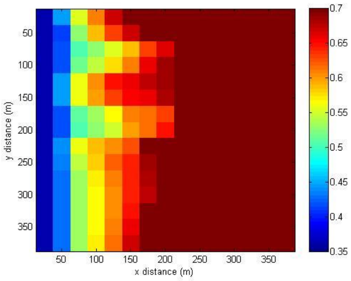


Figure 3 Oil Saturations after 100 Days

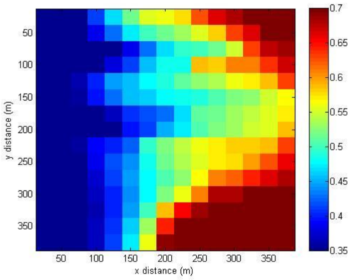


Figure 4 Oil Saturations after 300 days

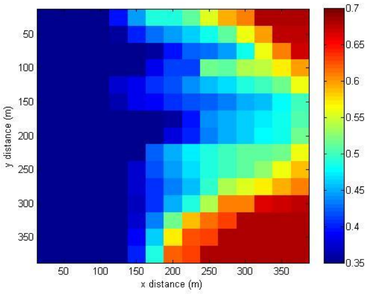


Figure 5 Oil Saturations after 500 days

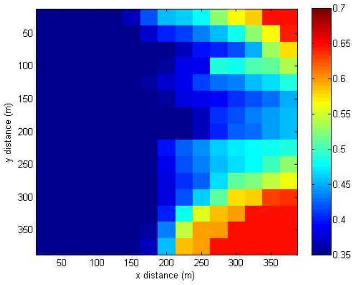


Figure 6 Oil Saturations after 700 days

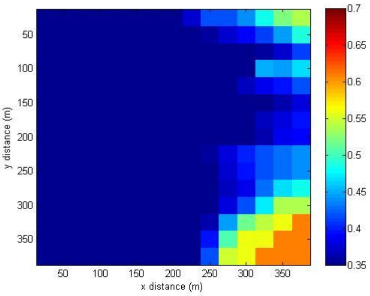


Figure 7 Oil Saturations after 900 days

The evolution of the net present value (NPV) and the injected porous volumes (PVs) are presented in the figures 8 and 9 respectively:

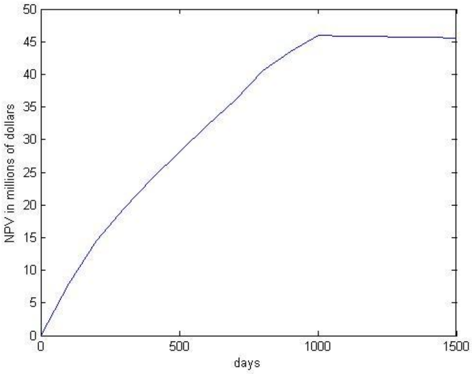


Figure 8 Evolution of Net Present Value throughout the production time

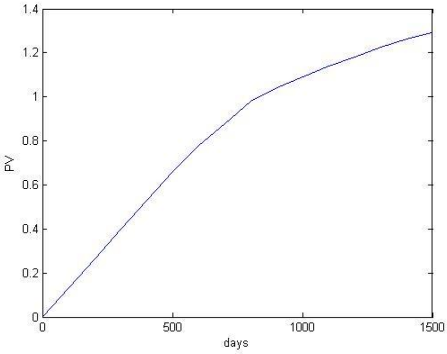


Figure 9 Evolution of injected porous volumes throughout the production

Figures 10 and 11 present the values of the manipulated controls (bottom whole pressures and water injection rates).

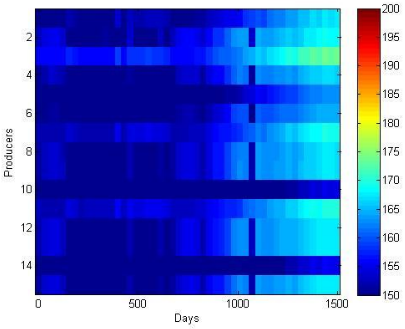


Figure 10 Bottom whole pressures in bars at the 15 producers throughout the production time

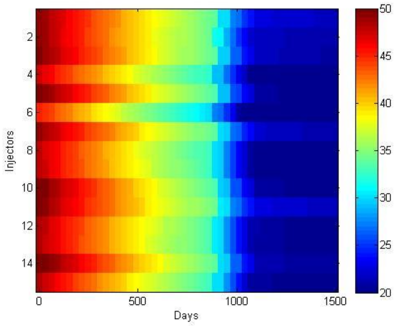


Figure 11 Water injection rates in cubic meters at the 15 injectors throughout the production time

Figures 3-7 clearly show how oil is swept out from the reservoir by the injected water throughout the production. Figure 8 and 9 distinctly demonstrate that the maximum npv (46 million dollars) was reached after injecting 1.08 pv at the 1000th day of the production, which means that the value of oil produced after this time did not compensate for the prices of water injection and water separation that also contribute to the economic potential of the reservoir. Consequently, according to the optimizer the wells should be shut down at 1000th day. This kind of simulation studies help to answer a very open question when to stop the production and how much profit expect from the reservoir.

VIII. Conclusions and Future Work

We have implemented the mathematical model of the two phase flow reservoir with the use of two point flux approximation (TPFA) and the single point upstream (SPU) scheme for computing the fluxes. The partial differential equations were derived by using the property of mass conservation and solved by discretising them in space and time by using finite volume method (FVM) and first order implicit Euler method respectively. The developed black oil simulator was applied in the nonlinear model predictive control (NMPC) framework combined with the simultaneous method for optimising the oil production in terms of the net present value (NPV) as the objective cost. As an optimisation algorithm, interior point method in the line search framework; see Wächter et al. [2006] and Schenk [2007], was chosen by using large scale optimisation package Ipopt; see Wächter et al. [2010]. The package distribution was plugged in as dynamic link library (DLL) to implementation of the reservoir model. The simulator and routines for representing fully discrete nlp were written in C++ object oriented language (OOL) in Microsoft Visual Studio Integrated Development Environment (MSVS IDE).

The found solution to the test problem clearly shows that the simultaneous method by direct collocation has a clear and merit potential for solving real case problem as the results obtained in this work make physical sense. This is very important as in this approach model is not solved in a conventional way, sequentially at each iteration but a simultaneous search for points satisfying model equations is done by the algorithm. Consequently, it could be the case that the model constraints are not satisfied if the algorithm terminates before converging. The relatively steep transition in the saturations between the neighbouring grid blocks is a suggestion for incorporating the mathematical term representing sweep efficiency in the objective cost function or reducing the size of the grid blocks. In the future work real life scenarios of productions for satisfyingly small time steps will be solved. This can be accomplished by deriving and implementing analytical expressions for second order derivatives constructing the Hessian of the Lagrangian matrix. At the current stage, this matrix is approximated by BFGS method which does not enable to represent it in a sparse way which means considering only non-zero elements and reducing the problem size.

Acknowledgements

Very grateful acknowledgements for contributing to this work are given to Associate Professors at Centre For Energy Resources Engineering (CERE): Wei Yan, Alexander Shapiro, as well as PhD students: Bjørn Maribo – Mogensen and Hao Yuan.

References

Aziz, K. and Settari, A. *Petroleum Reservoir Simulation.* London, first edition: Applied Science Publishers Ltd,, 1971.

Berenblyum, R.A., Shapiro, A.A., Jessen, K. and E.H. Stenby. "Black oil streamline simulator with capilary effects." *SPE Annual Technical Conference and Exhibition.* Denver, Colorado, 2003.

Biegler, L.T., Martinsen, F. and Foss, B. A. "Application of optimisation algorithms to nonlinear mpc." *Department of Chemical Engineering, Carnegie Mellon University, Department of Engineering Cybernetics, Trondheim, Norway*, 2004.

Byrd, R.H., Hribar, M.E. and Nocedal, J. "An interior point algorithm for large scale nonlinear programming." *Siam J. Optimisation* 9 (1999): 877-900.

Capolei, A., Völcker, C. and Frydendall, J. and Jørgensen, J.B. *Oil reservoir production using single-shooting and esdrik methods.* Kongens Lyngby, Denmark: Department of Informatics and Mathematical Modelling (IMM), Centre for Energy Resources Engineering (CERE), Technical University of Denmark (DTU), 2011.

Di Oiliveira, N. M. C. and Biegler L.T. "An extension of newton-type algorithms fon nonlinear process control." *Automatica 31(2)*, 1995: 281-286.

Jansen, D. R. and Brouwer, J.D. "Dynamic optimization of water flooding with smart wells using optimal control theory." *SPE Journal, 9(4)*, 2004: 391–402.

Jansen, J.D., Bosgra, O.H and Van Den Hof, P.M.J. "Model-based control of multiphase flow in a subusrface reservoirs." *Journal of Process Control 18* (Journal of Process Control 18), 2008: 846-855.

Li, W.C. and Biegler, L.T. "Multi-step, newton-type control strategies for constrained nonlinear processes." *Chem. Eng. Res. Des. 67*, 1989: 562-577.

Ringset, R., Imsland, L. and Foss, B. "On gradient computation in single-shooting nonlinear model predictive control." *Proceedings of the 9th Internation Symposium on Dynamics and Control of Process Systems.* Leuven, Belgium, 2010.

Saputelli, L., Nikolaou, M. and Economides, M.J. "Real-time reservoir management: a multi-scale adaptive optimisation and control approach." *Computational Geosciences*, 2006.

Schenk, O., Wächter, A. and Hagemann, M. "Combinatorial approaches to the solution of saddle point problems in large-scale parallel interior-point optimisation." *Comp. Opt. Applic. 36* (Comp. Opt. Applic. 36 (2007), 321--341), 2007: 321--341 .

Suwartadi, E., Krogstad, S. and Foss, B. "On state constraints of adjoint optimisation in oil reservoir water-flooding." *Reservoir Characterisation and Simulation Conference.* Abu Dhabi, UAE, 2009.

Völcker, C. *Production optimisation of oil reservoirs.* Kongens Lyngby, Denmark: Departmen for Informatics and Mathematical Modelling (IMM), Centre for Energy Resources Engineering (CERE), Technical University of Denmark (DTU), 2012.

Völcker, C., Jørgensen, J.B. and Stenby, E.H. *Oil reservoir production optimisation using optimal control.* Kongens Lyngby, Denmark: Department of Infomatics and Mathematical Modelling(IMM), Centre for Energy Resources Engineering (CERE), Technical University of Denmark (DTU), 2010.

Völcker, C., Jørgensen, J.B., Thomsen, P.G. and Stenby, E.H. "Simuation of the subsurface two-phase flow in an oil reservoir." *Proceedings of the European Control Conference.* Budapest, Hungary, August 23-26, 2009. 1221-1226.

Wächter, A. "A tutorial for downloading, installing, and using Ipopt." 2011.

Wächter, A. and Biegler, L.T. "On the implementation of an interior point-point filter line-search algorithm for large scale nonlinear programing." *Springer-Verlag.* 2006.

Appendix B

Ipopt in MSVS 2008

Ipopt in MSVS 2008:

A Tutorial for Setting up Ipopt Solver in Visual Studio 2008 IDE

Centre for Energy Resources Engineering(CERE),

Scientific Computing,

DTU Informatics(IMM),

Technical University of Denmark(DTU)

Dariusz Lerch, s091596

September 23, 2011

Contents

1 Abstract

Ipopt(Interior Point Optimiser) is an open source mathematical library for solving optimisation problems which can be utilised in Microsoft Visual Studio Integrated Development Environment(IDE). This document is a guide for installing Ipopt with its relevant third party components and setting it up in Visual Studio. At the beginning reader is given a brief introduction about Ipopt distribution and mathematical libraries that it applies. What is more, installation process and possible ways of getting Ipopt are given. Finally, the user is led step by step through compilation process of different types of Ipopt libraries and implementing them in a new Visual Studio project aimed at stating and solving the optimisation problem. This paper was written based on Introduction to Ipopt tutorial by Andreas Wächter1 and readme files2 that come together with COIN-Ipopt package of the same author.

2 Introduction

Ipopt is an open source package for nonlinear optimisation. It can compute and solve non-linear problems defined by an objective function $f(x)$, and vector functions of equality and inequality constraints. Basically, Ipopt calculations are based on interior point line search filter method, thanks to which local minimiser can be found. In order to solve optimisation problem in Ipopt, one needs to interface with its libraries by coding a problem in C, C++ programing languages or alternatively use AMPL, which is a special language for modelling mathematical problems.
Primarily, Ipopt was written in Fortran, however due to further elaboration, it was completely reimplemented and rewritten into C++ object oriented language. Although, originally Ipopt was used in Linux, it was written in a very generic way enabling the cross-platform usage. In other words, it is possible to use Ipopt solver in the Windows operating system and compile it against Microsoft Visual C++ compiler. In this tutorial, the reader will be led step by step how to install and build Ipopt against separately distributed third party code, so it can be applied and utilised in the Visual Studio 2008 IDE.

3 Ipopt Distribution

Note: IPOPTBASEDIR refers to the main directory where the content of downloaded Ipopt zip file is unpacked. In this directory ThirdParty, Build-

Tools and Ipopt directories are located.
In this section the structure of Ipopt distribution directory and its third party code will be described.

3.1 Ipopt Solver

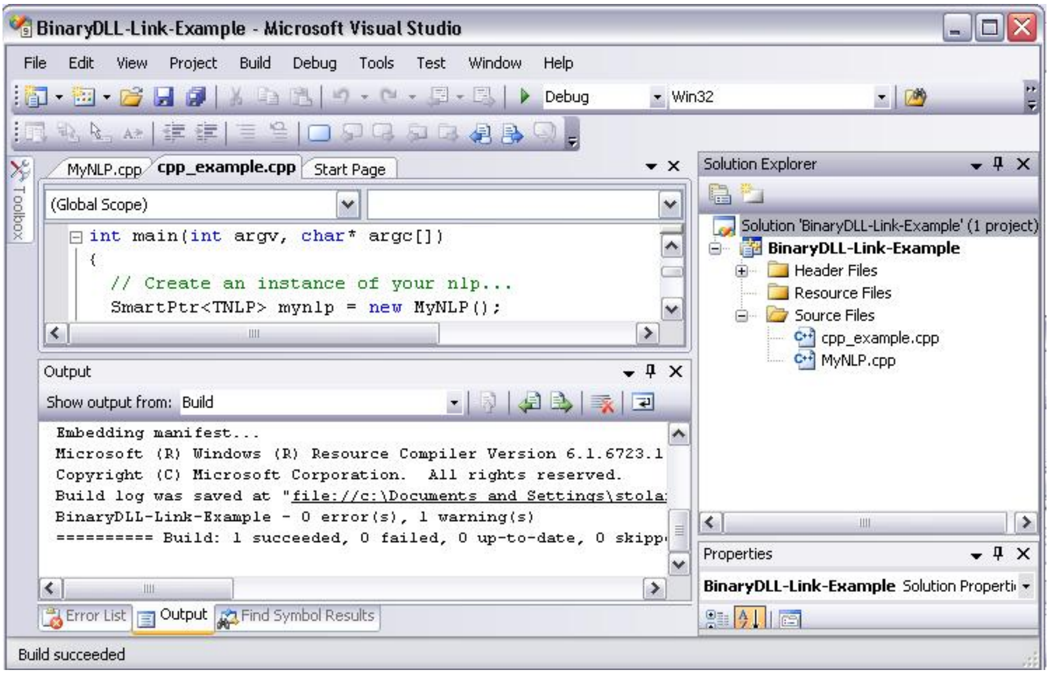
Ipopt is released as an open source code under the Eclipse Public License and is available from the COIN-OR Initiative web site as one of the COIN-OR Projects (https://projects.coin-or.org/Ipopt). Once Ipopt is downloaded on the user's computer, either as tarball or from subversion, it comes with a given directory structure that should not be violated as different Visual Studio projects keep their relative dependencies with respect to the downloaded base directory. In the IPOPTBASEDIR/Ipopt/MSVisualStudio/ there are three Microsoft Visual C++ solution directories. Although each of them applies Ipopt for solving optimisation problem, there exists slight difference in their application and suitability for specific user preferences. All three Visual Studio solutions and their functionalities are described in the following subsections.

3.1.1 Binary DLL Link Example Solution (BinaryDLL-Link-Example folder)

Once Binary DLL Link Example solution file is open, one can see only an example project in the project tree view(figure 1). It is so because this solution keeps a linker input to already precompiled Ipopt binary library that should be downloaded from ICOIN - OR alternative projects website.
This solution is recommended in case someone wants to use Ipopt with free distributed netlib libraries, without possibility to specify optimisation problem in AMPL language, since it is the least time consuming one. It only requires downloading the correct binary dynamic link library form the COIN-OR Ipopt project web site and placing it in the directory where program executable file is created. For further information please go to corresponding compilation subsection.

3.1.2 Visual C++/C DLL Solution(v8 folder)

This is a visual studio solution for compiling Ipopt with third party code and then linking it statically to the program specifying the optimisation problem. It is a static solution as C++ library projects are statically referenced and linked with each other in the compilation process.
If solution file is open, one can see that it contains projects corresponding



```
int main(int argv, char* argc[])
{
  // Create an instance of your nlp...
  SmartPtr<TNLP> mynlp = new MyNLP();
```

Show output from: Build

```
Embedding manifest...
Microsoft (R) Windows (R) Resource Compiler Version 6.1.6723.1
Copyright (C) Microsoft Corporation.  All rights reserved.
Build log was saved at "file://c:\Documents and Settings\stola
BinaryDLL-Link-Example - 0 error(s), 1 warning(s)
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipp
```

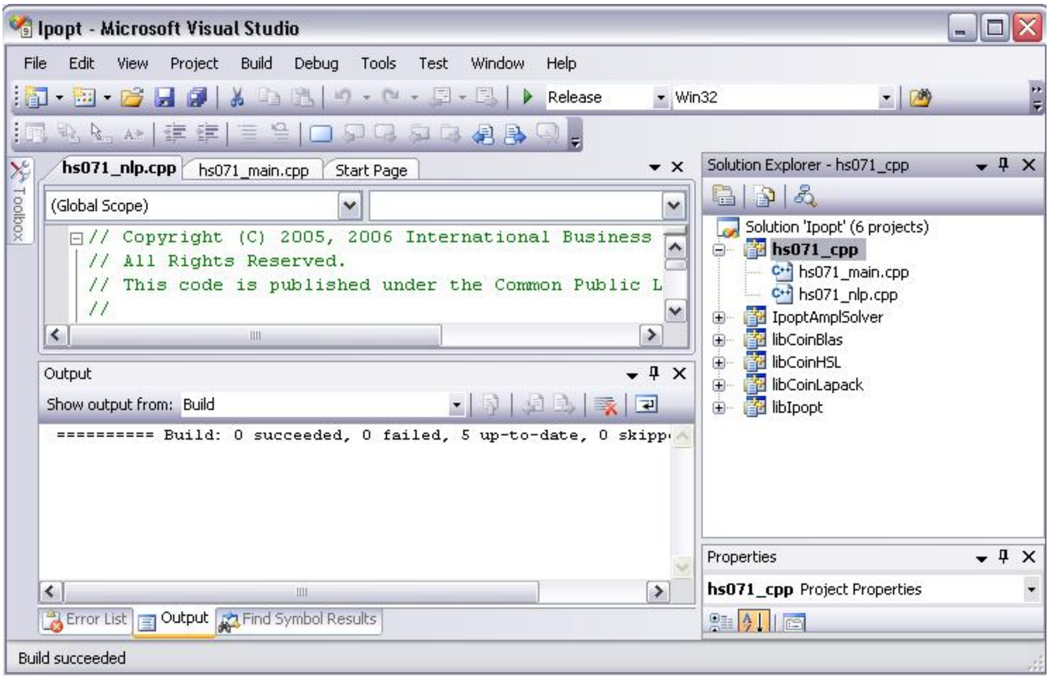
Figure 1: Visual Studio Binary DLL-Link Example Solution View

to third party code modules like: libCoinLapack, libCoinBlas, libCoinHSL. In order to get familiar with the project structure ,please have a look on figure 2 presenting solution tree view of the Visual Studio solution explorer. Since third party Visual C++ library projects are added as dependencies to Ipopt project, it is very important to build all referenced third party code projects before compiling Ipopt project. It is also possible to build the whole solution by right clicking on the solution icon (a top icon in the project solution explorer tree view) and selecting Build Solution option since the whole solution comes with already specified built order.Once this is done, it is possible to run hs071 example project, which solves simple optimisation problem. Please note that any of the third party projects keeps references to C and C++ files, although some third party components have been originally written in Fortran. It means that before compilation it is necessary to run Fortran to C compiler on some of third party modules. Otherwise Visual Studio compiler will report an error saying that some referenced C files are missing. A detailed information for downloading and preparing third party code for building against Ipopt library is given in the corresponding Compilation subsection.

3.1.3 Visual C++/Fortran DLL Solution(v8 - ifort folder)

This solution enables user to build Ipopt dynamic link library moslty directly from the Fortran files of the third party code. If the solution file is open, one can see that it contains Fortran and C++ projects that build the third party code modules (figure 3). These are: CoinBlas, CoinLapack, CoinMetis, CoinMumsF90, libhsl, libhsl-no-MA57. In order compile some of these projects,an Intel Fortran compiler should be configured with the Visual Studio IDE. One can say that this solution is a bit less time consuming to set up, if the Intel Fortran compiler is installed since Fortran files are directly compiled without the need of translating them into C ones as in the previous C/C++ solution. All projects keep references to their corresponding third party Fortran files, which should be located in the IPOPTBASEDIR\ThirdParty directory, so they can be compiled. In the solution tree view one can see that there are a few projects that build sparse symmetric solvers libraries, e.g: CoinMetis, libhsl, libhsl-no-57. It is possible to build the Ipopt library against all of them, or only specific ones, depending on the particular application of Ipopt package and user preferences. If there is any library project that user does not need, it can be simply unmarked from the Ipopt dependencies list and excluded form the compilation.

As in the previous case the Ipopt C++ library project keeps references (project dependencies) to the third party code projects so the utilised For-



```
// Copyright (C) 2005, 2006 International Business
// All Rights Reserved.
// This code is published under the Common Public L
//
```

Show output from: Build

```
========== Build: 0 succeeded, 0 failed, 5 up-to-date, 0 skipp
```

Build succeeded

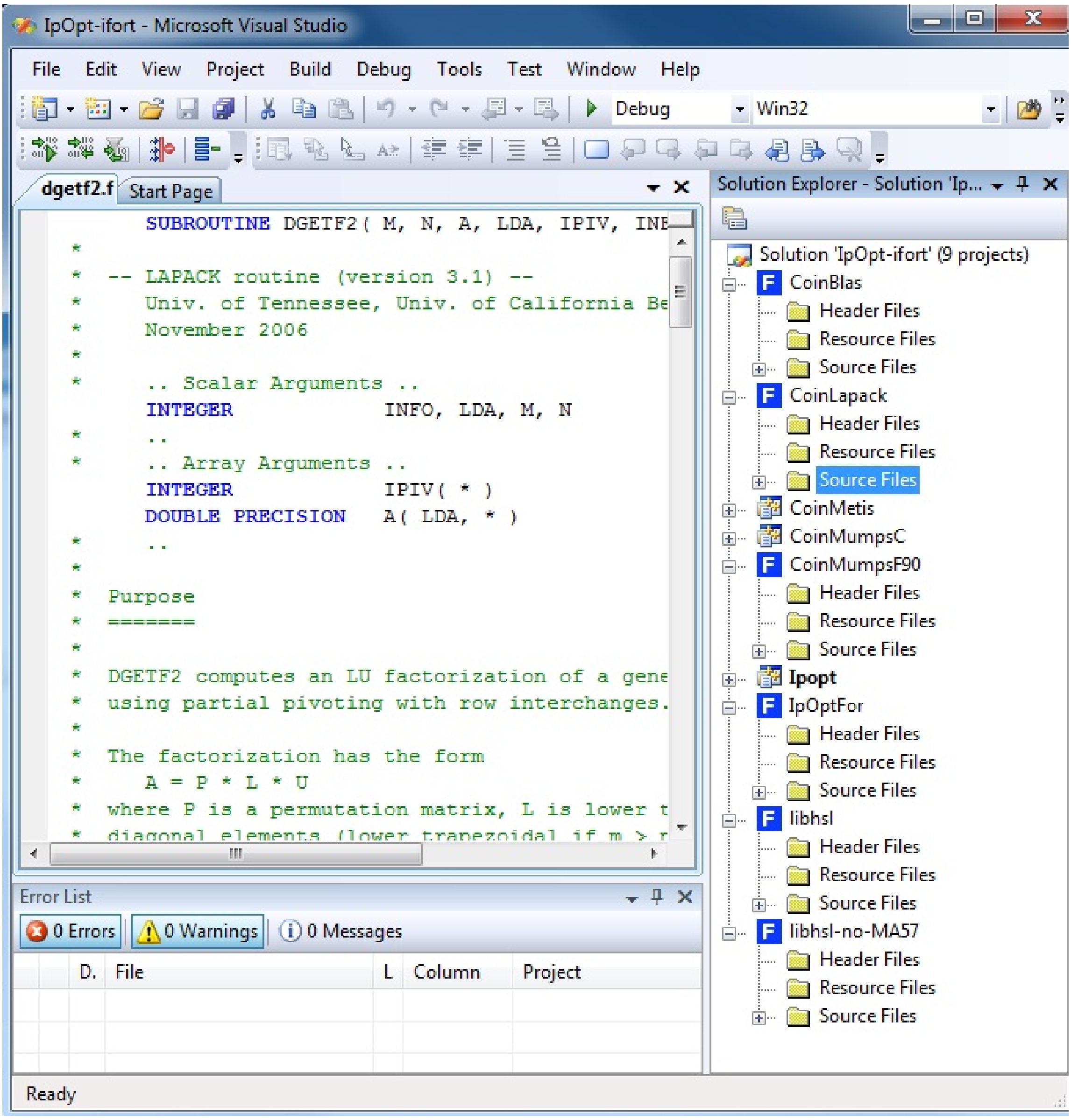
Figure 2: Visual C++/C DLL Solution View

tran project should be build in the first place. The order for building these projects is given in Ipopt Compilation section or can be checked by right clicking on the solution icon in the project tree view and selecting project dependencies option in the right click content menu.
Please note that it is also possible to build the whole solution by right clicking on the solution icon (a top icon in the project solution explorer tree view) and select build since the whole solution comes with already specified built order.
After building Ipopt, one will end up with Ipopt.dll file that can be added to any C++ project specifying the optimisation problem. This result library file should be roughly similar to one that is utilised by BinaryDLL-link-Example solution and can be downloaded from COIN-OR project web site, except for, in this case user has an option of specifying exactly which third party code he would like to have Ipopt build against. This is very important when one wants to implement optimised libraries of the third party code, e.g. MKL or MA57 which are not distributed by the COIN-OR initiative. In case of using standard not optimised , free distributed third party code libraries, it is recommended to download Ipopt dll from Coin - OR initiative web site and use Binary DLL Example Solution.

3.2 Third Party Code

Unfortunately Ipopt is not distributed with the so called third party code that it utilises for calculations. Before compiling Ipopt, third party code should be downloaded from the right online sources and placed in the Third-Party directory in the IPOPTBASEDIR folder. In other words, it is user responsibility to download the third party code. Exact steps on how to do this are given in Download and Installation section. Third party code determines functionality of Ipopt and consequently user does not need to download and set up all of it but only these libraries that correspond to the desired problems. Below different components of the third party code with their functionality are described:

* BLAS (Basic Linear Algebra) - it is possible to obtain a source code of this library directly from www.netlib.org. There exists however, possibility of getting already precompiled and optimized version of this library. Since it is one of the basic and sufficient ones, an optimized version of it can result in big speed up of computation time. An example of optimizaed BLAS library is MKL(Math Kernel Library) released by hardware vendor Intel.
  Visual Studio projects of Ipopt also posses settings for compilation



```
      SUBROUTINE DGETF2( M, N, A, LDA, IPIV, INF
*
*  -- LAPACK routine (version 3.1) --
*     Univ. of Tennessee, Univ. of California Be
*     November 2006
*
*     .. Scalar Arguments ..
      INTEGER            INFO, LDA, M, N
*     ..
*     .. Array Arguments ..
      INTEGER            IPIV( * )
      DOUBLE PRECISION   A( LDA, * )
*     ..
*
*  Purpose
*  =======
*
*  DGETF2 computes an LU factorization of a gene
*  using partial pivoting with row interchanges.
*
*  The factorization has the form
*     A = P * L * U
*  where P is a permutation matrix, L is lower t
*  diagonal elements (lower trapezoidal if m > r
```

0 Errors | 0 Warnings | 0 Messages

Figure 3: Visual Studio Fortran/C++ DLL Solution View

against MKL. A detailed explanation on utilising these settings is given in the Compilation section.
There exist also other versions of BLAS distributed by processor vendors e.g.

  * ACML(AMD Core Math library) by AMD
  * ESSL (Engineering Scientific Subroutine Library) by IBM

* LAPACK (Linear Algebra Package) - this component is only required by Ipopt if the user tries to compute solution with quasi-Newton options. In this case, if LAPACK is missing, the software will report an error. In case of calling LAPACK package from Ipopt, it is much more efficient to do so in Windows operating system as LAPACK libraries are mostly distributed as not optimised for the Linux operating system.

* HSL(Harwell Subroutine Library) is a mathematical Fortran library for a very large scale scientific computation. Consequently Ipopt does not need to implement all of it but only some of its routines depending on the tasks it should solve. For example Ipopt needs to call and obtain solution from sparse symmetric solver. In order to do that it needs to be built against one of the following solvers contained in HSL.

  * MA 27
  * MA 57

  It is also possible to obtain sparse symmetric solvers from other vendors:

  * MUMPS (Multifrontal Massively Pralllel Sparse Direct Solver)
  * PARADISO(The Parallel Direct Solver)

* ASL - this component is completely optional and does not contribute to calculations in any way. ASL contains set of solver interfaces to the AMPL language which can be used for specifying constrained optimisation problem. This code is useful only when the user decides to interface with Ipopt not through C++ program but script in AMPL language. Using AMPL language is much easier than C++ coding since it is a language developed specifically for defining mathematical problems. Applying AMPL however, results in longer calculation time as it enriches the program in another layer. In order to get further information about AMPL please go to the following web site:
  http://www.ampl.com/

3.3 Additional Remarks

All mentioned sparse symmetric solvers can be loaded into Ipopt as dynamic link libraries. Once it is done the user will be able to call different solvers for solving optimisation problems with the most suitable settings for the given problem in terms of efficency and accuracy of the computations.(e.g. some solvers can converge faster than the others, because they are optimised). Please note that some of the third party code components are written in Fortran. Consequently in order to build and load them as static or dynamic link libraries in Visual Studio, one will need either a Fortran to C compiler or an Intel Fortran compiler configured for Visual Studio. More detailed information about getting and building third party code in Visual Studio will be given in the Compilation section.

4 Download and Installation

This section explains in details how to obtain Ipopt and third party code. What is more, it instructs the user where to place single modules, so they can be automatically utilized by Visual Studio projects that come with the package.

4.1 Getting Ipopt

There are two possible ways of getting Ipopt from ICOIN-OR Ipopt project web site. These are the following:

* as a tarball

* through subversion

4.1.1 Getting Ipopt as a Tarball

In order to download Ipopt as a tarball please go to the following URL in your web browser:
-http://www.coin-or.org/download/source/Ipopt
and download the Ipopt zip package Ipopt-x.y.z.zip, where x.y.z is the version number. The latest stable version number is 3.9.1.

4.1.2 Getting Ipopt via Subversion

In order to get Ipopt via subversion, a subversion client must be installed on the user's computer system. A good example of subversion client is Tortois-

eSVN which is an open source software and can be downloaded from:
http://tortoisesvn.tigris.org/
Tortoise software is added to the Windows shell extension after installation so it is possible to right click on the folder where Ipopt should be located and in the mouse right click content menu select svn checkout option as shown on the figure 4. This action will open a new Tortoise window(figure 4) where user can specify the URL address of the repository and path of checkout directory on a computer drive where Ipopt should be located. In order to connect to latest Ipopt project repository please type there the following URL repository adress:
https://projects.coin-or.org/Ipopt/browser/stable/3.9
and click Ok. This will connect the user to the Ipopt repository and download the distribution into specified folder.

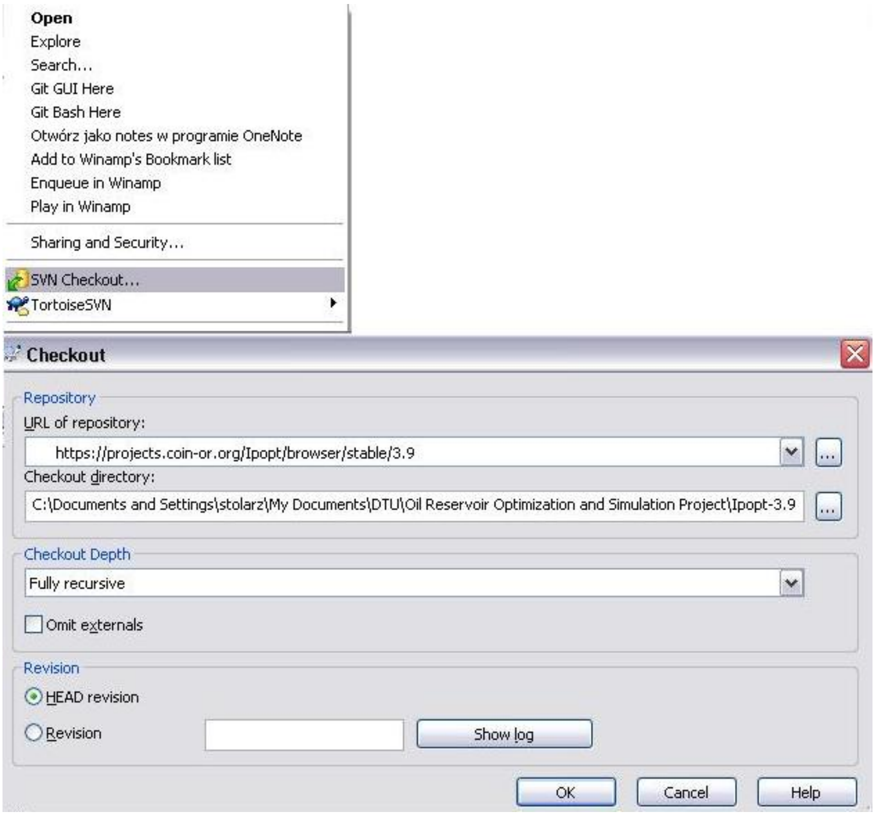
4.2 Getting Third Party Code

Third Party Code is an independent set of libraries that is released on different licence conditions than Ipopt routines. Consequently, it cannot be obtained from CPOIN-OR initiative web site, but has to be downloaded directly from other vendors sites. In this section a few examples explaining how to obtain third party code will be presented. It is also essential to place the third party code in the right directory which is:
IPOPTBASEDIR\ThirdParty\*
where * should correspond to the given name of the third party code library, e.g:

- IPOPTBASEDIR\ThirdParty\LAPACK
- IPOPTBASEDIR\ThirdParty\BLAS

4.2.1 Getting BLAS

In order to obtain BLAS library for Windows operating system one should either use wget or donwload the code directly. To install wget, it is necessary to work in Unix/Linux like environment, for example Cygwin which is a Linux emulator for windows. Having wget installed enables to run get.Blas file which is located in the IPOPTHOMEDIR\ThirdParty\BLAS which downloads the Blas code to the desired directory. To download the code directly from the source page, please go to the following URL:
http://www.netlib.org/blas/blas.tgz
and unpack the package in the:



Repository

URL of repository: https://projects.coin-or.org/Ipopt/browser/stable/3.9

Checkout directory: C:\Documents and Settings\stolarz\My Documents\DTU\Oil Reservoir Optimization and Simulation Project\Ipopt-3.9

Checkout Depth: Fully recursive

□ Omit externals

Revision

● HEAD revision

○ Revision

Figure 4: Tortoise Content Menu and Checkout Window

IPOPTBASEDIR\ThirdParty\BLAS
so that all the BLAS Fortran files are located there.

4.2.2 Getting ASL

Although ASL is not necessary for compiling Ipopt, it is very convinient to build Ipopt against this library especially if the user is not very familiar with C++ programing language. In order to download ASL code please go the the following URL adress:
http://netlib.sandia.gov/ampl/index.html,
click on the solvers(tar) link and unpack this package in such a way that the folder solvers is located in the following directory:
IPOPTBASEDIR\ThirdParty\ASL.
The solvers folder contains library of routines for interfacing different optimisation solvers with AMPL modeling language. For further information about solvers package please have a look on solvers readme file under following URL:
http://www.netlib.org/ampl/solvers/README

4.2.3 Getting HSL

Information on the Harwell Subroutine Library (HSL) is available at
http://www.cse.clrc.ac.uk/nag/hsl/
Some of the required HSL routines are available in the HSL Archive which can be obtained for free for non-commercial purposes. It is user responsibility to make sure that he utilises these routines according to the licence conditions. Here are listed sparse symmetric linear solvers with their licence conditions:

- MA57 this solver is distributed as part of commercial copy of the HSL library. It can also be obtained by academic who wants to use the "HSL 2007 for Researchers" library

- MA27 - this solver is distributed as part of HSL Archive.

- MC19 - this solver is distributed as part of HSL Archive.

In order to download mentioned HSL Archive subroutines (like ma27 and mc19) one needs to register at the HSL Archive website:
http://hsl.rl.ac.uk/archive/hslarchive.html
or the HSL 2007 for Researchers website:
http://hsl.rl.ac.uk/hsl2007/hsl20074researchers.html
After successful registration process, it is possible to log in and go to the page

that lists all the available HSL packages for download and click on the name of the desired routine (like MA27). This redirects the user to the download page with the button "Download Package (comments removed)". Here the user should click on that button and leave the precision choice at default setting which is "Double Precion". This action brings up a web page which contains the code for the subroutine as text. All that has to be done now, is to save this page into a corresponding Fortran file, e.g ma27ad.f in case of MA27 routine. Alternatively if the web browser does not provide "Save page to" option, one can simply select all the content of the web page and then copy and paste it into mentioned Fortran file. The same steps should be performed in case of downloading MC19 routine which should be saved into mc19ad.f file.

When downloading MA57 routine, it is essential to download not only the source code of the routine but also all the dependencies. This can be done by clicking on "Download HSL dependencies" on the ma57 download page which gives everything packed into a one file. Below the dependencies for the MA57 routine are listed:

- fd15ad.f
- mc21ad.f
- mc34ad.f
- mc47ad.f
- mc59ad.f
- mc64ad.f

In case of using METIS routine a metis.f dummy file is required. However it is strongly recommended to use METIS with configure flag to specify precompiled Metis library. For further information about using METIS please have a look at the ThirdParty/Metis directory.

It is not necessary to download any dependencies for MA27 and MC19.

4.2.4 Getting LAPACK

In order to download LAPACK please go to official LAPACK web site:
http://www.netlib.org/lapack/
and click on one of the release links, e.g. LAPACK version 3.1 or LAPACK version 3.2.2. LAPACK content should be unpacked into the:
IPOPTBASEDIR\ThirdParty\LAPACK\LAPACK directory

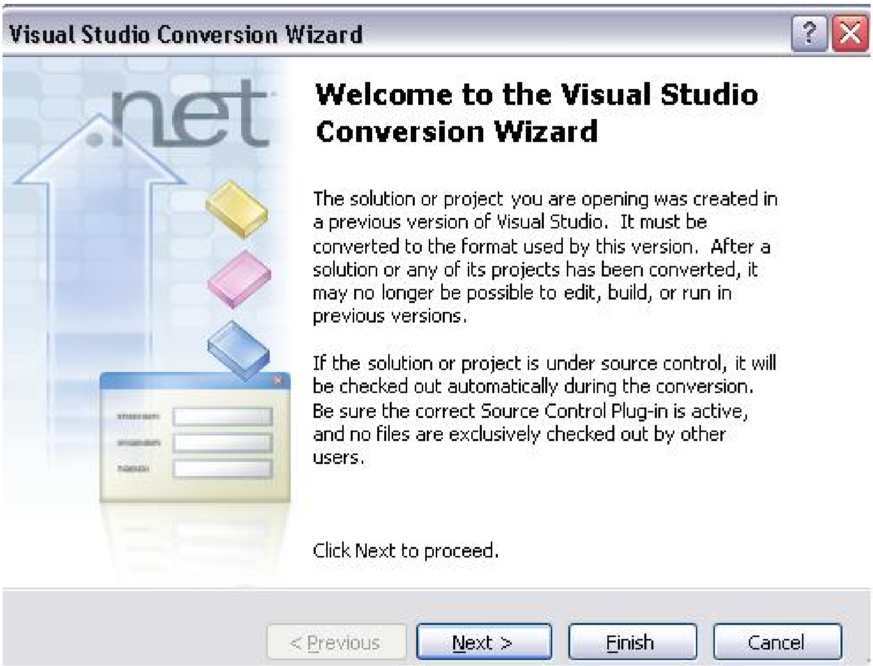
NOTE: The LAPACK source files are expected to be in the subdirectories:
IPOPTBASEDIR\ThirdParty\LAPACK\LAPACK\SRC
IPOPTBASEDIR\ThirdParty\LAPACK\LAPACK\INSTALL

5 Ipopt Compilation in Microsoft Visual Studio

Remarks: The information gathered in this section is based on and sums up Wächter at 2 in the IPOPTBASEDIR/Ipopt/MSVisualStudio/ directory there are three Microsoft Visual Studio Solution folders. This section describes step by step how to compile and build each of them. Before starting any of the steps mentioned in this section, it is strongly recommended that the required third party code components have already been downloaded and placed in the instructed directories. In order to get information on downloading third party code please go to Download and Installation section or read INSTALL.* files in the third party code directories, e.g IPOPT-BASEDIR\ThirdParty\BLAS\INSTALL.BLAS.
All solution are Microsoft Visual Studio 2005 ones but they are completely transferable to Visual Studio 2008. One tries to open any of the solutions with Microsoft Visual Studio 2008 a conversion wizard window will pop up(figure 5) and take the user step by step through conversion process.

5.1 Compilation of Binary DLL Link Example Solution(BinaryDLL-Link-Example folder)

Note: DLLBASEDIR refers to the main directory where downloaded dll package was unpacked.
To compile this solution, one needs to obtain the Ipopt dll package and unpack its content into the IPOPTBASEDIR\Ipopt\MSVisualStudio\BinaryDLL-Link-Example directory so that the "include\coin" folder of the binary dll package will be located in BinaryDLL-Link-Example solution folder. The include\coin folder is specified as additional include directory for the solution and contains header files defining implementation cpp files that were compiled into Ipopt dll. It is also possible to unpack dll package according to the user preferences but then, one should make sure that the right path to the include\Coin folder is specified as additional include directories of the solution.
Binary dll packages are available at the COIN-OR Ipopt project web site



Welcome to the Visual Studio Conversion Wizard

The solution or project you are opening was created in a previous version of Visual Studio. It must be converted to the format used by this version. After a solution or any of its projects has been converted, it may no longer be possible to edit, build, or run in previous versions.

If the solution or project is under source control, it will be checked out automatically during the conversion. Be sure the correct Source Control Plug-in is active, and no files are exclusively checked out by other users.

Click Next to proceed.

Figure 5: Visual Studio Conversion Wizard

under following URL adress:
http://www.coin-or.org/download/binary/Ipopt/
To compile and run the solution, correct version(debug or release depending on the build settings) of the Ipopt dll should be copied to the directory where the compiled executable is located. E.g. In case of working on Windows operating system on 32 bit machine please copy the files:
Ipopt*.dll and Ipopt.lib from:
IPOPTBASEDIR\Ipopt\MSVisualStudio\BinaryDLL-Link-Example\lib\win32\degug\to:
IPOPTBASEDIR\Ipopt\MSVisualStudio\BinaryDLL-Link-Example\Debug\
as this is a default location for the executable file.
It is also possible to locate Ipopt*.lib and dll file according to user preferences, as long as, he makes sure that the right additional dependency directories and additional dependencies for the linker input were specified.

Additional Remarks In DLLBASEDIR\lib\win*, where * reprsents the Windows operating system architecture (e.g 32), there are debug and release folders which hold versions of the Ipopt dll and lib files respectively. It is very important that user uses the dll and lib files of the appropriate version for the build settings since debug and release versions are not binary compatible. This is mainly because Microsoft Visual C++ compiler treats standard library string arguments of methods in the Ipopt C++ interface in a slightly differently way. Consequently, using release dll in debug code and vice versa will lead to stack corruption, runtime errors, or other hard-to-explain crashes.
DLLBASEDIR\lib\win* also contains MKL folder where dll , linked against the optimized Intel MKL Blas and Lapack libraries, is located. One can benefit from these optimised libraries on multi-core processors. In case of single core machines they do not appear to have much effect or can even lead to slightly decreased performance.
When downloading dll package, it is essential to download one for the appropriate architecture and operating system. It is also important that the version number of the binary dll file corresponds to the version number of the Ipopt distribution, e.g:
In case of having version 3.9.1 of Ipopt distribution on Windows 32 bit operating system please download Ipopt-3.9.1-win32-win64-dll.7z package from:
http://www.coin-or.org/download/binary/Ipopt/

5.2 Compilation of Visual C++/C DLL Solution(v8 folder)

This solution compiles Ipopt library project against third party code C++ projects . These are HSL, LAPACK, BLAS and ASL.Please note that ASL has only to be compiled if, someone wants to interface Ipopt through AMPL language. In case of coding problems in C++ , ASL project does not have to be compiled and can be omitted in the build process.
Since some of third party code is written in Fortran, and Visual Studio keeps references to C files it is also necessary to obtain the f2c Fortran to C compiler. Below are presented steps that user should go through to build the solution:

1. From netlib, download the file:
   http://www.netlib.org/f2c/libf2c.zip
   and extract it in:
   IPOPTBASEDIR\Ipopt\MSVisualStudio\v8
   (do not specify a "libf2c" subdirectory in the path) so that one will see the file:
   BASDIR\Ipopt\MSVisualStudio\v8\libf2c\makefile.vc

2. Open a Visual Studio DOS prompt and go into the directory:
   cd BASDIR\Ipopt\MSVisualStudio\v8\libf2c\
   Here one should type:
   nmake -f makefile.vc all
   If there is a problem related to the comptry.bat file, please edit the file makefile.vc and just delete the one occurance of the word "comptry.bat".
   If an error stating that unistd.h is not found occurs, edit the file 'makefile.vc' and add "-DNO_ISATTY" to CFLAGS.

3. Download the Fortran to C f2c.exe executable, place it somewhere on the hard drive and add its location to the Windows path variable. Adding f2c.exe to the path variable will enable user to access Fortran to C compiler from any location on the hard drive.
   Fortran to C compiler can be obtained from the following link:
   http://www.netlib.org/f2c/mswin/

4. Please note that this step is only necessary if it has not been done, when going through Download and Installation section in this document
   Download the Blas, Lapack, and HSL source code into (for further information about this step please go to Download and Installation

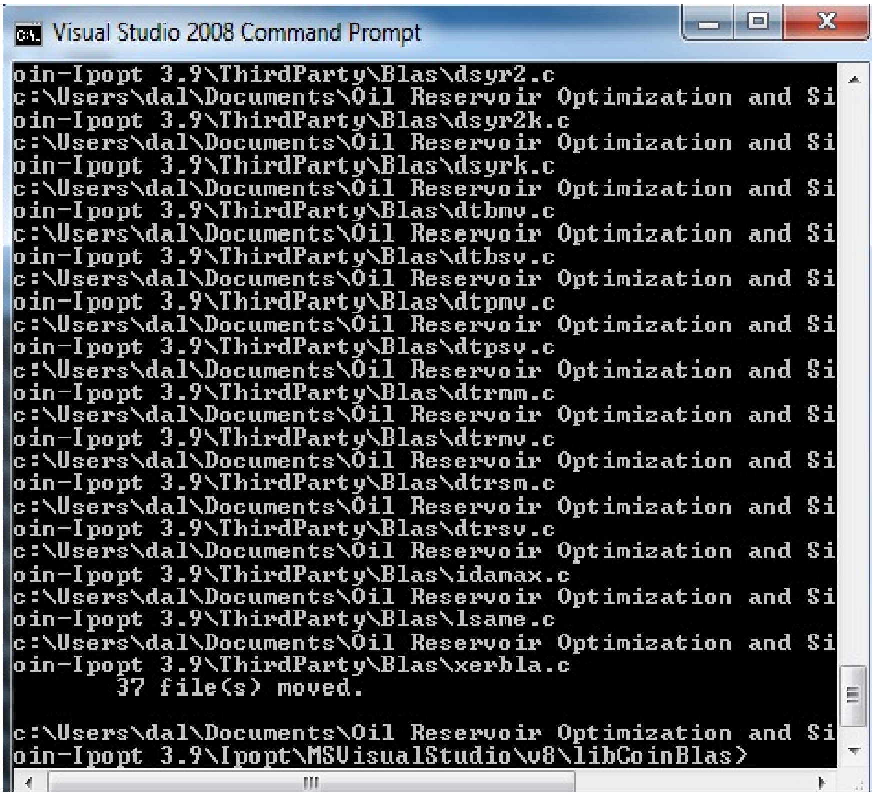
section in this document or read the INSTALL.* files in the appropriate subdirectories of ThirdParty folder):
BASDIR\ThirdParty

5. In a Microsoft Visual Studio Console window, go to the directory
BASDIR\Ipopt\MSVisualStudio\char92v8\libCoinBlas
and run the batch file:
convert_blas.bat
This runs the Fortran to C f2c compiler and generates new C files of the BLAS library that was originally developed in Fortran. After running bat file, one should get the following output in the console window (figure 6)

6. Repeat the previous step in the following directories:
BASDIR\Ipopt\MSVisualStudio\v8\libCoinLapack
BASDIR\Ipopt\MSVisualStudio\v8\libCoinHSL
using the convert_*.bat files that are located there, where * corresponds to name of the third party code module, e.g. HSL. The outputs of the bat files in console window for translating HSL and LAPACK from Fortran to C are presented in the figures 7 and 8 respectively. If everything went well, user should obtain similar output on his console.

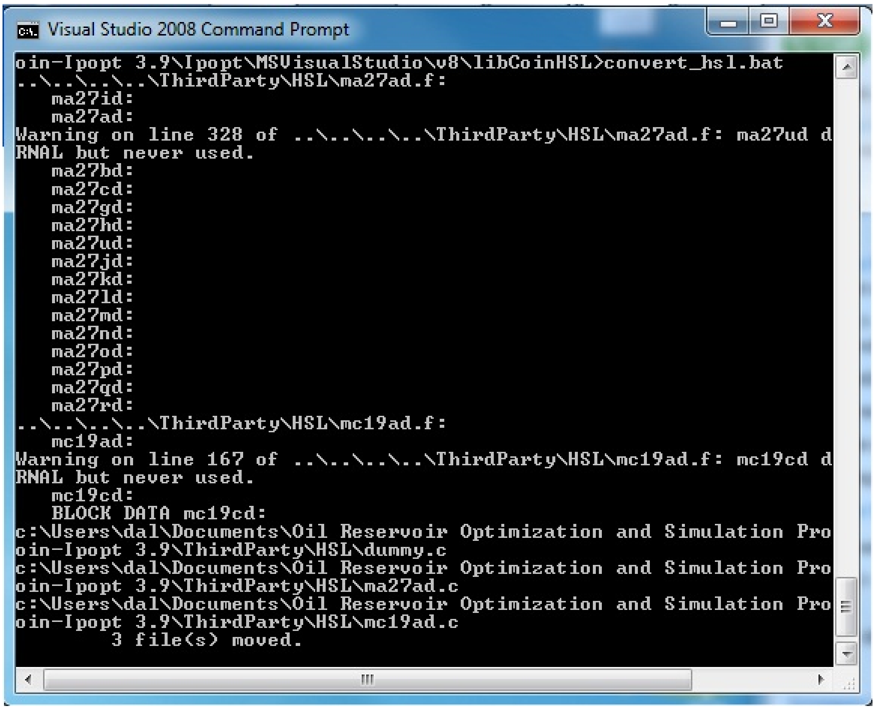
7. This step compiles ipopt.exe AMPL solver executable and is only required if one wants to interface with Ipopt through AMPL language for specifying optimisation problems. In order to do it please follow the steps below:
Download the ASL code into the directory(for information about downloading the ASL code please see Getting ASL subsubsection in the Download and Installation section of this document or see INSTALL.ASL file in the directory given below):
BASDIR\ThirdParty\ASL
Then, in a Visual Studio DOS prompt in that directory, type:
copy details.c0 details.c
nmake -f makefile.vc
If comptry.bat reports errors, delete its occurrence in the makefile.vc. Calling makefile compiles and builds ipopt.exe AMPL solver executable. If this process is executed successfully, one should get the console output as presented on the figure 9.

Now it is possible to open the solution file:
BASDIR\Ipopt\MSVisualStudio\v8\Ipopt.sln



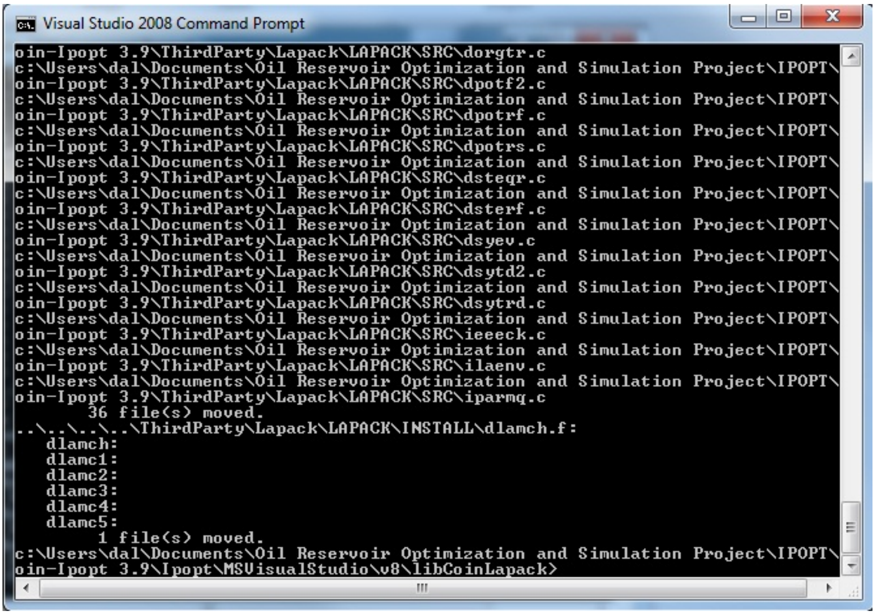
```
oin-Ipopt 3.9\ThirdParty\Blas\dsyr2.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dsyr2k.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dsyrk.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtbmv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtbsv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtpmv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtpsv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtrmm.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtrmv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtrsm.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\dtrsv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\idamax.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\lsame.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\ThirdParty\Blas\xerbla.c
       37 file(s) moved.

c:\Users\dal\Documents\Oil Reservoir Optimization and Si
oin-Ipopt 3.9\Ipopt\MSVisualStudio\v8\libCoinBlas>
```

Figure 6: Convert Blas Bat File Console Output



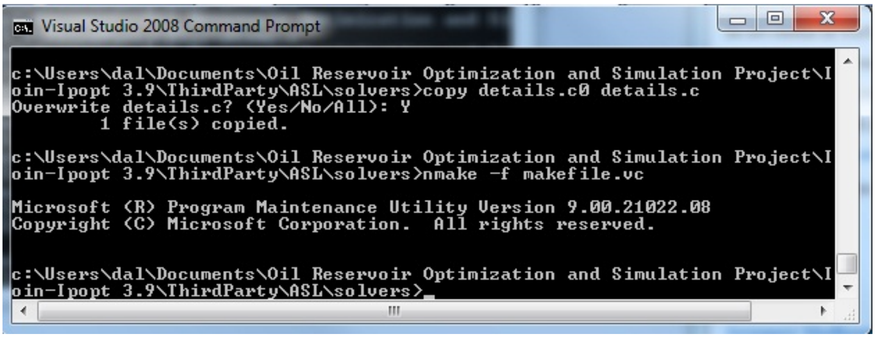
```
oin-Ipopt 3.9\Ipopt\MSVisualStudio\v8\libCoinHSL>convert_hsl.bat
..\..\..\..\ThirdParty\HSL\ma27ad.f:
   ma27id:
   ma27ad:
Warning on line 328 of ..\..\..\..\ThirdParty\HSL\ma27ad.f: ma27ud d
RNAL but never used.
   ma27bd:
   ma27cd:
   ma27gd:
   ma27hd:
   ma27ud:
   ma27jd:
   ma27kd:
   ma27ld:
   ma27md:
   ma27nd:
   ma27od:
   ma27pd:
   ma27qd:
   ma27rd:
..\..\..\..\ThirdParty\HSL\mc19ad.f:
   mc19ad:
Warning on line 167 of ..\..\..\..\ThirdParty\HSL\mc19ad.f: mc19cd d
RNAL but never used.
   mc19cd:
   BLOCK DATA mc19cd:
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Pro
oin-Ipopt 3.9\ThirdParty\HSL\dummy.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Pro
oin-Ipopt 3.9\ThirdParty\HSL\ma27ad.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Pro
oin-Ipopt 3.9\ThirdParty\HSL\mc19ad.c
        3 file(s) moved.
```

Figure 7: Convert HSL Bat File Console Output



```
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dorgtr.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dpotf2.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dpotrf.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dpotrs.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dsteqr.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dsterf.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dsyev.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dsytd2.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\dsytrd.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\ieeeck.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\ilaenv.c
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\ThirdParty\Lapack\LAPACK\SRC\iparmq.c
       36 file(s) moved.
..\..\..\..\ThirdParty\Lapack\LAPACK\INSTALL\dlamch.f:
   dlamch:
   dlamc1:
   dlamc2:
   dlamc3:
   dlamc4:
   dlamc5:
        1 file(s) moved.
c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\IPOPT\
oin-Ipopt 3.9\Ipopt\MSVisualStudio\v8\libCoinLapack>
```

Figure 8: Convert LAPACK Bat File Console Output



```

c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\I
oin-Ipopt 3.9\ThirdParty\ASL\solvers>copy details.c0 details.c
Overwrite details.c? (Yes/No/All): Y
        1 file(s) copied.

c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\I
oin-Ipopt 3.9\ThirdParty\ASL\solvers>nmake -f makefile.vc

Microsoft (R) Program Maintenance Utility Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.


c:\Users\dal\Documents\Oil Reservoir Optimization and Simulation Project\I
oin-Ipopt 3.9\ThirdParty\ASL\solvers>_
```

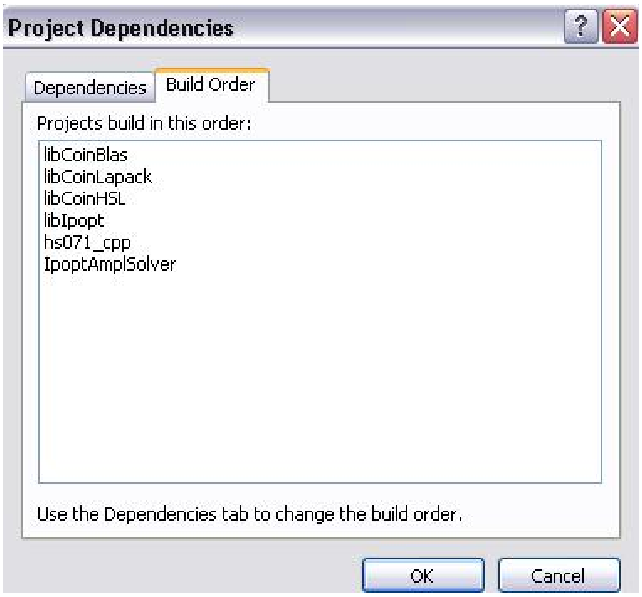
Figure 9: ASL Makefile Console Output

right click on the solution icon in the solution explorer tree view and in the right click content menu select Build Solution option.This will compile the Ipopt library, and C++ example project. It is also possible to compile each project separately. In order to do it please follow the given build order since third party code projects are also dependent on each other:

1. libCoinBLas
2. libCoinLapack
3. libCoinHSL
4. libIpopt
5. hs071_cpp
6. IpoptAmplSolver (optional, only for compiling ipopt.exe AMPL solver executable if ASL code was downloaded)

Please note that in case of building the whole solution the right build order is already specified and all projects are linked with each other so it is not necessary to worry which library project should be compiled first. To see the build order in Visual Studio please right click on solution or any project in the Visual Studio solution explorer and select Project Dependencies option in the right click content menu. This action opens a new window where build order and project dependencies can be specified for any project in the solution(figure 10). It is possible to swap between project by Projects drop down box. and edit their dependencies. On the Figure 11 project dependencies for hs071_cpp example project are shown. As it can bee seen, IpoptAmplSolver project is excluded since hs071_cpp does not implement AMPL modelling language, hence it is not necessary to compile a solver interface for it.
Now it is possible to add another C++ project that will represent the optimisation problem and call Ipopt solver to get solution. To add a new project just click on the solution icon in the Visual Studio solution explorer(tree view) and in the right click content menu select Add -> New Project and select typical C++ CLR Console Application project in the Add New Project window among available projects(Figure 12).After creating a new project it is important to specify the correct paths for the additional include directories,and also additional dependencies and library directories for the linker. To do it, please follow steps given below.

1. Right click on the new project in the Visual Studio project tree view and select properties in the content menu. In the Property Page window



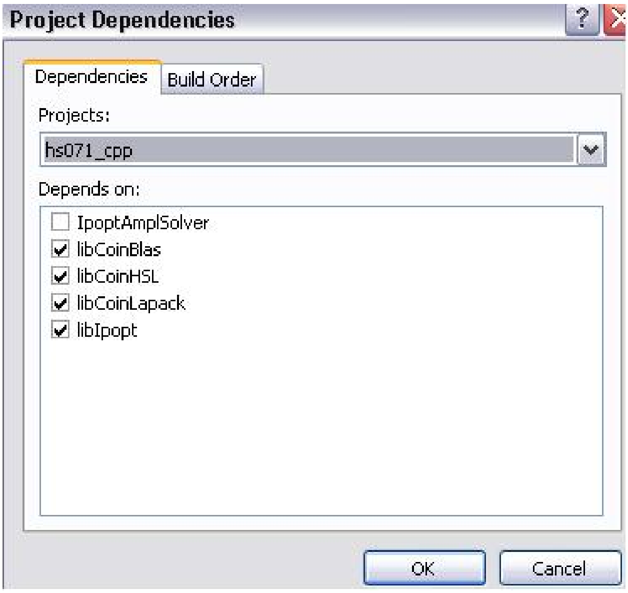
Project Dependencies

Projects build in this order:

libCoinBlas
libCoinLapack
libCoinHSL
libIpopt
hs071_cpp
IpoptAmplSolver

Use the Dependencies tab to change the build order.

Figure 10: Visual C++/C DLL Solution Project Build Order



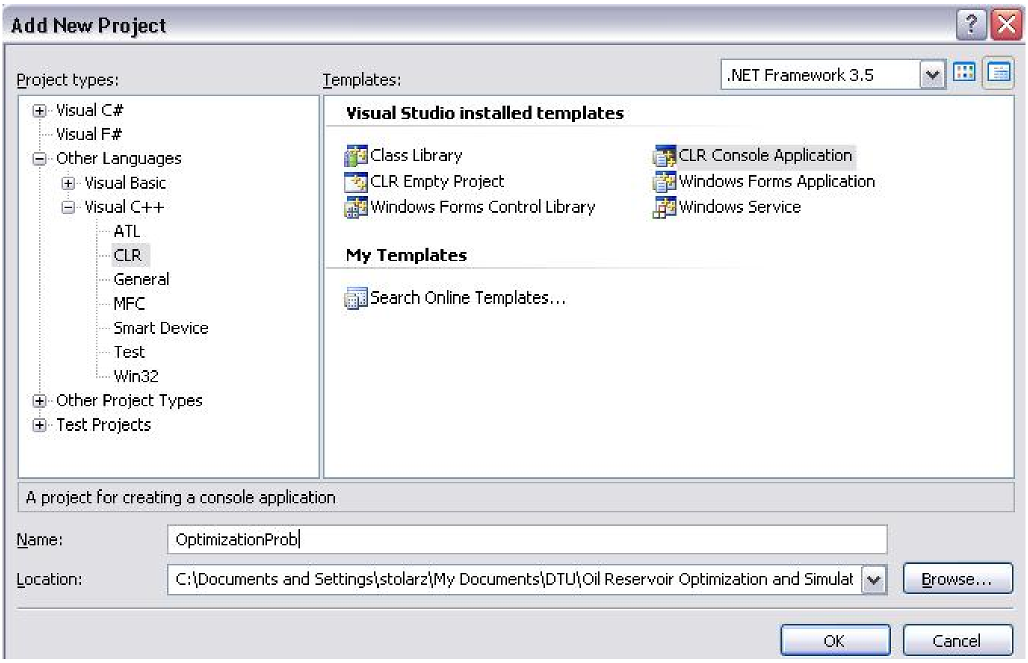
Projects:

hs071_cpp

Depends on:

- □ IpoptAmplSolver
- ☑ libCoinBlas
- ☑ libCoinHSL
- ☑ libCoinLapack
- ☑ libIpopt

Figure 11: Visual C++/C DLL Solution Project Dependencies

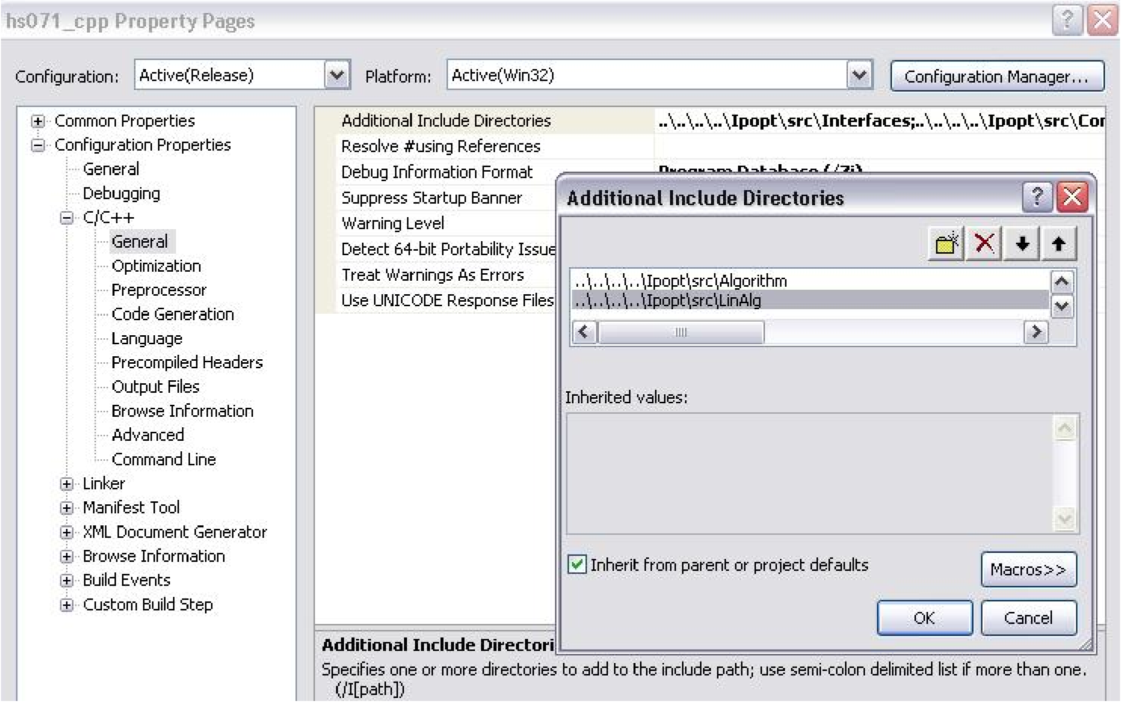


A project for creating a console application

Name: OptimizationProb

Location: C:\Documents and Settings\stolarz\My Documents\DTU\Oil Reservoir Optimization and Simulat

Figure 12: Add New Project Window



Configuration: Active(Release) Platform: Active(Win32)

Additional Include Directories ..\..\..\..\Ipopt\src\Interfaces;..\..\..\..\Ipopt\src\Cor

Resolve #using References

Debug Information Format Program Database (/Zi)

Suppress Startup Banner

Warning Level

Detect 64-bit Portability Issue

Treat Warnings As Errors

Use UNICODE Response Files

Additional Include Directories

..\..\..\..\Ipopt\src\Algorithm
..\..\..\..\Ipopt\src\LinAlg

Inherited values:

☑ Inherit from parent or project defaults

Additional Include Directori

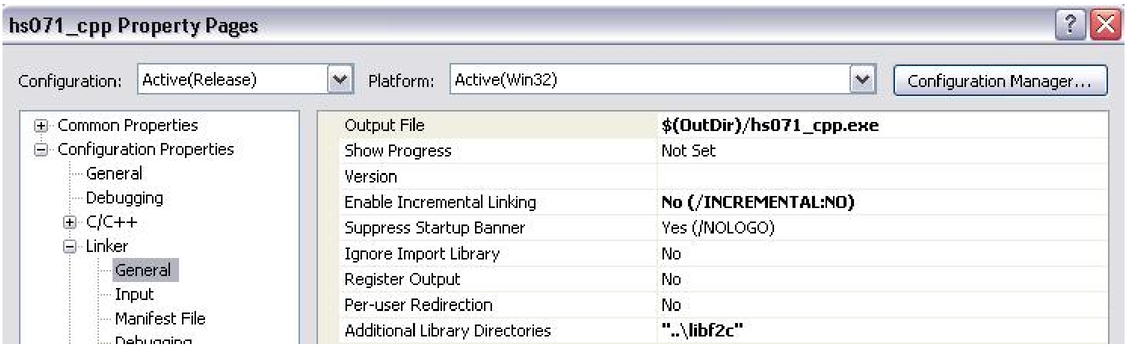
Specifies one or more directories to add to the include path; use semi-colon delimited list if more than one. (/I[path])

Figure 13: Visual C++/C DLL Solution Additional Include Directories

please go to Configuration/(C/C++)/General and add the following include directories:
..\..\..\..\Ipopt\src\Interfaces
..\..\..\..\Ipopt\src\Common
..\..\..\..\BuildTools\headers
..\..\..\..\Ipopt\src\Algorithm
..\..\..\..\Ipopt\src\LinAlg
as it was done for the hs071_cpp example project(figure 13)

2. In the Property Page window please go to Configuration/Linker/General and add the following relative path to Additional Library Directories:
..\libf2c (figure 14)

3. In the same window, please go to Linker/Input and add the following

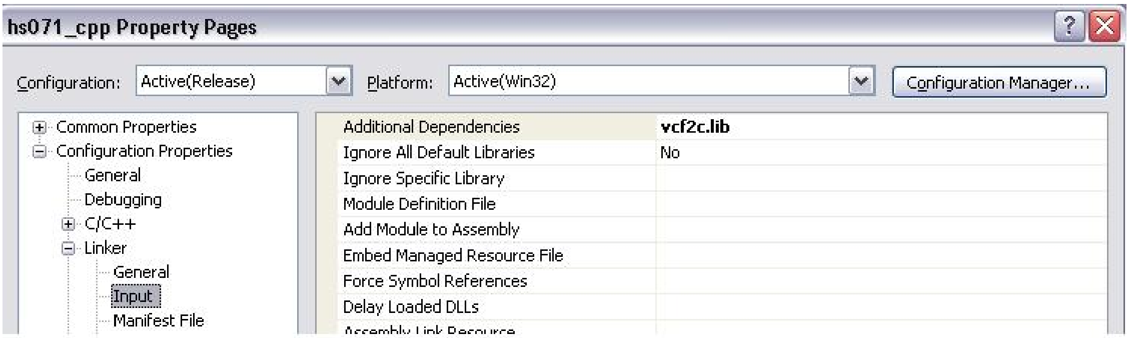


Configuration: Active(Release)

Platform: Active(Win32)

Output File	$(OutDir)/hs071_cpp.exe
Show Progress	Not Set
Version	
Enable Incremental Linking	No (/INCREMENTAL:NO)
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	"..\libf2c"

Figure 14: Visual C++/C DLL Solution Additional Library Directories



Configuration: Active(Release)

Platform: Active(Win32)

Additional Dependencies	vcf2c.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Figure 15: Visual C++/C DLL Solution Additional Dependencies

additional dependency vcf2c.lib in the Additional Dependencies field (Figure 15)

Now it is possible to start implementing optimisation problem in the new C++ project. For further information about interfacing with Ipopt using C++ language please see Ipopt documentation on the COIN - OR initiative Ipopt project documentation web page:
http://www.coin-or.org/Ipopt/documentation/
It is also possible to obtain a pdf version of documentation and tutorial in the documentation section on COIN - OR initiative Ipopt project web page:
http://www.coin-or.org/Ipopt/

5.3 Compilation of Visual C++/Fortran DLL Solution (v8-ifort folder)

This section instructs the user how to compile Visual C++/Fortran dll solution that results in the dynamic link library that can be utilised in any Visual Studio project. It relies mainly on the Intel Fortran compiler to compile the third party Fortran components (especially since the MUMPS solver uses F90). This dll includes the BLAS, LAPACK and MUMPS solvers which can be freely distributed. Alternatively, the solution contains a configuration where the resulting dll uses the (optimised, multi-threaded) Intel MKL versions of the BLAS and LAPACK libraries rather than the free-distributed netlib versions. Please note that it is user responsibility to obtain optimised multi-threaded versions of the BLAS and LAPACK libraries.

It is possible to compile separately non-free HSL and PARDISO solvers into a dll which can be dynamically loaded by the Ipopt dll when these solvers are selected by the user in the C++ program. Thus, there is a separate project to compile the HSL dll in the Visual Studio solution.

In order to compile the solution please follow the steps given below:

1. This step is only necessary if it has not been done when going through Download and Installation section.
   Download the Blas, Lapack, and MUMPS and Metis source code into(for further information about this step please go to Download and Installation section in this document or read the INSTALL files in the appropriate subdirectories of ThirdParty folder):
   IPOPTBASEDIR\ThirdParty

2. Edit the file:
   BASDIR\Ipopt\MSVisualStudio\v8-ifort\Ipopt\IntelPaths.vsprops
   to reflect the base path to the Intel compiler on the hard drive to allow the link step succeed in finding the appropriate libraries.

3. Open the solution file:
   BASDIR\Ipopt\MSVisualStudio\v8-ifort\IpOpt-ifort.sln

After selecting a build configuration (win32/x64 release/MKL release/debug), one can build the Ipopt project to obtain the Ipopt dll for that configuration. The project dependencies will make sure the all third party components are built as well. The compile step of the MUMPS Fortran project might fail for a couple times until all interdependencies between the F90 files are resolved properly.

If one wants to use the HSL solver, then it is necessary to obtain the HSL source code(please see Download and Installation section of this document) and build either the libhsl project in case of having both the sources for the MA27 and MA57 solvers, or libhsl-no-MA57 in case of having access to the MA27 solver. Both projects result in libhsl.dll, which should be placed into path to make it available to Ipopt for more information; see Wächter at 2.

Additional Remarks When linking against Ipopt.dll in newly created C++ project, one should use the IpoptApplicationFactory method exported by the DLL to obtain an IpoptApplication pointer rather than using the new operator. By doing so it is possible to call the methods of the IpoptApplication object as usual. Note, however, that the virtual interfaces exposed (recursively) by the DLL do not provide access to the more exotic interfaces or member functions in the Ipopt library. Another option for calling Ipopt methods is to use the C interface of Ipopt which is also exported by the DLL. For more information about IpoptApplication pointers please go to Ipopt tutorial and documentation available at1:
www.coin-or.org/Ipopt/documentation

6 References

1. Andreas Wächter, *Introduction to Ipopt: A tutorial for downloading, installing, and using Ipopt*, *Rev (1930)*, 2010, *https://projects.coin-or.org/Ipopt*

2. Andreas Wächter, Read Me files in Ipopt download package *https://projects.coin-or.org/Ipopt*

APPENDIX C

Derivatives of the Flux and Concentration Terms

In this appendix we explain further the analitycal expressions for the derivatives of the terms used in (5.2.8) and (5.2.9).

C.1 Concentrations

Since the concentration of either oil or water phase in the given block is a function of pressure and water saturation in this block only the derivatives of the concentration terms with respect to the states of the same block will be non-zero. The derivatives of water concentration in the block i,j with respect to presure and water saturation i, j are the following:

$$\frac{\partial C_{w,i,j}}{\partial S_{w,i,j}} = \frac{\partial(\phi\rho_w(P_{o,i,j})S_{w,i,j})}{\partial S_{w,i,j}} = \phi\rho_w(P_{o,i,j}) \tag{C.1.1a}$$

$$\frac{\partial C_{w,i,j}}{\partial P_{o,i,j}} = \frac{\partial(\phi\rho_w(P_{o,i,j})S_{w,i,j})}{\partial P_{o,i,j}} = \phi S_{w,i,j}\frac{\partial\rho_w(P_{o,i,j})}{\partial P_{o,i,j}} \tag{C.1.1b}$$

The derivatives of oil concentration in the block i,j with respect to presure and water saturation i, j are the following:

$$\frac{\partial C_{o,i,j}}{\partial S_{w,i,j}} = \frac{\partial(\phi\rho_o(P_{w,i,j})(1 - S_{w,i,j}))}{\partial S_{w,i,j}} = -\phi\rho_o(P_{o,i,j}) \tag{C.1.2}$$

$$\frac{\partial C_{o,i,j}}{\partial P_{o,i,j}} = \frac{\partial(\phi\rho_o(P_{o,i,j})(1 - S_{w,i,j}))}{\partial P_{o,i,j}} = \phi(1 - S_{w,i,j})\frac{\partial\rho_o(P_{o,i,j})}{\partial P_{o,i,j}} \tag{C.1.3}$$

Where the derivatives of the water and oil densities with respect to pressures are the following:

$$\frac{\partial\rho_o(P_{o,i,j})}{\partial P_{o,i,j}} = c_o\rho_o(P_{o,i,j}) \tag{C.1.4a}$$

$$\frac{\partial\rho_w(P_{o,i,j})}{\partial P_{o,i,j}} = c_w\rho_w(P_{o,i,j}) \tag{C.1.4b}$$

C.2 Fluxes

We have already explained in section thte in case of flux derivatives we can expect the non-zero elements when differentiating the water/oil fluxes of the block i,j with respect to states of this block as well as with respect to states of the neighbours. By neighbours we understand the blocks sharing the same interface on one of the sides, e.g. the east, west, south or north. The reason for this is that we used two point flux approximation and single point upstream scheme for flux computation where fluxes at the interfaces between 2 blocks are computed using the pressure gradient between those 2 blocks. The derivatives of water flux terms in the block i,j with respect to water saturation in this block are the following:

$$\frac{\partial F_{w,i,j}}{\partial S_{w,i,j}} = -\frac{\partial\left(\frac{\Delta N_{w,x,i,j}}{\Delta x} + \frac{\Delta N_{w,y,i,j}}{\Delta y}\right)}{\partial S_{w,i,j}} \tag{C.2.1}$$

where the derivative of flux contirbution in x direction is the following:

$$\frac{\partial\frac{\Delta N_{w,x,i,j}}{\Delta x}}{\partial S_{w,i,j}} = \frac{\partial(N_{w,x,i+,j} - N_{w,x,i-,j})}{\partial S_{w,i,j}}\frac{1}{\Delta x} \tag{C.2.2}$$

and the derivatives of the fluxes at the interfaces are the following:

$$\frac{\partial N_{w,x,i+,j}}{\partial S_{w,i,j}} = \begin{cases} 0 & P_{i+1,j} - P_{i,j} \geq 0 \\ -k_{x,i+,j}\frac{P_{i+1,j}-P_{i,j}}{x_{i+1}-x_i}\frac{\partial\zeta_{w,i,j}}{\partial S_{w,i,j}} & P_{i+1,j} - P_{i,j} < 0 \end{cases} \tag{C.2.3a}$$

$$\frac{\partial N_{w,x,i-,j}}{\partial S_{w,i,j}} = \begin{cases} 0 & P_{i,j} - P_{i-1,j} < 0 \\ -k_{x,i-,j}\frac{P_{i,j}-P_{i-1,j}}{x_i-x_{i-1}}\frac{\partial\zeta_{w,i,j}}{\partial S_{w,i,j}} & P_{i,j} - P_{i-1,j} \geq 0 \end{cases} \tag{C.2.3b}$$

While the derivatives of oil fluxes are computed in the same way, let us focus on the derivatives of water/oil flux tems with respect to oil pressure.

$$\frac{\partial F_{w,i,j}}{\partial P_{o,i,j}} = -\frac{\partial\left(\frac{\Delta N_{w,x,i,j}}{\Delta x} + \frac{\Delta N_{w,y,i,j}}{\Delta y}\right)}{\partial P_{o,i,j}} \tag{C.2.4}$$

$$\frac{\partial \frac{\Delta N_{w,x,i,j}}{\Delta x}}{\partial P_{o,i,j}} = \frac{\partial (N_{w,x,i+,j} - N_{w,x,i-,j})}{\partial P_{o,i,j}} \frac{1}{\Delta x} \tag{C.2.5}$$

$$\frac{\partial N_{w,x,i+,j}}{\partial P_{w,i,j}} = \begin{cases} k_{x,i+,j} \frac{\zeta_{w,i+1,j}}{x_{i+1}-x_i} & P_{i+1,j} - P_{i,j} \geq 0 \\ -\frac{k_{x,i+,j}}{x_{i+1}-x_i}(\frac{\partial \zeta_{w,i,j}}{\partial P_{o_i,j}}(P_{i+1,j} - P_{i,j}) - \zeta_{w,i,j}) & P_{i+1,j} - P_{i,j} < 0 \end{cases} \tag{C.2.6a}$$

$$\frac{\partial N_{w,x,i-,j}}{\partial P_{w,i,j}} = \begin{cases} -k_{x,i-,j} \frac{\zeta_{w,i-1,j}}{x_i-x_{i-1}} & P_{i,j} - P_{i-1,j} < 0 \\ -\frac{k_{x,i-,j}}{x_i-x_{i-1}}(\frac{\partial \zeta_{w,i,j}}{\partial P_{o_i,j}}(P_{i,j} - P_{i-1,j}) + \zeta_{w,i,j}) & P_{i,j} - P_{i-1,j} \geq 0 \end{cases} \tag{C.2.6b}$$

The water/oil flux derivatives with respect to states of the neighbouring blocks are computed following the same logic.

Appendix D

List of Conferences and Workshops

This appendix lists all the conferences and workshops at which the work done in this thesis was presented in form of either poster or oral presentation.

- Centre for Energy Resources Engineering (CERE) Discussion Meeting in Hillerød, Denmark on 6-8 June 2011
- Student Technical Conference STC 2011 in Wietze, Germany on 13-14 October 2011
- 17th Nordic Process Control Workshop hosted by Technical University of Denmark (DTU), Kgs Lyngby, Denmark on 25-27 January, 2012
- East Meets West International Student Petroleum Congress in Krakow, Poland hosted by AGH University of Science and Technology on 25-27 April 2012
- Centre for Energy Resources Engineering (CERE) Discussion Meeting in Hillerød, Denmark on 13-15 June 2012

Appendix E

Project Objectives (Contract)

As requested the aim of this document is to give an overall overview on what is going to be done in the Master Project entitled Modeling and Production Optimisation of Oil Reservoirs.

First of all, we will introduce ourselves into optimization topic, with special emphasis on non-linear constrained optimization. This will be done by solving some simple non-linear programs, so called nlps by using Interior Point Optimiser.

Next step is to do the research in the field of optimization of upstream oil production and try to sum up, what so far has been done in this area,, e.g. optimization methods have been used e.g. single shooting with adjoint-based gradient computation.

In addition, we will introduce our own approach for solving oil problem, which is direct collocation method, where we will translate the continuous time problem into discrete nlp. Before doing that however, we will get familiar with the tool for large scale non-linear optimization called Ipopt. The tool will be set up in Visual Studio environment and simple nlps and optimal control problems will be solved (Van Der Pol Oscillator problem) before tackling oil problem.

What is more, we will derive the mathematical model of the two phase (oil and water) immiscible flow reservoir and demonstrate how we come up with dynamic constraints on an objective cost function.

After all these steps, we will tackle oil problem using direct collocation method which will translate it to discrete nlp. The discrete nlp with its corresponding reservoir model will be implemented in Visual Studio and solved using Ipopt.

Finally, results will be shown, and conclusion will be made.

Danmark, Kongens Lyngby,
March 2012,
Dariusz Michal Lerch

Bibliography

[1] Schlumberger. *Eclipse Black Oil Simulator*. Available Online: http://www.slb.com/services/software/reseng/blackoil.aspx.

[2] C. Völcker, J.B. Jørgensen, P.G. Thomsen, and E.H. Stenby. Simulation of subsurface two phase flow in an oil reservoir. 2009.

[3] J. F. M Van Doren, R. Markovinovic, and J. D. Jansen. Reduced-order optimal control of water flooding using pod. *Computational Geosciences*, pages 137–158, 2006.

[4] John Murphy. *All-Electric Trees, Subsea Separation, Smart-Well Systems Driving Subsea Production*. Available Online: http://www.oilandgasonline.com/doc.mvc/All-Electric-Trees-Subsea-Separation-Smart-We-0001, June 2000.

[5] J.D Jansen, S.D Douma, D.R. Brouwer P.M.J. Van Den Hof, O.H. Bosgra, and A.W. Heemink. Cloosed-loop reservoir management. In *Reservoir Simulation Symposium*, volume 18, Texas, Woodlands, USA.

[6] Tendeka. *Wireless Smart Well*. Available Online: http://www.tendeka.com/products-and-services/solutions/wireless-smart-well/.

[7] R. Ringset, L. Imsland, and B. Foss. On gradient computation in single-shooting nonlinear model predictive control. Leuven, Belgium, 2010. Available Online: http://www.nt.ntnu.no/users/skoge/prost/proceedings/dycops-2010/Papers_DYCOPS2010/WeMT2-01.pdf.

[8] C. Völcker. *Production Optimisation of Oil Reservoirs*. PhD thesis, Technical University of Denmark (DTU), Centre for Energy Resources Engineering (CERE), 2012.

[9] A. Capolei, C. Völcker, J. Frydendall, and J.B. Jørgensen. *Oil Reservoir Production Optimisation using Single Shooting and ESDIRK Methods*. Technical University of Denmark (DTU), Department of Informatics and Mathematical Modelling (IMM), Centre for Energy Resources Engineering (CERE), 2011.

[10] L.T. Biegler, F. Martinsen, and B.A. Foss. *Application of optimisation algorithms in nonlinear mpc*. Department of Chemical Engineering, Carnegie Mellon University, Department of Engineering Cybernetics, Trondheim, Norway, 2004.

[11] T. A. N. Heirung, M. R. Wartmann, J. D. Jansen, B. E. Ydstie, and B. A. Foss. Optimisation of the water-flooding process in a small 2d horizontal oil reservoir by direct transcription. In *Preprints of the 18th IFAC World Congress*, Milan, Italy, August 28 - September 2 2011.

[12]

[13] E. Suwartadi, S. Krogstad, and B. Foss. On state constraints of ajoint optimisation in oil reservoir water-flooding. In *Proceeding of the Reservoir Characterisation and Simulation Conference*, Abu Dhabi, United Arab Emirates, December 2009.

[14] J.F.B.M Kraaijevanger J.F.B.M, B.V. EP, P.J.P Egberts, J.R. Valstar, and H.W Buurman. Optimal waterflood design using the adjoint method. In *SPE Reservoir Simulation Symposium, address=*.

[15] WTRG Economics. *Oil Prices 1970-2011*. Available Online: http://www.wtrg.com/prices.htm.

[16] Andy Rowell. *How about 250 USD a Barrel*. Available Online: http://priceofoil.org/2008/06/11/how-about-250-a-barrel/, http://octane.nmt.edu/gotech/Marketplace/Prices.aspx, 2008.

[17] L. T. Biegler. *A Nonlinear Programming Path to NMPC and Real-Time Optimization*. Carnegie Mellon University, USA.

[18] Lien M., Brouwer D.R, Manseth T., and Jansen J.D. Multiscale regularisation of flooding optimisation for smart field management. *SPE Journal 13(2)*, pages 195–204, 2008.

[19] G.A Virnovski. Water flooding strategy design using optimal control theory. *Proc 6^{th} European Symp on IOR, Stavanger, Norway*, pages 437–446, 1991.

[20] I.S Zakirov, S.I Aanonsen, and B.M Palatnik. Optimisation of reservoir performance by automatic allocation of well rates. *Proc 5^{th} European Conference on the Mathematics of Oil Recovery(ECMOR V), Leoben, Austria*, 1996.

[21] J.A. Skjervheim, G. Evensen, S. I. Aanonsen, B. O. Ruud, and T.A. Johansen. Incorporating 4d seismic data in reservoir simulation models using ensemble kalman filter. *SPE Journal 12(3)*, pages 137–158, 2007.

[22] H. Asheim. Maximisation of water sweep efficiency by controlling production and injection rates. *Paper SPE 18365 presented at the European Petroleum Conference, London, UK, October 16-18.*, 1988.

[23] J.D. Jansen, O.H. Bosgra, and P.M.J. Van Del Hol. Model based control of multiphase flow in subsurface oil reservoirs. *Journal of Process Control 18*, pages 846–855, 2008.

[24] G.M. Van Essen, M. J. Zandvlier, P.M.J. Van Den Hof, O.H. Bosgra, and J.D. Jansen. Robust water flooding optimisation of multiple geological scenarios. *Paper SPE 102913 Presented at the SPE Annual Technical Conference and Exhibition, San Antonio, USA, 24-27 September*, 2006.

[25] J. Gondzio and A. Grothey. *Direct Solution of Linear Systems of Size* 10^9 *Arising in Optimization with Interior Point Methods.*

[26] A. Wächter and L.T. Biegler. On the implementation of an interior-point filter line-search algorithm for large scale non-linear programming. *Springer-Verlag*, pages 25–57, April 2005.

[27] A. Wächter and L.T. Biegler. Line search filter methods for non-linear programming: Motivation and global convergence. *Technical Report RC23036, IBM T.J. Watson Research Center, Yorktown Heights, USA*, 2001.

[28] A. Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, United States., 2000.

[29] O. Schenk, A. Wächter, and M. Hagemann. Combinatorial approaches to the solution of saddle-point problems in large-scale parallel interior-point optimization. Technical Report RC 23824, IBM T. J. Watson Research Center, Yorktown, USA, December 2005.

[30] J. Nocedal, A. Wächter, and R. A. Waltz. Adaptive barrier strategies for nonlinear interior methods. Technical Report RC 23563, IBM T. J. Watson Research Center, Yorktown, USA, March 2005. revised January 2006.

[31] E. Suwartadi. *Gradient-based Methods for Production Optimization of Oil Reservoirs.* PhD thesis, Department of Engineering Cybernetics, Faculty of Information Technology, Mathematics and Electrical Engineering, Norwegian University of Science and Technology, Trondheim, Norway.

[32] L. Saputelli, M. Nikolaou, and M.J. Economides. Real-time reservoir management: a multi-scale adaptive optimisation and control approach.

[33] J.B. Rawlings. Tutorial overview of model predictive control. *IEE Control Systems Magazine*, June, 2000.

[34] J.K Huusom, N.K Poulsen, S.B Jørgensen, and J. Bagterp Jørgensen. *SISO Offset-Free Model Predictive Control Based on ARX Models*. Lyngby 2800, Denmark, 2011.

[35] Wikipedia Commons. *Model Predictive Control*.

[36] A. Wächter . Introduction to ipopt: A tutorial for downloading, installing, and using ipopt.

[37] C. Guichard, J. Barc, R Eymard, and R. Massib. Finite volume schemes for multiphase flow simulation on near well grids. In *ECMOR XII European Association of Geoscientists and Engineers*, Oxford, UK, September 2010.

[38] I. Aavatsmark and R. Klausen. Well index in reservoir simulation for slanted and slightly curved wells in 3d grids. *SPE Journal*, (03), 2003.

[39] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: Steady-State and Time-Dependent Problems*. Industrial and Applied Mathematics (SIAM), Philadephia, US.

[40] H. Versteeg and W. Malalasekra. *An Introduction to Computational Fluid Dynamics : The Finite Volume Method*. Pearson, Harlow, England.

[41] K. Aziz and A. Settari. *Petroleum Reservoir Simulation*. Applied Science Publishers Ltd, London, England, first edition edition, 1971.

[42] J.H. Abou-Kassem and K. Aziz. Analytical well models for reservoir simulation. 1985. Available Online: http://www.onepetro.org/mslib/app/Preview.do?paperNumber=00011719&societyCode=SPE.

[43] C. Palagi and K. Aziz. Modelling vertical and horizontal wells with voronoi grid. *SPE Reservoir Engineering*, (02).

[44] J. A. Holmes. Modelling advanced wells in the reservoir simulation. *SPE Journal of Petroleum Technology*, November 2001. Available Online: http://www.onepetro.org/mslib/app/Preview.do?paperNumber=00072493&societyCode=SPE.

[45] C. Wolfsteiner, L. J. Durlofsky, and K. Aziz. Calculation of well index for nonconventional wells on arbitrary grids. *Computational Geosciences*, 7:61–82, November 2001.

[46] C. Wolfsteiner, L. J. Durlofsky, and K. Aziz. Approximate model for productivity of non-conventional wells in heterogeneous reservoir. *SPE Journal*, (06), 2003. Available Online: http://www.onepetro.org/mslib/app/Preview.do?paperNumber=00072493&societyCode=SPE.

[47] D. Peaceman. Interpretation of well-block pressures in numerical reservoir simulation. pages 391–402, June 1978. Available Online: http://www.onepetro.org/mslib/app/Preview.do?paperNumber=00006893&societyCode=SPE.

[48] C. Völcker and E. H. Stenby J.B. Jørgensen, P.G. Thomsen. Explicit singly diagonally implicit runge-kutta methods and adaptive stepsize control for reservoir simulation. In *ECMOR XII 12th European Conference on the Mathematics of Oil Recovery*, Oxford, United Kingdom, September 2010.

[49] D.R Brouver and J.D Jansen. Dynamic optimisation of water flooding with smart wells using optimal control theory. *SPEJ 9 (4)*, pages 391–402, 2004.

[50] R.A. Berenblyum, A. Shapiro, E.H. Stenby, K. Jensen, and F.M. Orr Jr. Black oil streamline simulator with capillary effects. *SPE Annual Technical Conference and Exhibition*, pages 133–141, 2003.

[51] M.R. Thiele and R.P. Batycky. Using streamline-derived injection efficiencies for improved water flood management. *SPEREE*, pages 187–196, 2006.