NTNU – Trondheim
Norwegian University of
Science and Technology

Specialization Project Fall 2022

A comparative study of distributed feedback-optimizing control strategies

Vegard Aas

December 16, 2022

Abstract

In industry, the processes often consist of several subsystems with a common constraint, for example, a shared resource. This paper considers the problem of steady-state real-time optimization (RTO) for a subsea gas-lifted oil production network with multiple wells and constrained access to shared gas-lift. Problems like this are often solved with a centralized optimization, which can be computationally expensive and tends to be slow.[13] Morari et al. proposed to eliminate the numerical optimization problems by indirectly moving the problem into the control layer. Such problems are known as feedback-optimizing control, which can be implemented using simple tools such as Proportional-Integral-Derivative (PID) controllers. Most importantly this optimizing method is accommodating for different operation units with different time scale.

In our previous work, we have experimentally validated a recently developed method of feedback-optimizing control called distributed feedback-based RTO[5]. This method is developed based on dual decomposition and optimally handles steady-state changes in active constraints. However, the constraints are controlled in a slower time scale by updating the dual variables. This leads to the need for significant "back-off" strategy, which could lead to profit loss in the long run. To eliminate or reduce the "back-off", Dirza et al.[6] introduces an alternative distributed feedback-optimizing control based on online primal decomposition using feedback and constraint controller(s) which distribute local setpoints without violating the common constraint to avoid or minimize use of a "back-off" strategy.

As the continuation of these approaches, we compare them on a model of a lab scale experimental rig that emulates a subsea gas-lifted oil production optimization network with uncertainty and measurment noise.

Contents

1 Introduction **1**

2 Theory **2**
2.1 Feedback Control . . . 2
2.1.1 PID Control . . . 3
2.1.2 SIMC Tuning Method . . . 3
2.2 Real-Time Optimization . . . 4
2.3 Feedback-Optimizing Control . . . 5
2.3.1 Distributed Feedback-Optimizing Control using Online Primal Decomposition . . . 5
2.3.2 Online Primal Decomposition Framework . . . 7
2.3.3 Distributed Feedback-Optimizing Control using Dual Decomposition . . . 8
2.3.4 Dual Decomposition Framework . . . 10

3 Model **11**
3.1 Optimization Problem . . . 11
3.2 Control Setup . . . 12
3.2.1 Primal Decomposition Setup . . . 12
3.2.2 Dual Decomposition Setup . . . 13

4 Results **15**
4.1 Determine Compensator Subsystem for Primal Decomposition Framework . . . 16
4.2 Comparison of Feedback-Optimizing Control . . . 17

5 Conclusion **22**
5.1 Further Work . . . 22

A Steady-State Gradient Estimation **24**
A.1 Extended Kalman Filter . . . 24
A.2 Forward Sensitivity Analysis . . . 24

B Dynamic Lab-Rig Model **25**

C MATLAB Code **27**
C.1 LabViewMain - Primal Decomposition . . . 27
C.2 LabViewMain - Dual Decomposition . . . 32
C.3 Extended Kalman Filter . . . 37
C.4 Dynamic Model of Lab Rig . . . 40
C.5 Main . . . 43

List of Figures

2.1 Block diagram of simple feedback control system [15] . . . 2
2.2 Step response of first-order system with delay, $g(s) = ke^{-\theta s}/(\tau_1 s + 1)$ [14] . . . 4
2.3 Online primal decomposition control structure. . . . 8
2.4 Block diagram of dual decomposition control structure. . . . 10
3.1 Schematic of lab rig which the model is based on. This schematic is adapted from Matias et al. [12] . . . 12
4.1 Disturbances . . . 15
4.2 Comparison of accumulated profit with different compensator wells. Well 3 vs well 1 and well 3 vs well 2 denotes the comparison of accumulated profit with well 3 as compensator versus accumulated profit with well 1 and 2 as compensator respectively. . . . 16
4.3 The constraint satisfaction (total implemented gas-lift) of both primal and dual decomposition. There is a magnifying plot in time window 5 to 6.5 min showing only constraint satisfaction of primal decomposition. . . . 17
4.4 The gas-lift flow rate setpoints ($\mathbf{u}^{sp} = \mathbf{Q}_{gl}^{sp}$) for every well for both primal and dual decomposition control method. . . . 18
4.5 The actual implemented gas-lift flow rates in every well for both primal and dual decomposition control method. . . . 19
4.6 The local Lagrange multipliers for primal decomposition. . . . 20
4.7 The local Lagrange multipliers for primal decomposition with no measurement noise. . . . 20
4.8 Lagrange multiplier for dual decomposition. . . . 20
4.9 Comparison of the total liquid flow in the simulation model . . . 21
4.10 Instantaneous profit of primal and dual decomposition compared to a naive approach. . . . 21
B.1 Diagram of a single well model. Q_l, P_{rh} and P_{pump} are measured. The reservoir valve opening v_0 is assumed as a measured disturbance and Q_g is controlled by the gas flowrate controller. . . . 25
C.1 Information flow in MATLAB code . . . 27

List of Tables

3.1 Controller and tuning parameters for primal decomposition . . . 13
3.2 Controller and tuning parameters for dual decomposition . . . 15

1 Introduction

Industrial process often consists of several subsystems with a common constraint, for example, a shared resource which should be distributed optimally amongst the subsystems. A common power plant which deliver steam to different sub-processes [8] [16] is an example of this optimization problem. Such problems are typically dealt with in the context of real-time optimization (RTO). The traditional RTO paradigm involves solving numerical optimization, which utilize detailed process models that are updated using process measurements, this tends to be slow and can often be computationally expensive. Moriari et. al. [13] proposed to eliminate the numerical optimization problem by indirectly moving the problem into the control layer. Such problems are known as feedback-optimizing control, which can be implemented using simple tools such as proportional-integral-derivative (PID) controllers. Most importantly this optimizing method is accommodating for different operation units with different time scale.

In this paper a problem of steady-state real-time optimizing for a subsea gas-lifted oil production network with multiple wells and constrained access to a shared gas-lift supply is considered [5] [6] [7] [11]. In such cases, it often is desirable to decompose the large scale problem and solve the subproblems locally for various reasons. As opposed to large scale centralized optimization, distributed decision making tools are often easier to maintain and implement. The different decomposition strategies for these large scale problems can largely be classified as primal decomposition and dual decomposition methods. [2]

In recent articles such as Krishnamoorthy et. al. [10] and Dirza et. al. [7], a distributed RTO framework based on dual decomposition that uses simple feedback control to ensure convergence to a stationary point of the system wide optimization problem is proposed. In this method the constraint economic optimization problem is transformed into an unconstrained optimization problem using Lagrangian relaxation. The optimal operation point is then asymptotically reached by constraint control using the Lagrange multipliers (dual variables) in a central constraint controller in the slow time scale, while the gradient of the Lagrangian is controlled to zero in a faster time scale using the physical manipulated variables (primal variables) in a cascade fashion.

Recently a distributed feedback-based real-time optimizing framework using online primal decomposition was suggested by Dirza et. al [6]. In this method the Lagrange multipliers for each individual subproblem is estimated and used in a central constraint controller to reach the optimal operation point. The goal of using the primal decomposition method is to achieve optimal operation with minimum dynamic constraint violation.

In this work we are going to provide a comparison of two feedback-optimizing control structures with decomposition strategies mentioned above, they will be implemented on the problem of steady-state real-time optimizing for a subsea gas-lifted oil production network with multiple wells and constrained access to a shared gas-lift supply mentioned earlier. Since there is a constraint access to a shared gas-lift supply, it is necessary to optimally allocate the gas-lift among the different wells. In this paper we do not consider any limitations from a topside production facility, therefore it is assumed always active input shared constraint.

The remainder of this paper is organized as follows. Section 2 describes some background theory. Section 3 presents the simulation model of the gas-lifted oil production system and the implementation of the methods. Section 4 presents the results from the simulation model before concluding the paper in section 5.

2 Theory

2.1 Feedback Control

The objective of control is to regulate a process. In many applications, it is important to control e.g. flow rates, levels, pressures or temperatures. The main objectives of control are setpoint tracking, disturbance rejection and ensuring safety. Setpoint tracking involves that a variable should follow a specified path in time. This could be because the optimal operating point of a process may vary over time. Disturbance rejection means that any disturbances affecting the process output should be compensated, this is important to ensure consistency and predictability of the process output. Some processes have risks of e.g. thermal runaway or increase in pressure that may pose a threat to environment and personnel. In these cases control is important to ensure safety.

In this paper we use feedback control, the structure of which is presented in figure 2.1. The advantage of feedback control is that the controller is error driven, therefore it compensates for disturbances automatically. As a result of this, there is no requirement for a detailed mathematical model of the relationship between output and disturbance to achieve good performance in the controller. One drawback of the error driven nature of the feedback controller however, is that it only can take corrective action after the error from the disturbance is detected in the output. Thus, significant dead time between measured output variable and the manipulated, or input variable could have significant impact on the performance of the controller. [9]

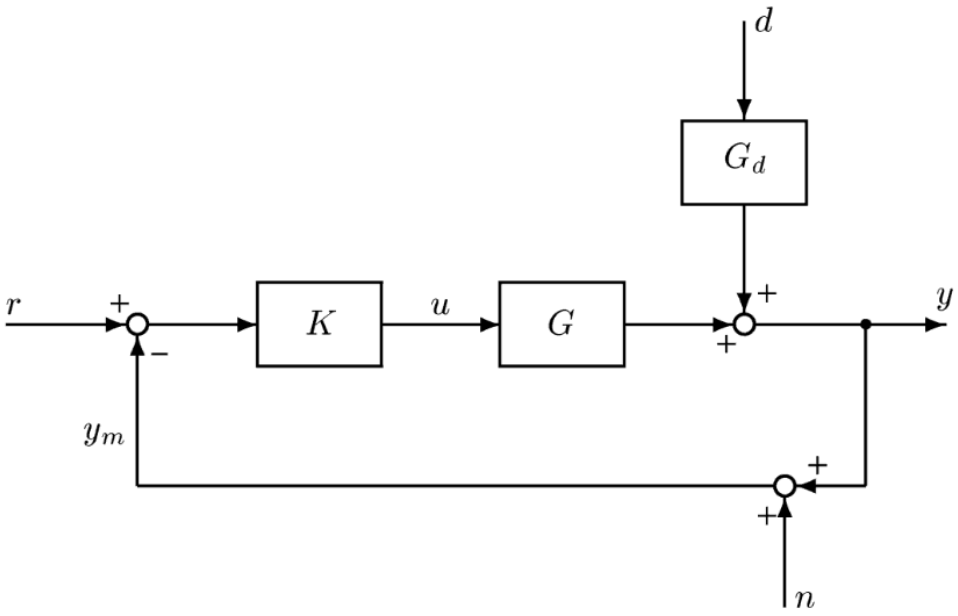

Figure 2.1: Block diagram of simple feedback control system [15]

In figure 2.1 the controller input is $r-y_m$ where r is the setpoint of the controller and $y_m = y+n$. y is the actual value while n is the measurement noise. Thus, the plant input is

$$u = K(r - y - n) \tag{2.1}$$

As mentioned above, the objective of control is to manipulate the input such that the error remains small with the presence of disturbances d. The control error e is defined as

$$e = y - r \tag{2.2}$$

This can be achieved by designing a controller K, which could be done by implementing a PID controller and SIMC tuning rules.[15]

2.1.1 PID Control

Other than the on-off-controller, the simplest available feedback controller is the proportional controller. This is a linear controller where the control signal is proportional to the control error. The output signal from a proportional controller is given by

$$u(t) = Ke(t) \tag{2.3}$$

where K is the proportional gain and e is the error given by,

$$e(t) = y_s(t) - y(t) \tag{2.4}$$

where y is the controlled variable measurement and y_s is the setpoint. Eq. 2.4 shows that there must be an error for the controller to produce an output. As a consequence, the proportional controller will always have an offset between the measured variable and the setpoint at steady-state. Therefore, a more complex controller is necessary to eliminate this offset.

Proportional-integral-derivative controller, or PID controller, is by far the most widely used control algorithm in the process industry. The PID algorithm in the time domain is described as follows

$$u(t) = K\left(e(t) + \frac{1}{\tau_I}\int_0^t e(\tau)d\tau + \tau_D\frac{de(t)}{dt}\right) \tag{2.5}$$

where e is the control error, τ_I and τ_D are the integral and derivative time respectively, t is the time and K is the proportional gain.

As seen in eq. 2.5 the controller output, u, is a sum of three elements. The P-term is proportional to the error, the I-term is proportional to the integral duration of the error, and the D-term is proportional to the derivative, the rate of change, of the error. Thus the tuning parameters of the PID-controllers are the proportional gain K, the integral time τ_I and the derivative time τ_D.

The purpose of integral action is to correct any steady state offset from a constant reference signal value. Larger integral time, τ_I, gives less I-action and slower control.

The derivative action is intended to predict the control action. It uses the rate of change in the error to make the predictions and avoid large error in the future. The derivative action is sensitive to measurement noise. Larger derivative time, τ_D, gives more D-action and faster control.[9]

2.1.2 SIMC Tuning Method

As mentioned in section 2.1.1 a PID controller only has three tuning parameters, however, it is not easy to determine these without a systematic approach. There are several methods for tuning PID controllers available in literature, Ziegler-Nichols[18] and Cohen-Coon[4] are some examples. In this paper, Skogestad Internal Model Control (SIMC) method is implemented. SIMC method was developed by Prof. Sigurd Skogestad where the objective of the rules is to be simple and memorizable, while still ensuring good controller performance.

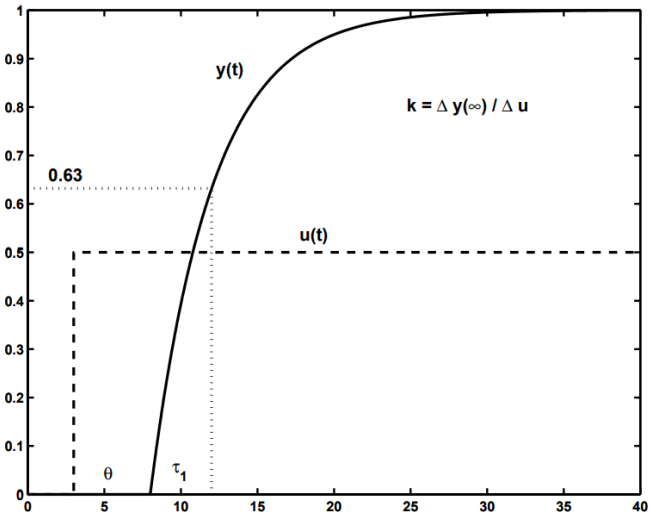

Figure 2.2: Step response of first-order system with delay, $g(s) = ke^{-\theta s}/(\tau_1 s + 1)$ [14]

The controller tunings of SIMC method are based on a first- or second-order plus delay model approximation of the process with the following model information:

- Plant gain, k
- Dominant time constant, τ_1
- Effective time delay, θ
- Second-order time constant, τ_2 (only used for dominant second-order process ($\tau_2 > \theta$))

This information can be obtained in several ways. Estimated from an open-loop step response, a closed-loop response with a P-controller, or from a detailed model where the effective dead time is approximated by use of the half rule. How to find these parameters from the open-loop step response is presented in figure 2.2.

Using the open-loop step response method, the model can be approximated as a first-order model plus time delay on the transfer function form

$$g(s) = ke^{-\theta s}/(\tau_1 s + 1) \tag{2.6}$$

The SIMC tuning rules can be derived following the Internal Model Control approach, where one specifies the first-order closed-loop response with time constant τ_c. The SIMC tuning rules result in the following PI controller settings

$$K = \frac{1}{k}\frac{\tau_I}{\tau_c + \theta} \tag{2.7}$$

$$\tau_I = min\{\tau_1, 4(\tau_c + \theta)\} \tag{2.8}$$

where τ_1 is the first-order time constant and k is the steady state plant gain as shown in figure 2.2. This results in just one remaining tuning parameter, which is the desired closed-loop time constant τ_c. [14]

2.2 Real-Time Optimization

Real-time optimization (RTO) is a technique for production optimization where the goal is to improve plant economic performance in real-time. In traditional steady-state RTO a steady-

state model is updated with the current plant state, the model is then used to compute the optimal operating point, which is implemented in the plant [3].

Industrial process often consist of large-scale systems such as a subsea oil production network with several wells (subsystems) where we want to maximize production. In these cases it often is desirable to decompose the optimization problem and solve the individual subproblems locally.

In this paper the system consists of N subsystems, which we combine to describe the optimization problem for the entire system. The subsystems are expressed by the set $\mathcal{N} = \{1, ..., N\}$. We assume that each subsystem is locally optimized.

Subsystem i is modelled as a nonlinear state-space system.

$$\begin{aligned} \dot{\mathbf{x}}_i &= \mathbf{f}_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) \\ \mathbf{y}_i &= \mathbf{h}_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) \end{aligned} \tag{2.9}$$

where $\mathbf{x}_i \in \mathbb{R}^{n_{x,i}}$, $\mathbf{u}_i \in \mathbb{R}^{n_{u,i}}$, $\mathbf{d}_i \in \mathbb{R}^{n_{d,i}}$ and $\mathbf{y}_i \in \mathbb{R}^{n_{y,i}}$ is the vector of states, inputs, disturbances and measurements of each subsystem. There may also be local constraints in each subsystem that are handled locally.

The overall network is considered as a nonlinear state-space system where the inputs, states and disturbances are defined as

$$\begin{aligned} \mathbf{u} &= [\mathbf{u}_1, ..., \mathbf{u}_N]^T \\ \mathbf{x} &= [\mathbf{x}_1, ..., \mathbf{x}_N]^T \\ \mathbf{d} &= [\mathbf{d}_1, ..., \mathbf{d}_N]^T \end{aligned} \tag{2.10}$$

The resulting steady-state optimization problem is

$$\min_{\mathbf{u}_i, \forall i \in \mathcal{N}} \quad J_{\mathcal{N}} = \sum_{i \in \mathcal{N}} J_{\mathcal{N}_i} \tag{2.11a}$$

$$\text{s.t.} \quad \mathbf{f}(\mathbf{x}, \mathbf{u}, \mathbf{d}) = 0, \tag{2.11b}$$

$$g(\mathbf{x}, \mathbf{u}, \mathbf{d}) \leq 0 \tag{2.11c}$$

here constraint 2.11b is related to the entire system model, while constraint 2.11c is a inequality constraint. $J_{\mathcal{N}_i}$ is the cost function for subsystem i.

2.3 Feedback-Optimizing Control

2.3.1 Distributed Feedback-Optimizing Control using Online Primal Decomposition

To solve the integrated optimization problem 2.11 it is required a detailed model, which may be unnecessary and unpractical in terms of computing power. Hence, we want to decompose problem 2.11, and solve it in a distributed manner. By decomposing a system we are also able to solve the individual subsystems in different time scales. In this paper we use an online optimizing method, which uses simple feedback control based on primal decomposition.

Constraint 2.11c can be defined as a linear constraint, $g(\mathbf{x}, \mathbf{u}, \mathbf{d}) = \sum_{i=1}^{N} g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) - g^{\max}$. In this paper we consider a shared resource, therefore the inequality constraints in 2.11c are converted to equality constraints. This does not change the structure of 2.11a and 2.11b, as the cost is additively separable and the system model is independent for each subsystem.

Introducing an initial value of local constraint, g_i^{sp}, where $g^{sp} = \sum_{i=1}^{N} g_i^{sp}$, for the coupling constraint variables and take care of the active coupling constraint satisfaction with a central

problem, the integrated optimization problem 2.11 can be depicted as the separable problem below.

$$\min_{\mathbf{u}_i, \forall i \in \mathcal{N}} \quad J_{\mathcal{N}} = \Sigma_{i \in \mathcal{N}} J_{\mathcal{N},i} \tag{2.12a}$$

$$\text{s.t.} \qquad \mathbf{f}_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) = 0, \forall i \in \mathcal{N}, \tag{2.12b}$$

$$g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) - g_i^{sp} = 0, \forall i \in \mathcal{N} \tag{2.12c}$$

$$\sum_{i \in \mathcal{N}} g_i^{sp} = g^{\max} \tag{2.12d}$$

When eq. 2.12d is satisfied, the primal feasibility is guaranteed for the coupling constraint 2.11c.

Problem 2.12 can be rewritten as a Lagrange function by relaxing the local constraint 2.12c. This Lagrange function can be decomposed into subproblems, then the optimization problem can be solved for each subsystem i.

$$\mathcal{P}_i(g_i^{sp}) := \min_{\mathbf{u}_i} \mathcal{L}_i(\mathbf{u}_i, g_i^{sp}, \lambda_i) \tag{2.13}$$

where

$$\mathcal{L}_i(\mathbf{u}_i, g_i^{sp}, \lambda_i) = J_{\mathcal{N},i} + \lambda_i g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i). \tag{2.14}$$

λ_i is the local Lagrange multiplier, which is associated with local constraint 2.12c. In steady-state optimal conditions the local constraint converges to the same value. [6]

Each subproblem estimates local Lagrange multipliers, which is used in central constraint controllers. These controllers update the setpoints iteratively, where the goal is to provide setpoints that satisfy the primal feasibility 2.12d.

$$\min_{g_1^{sp}, \ldots, g_N^{sp}} \quad \sum_{i \in \mathcal{N}} \mathcal{P}_i(g_i^{sp}) \tag{2.15a}$$

$$\text{s.t.} \qquad \sum_{i \in \mathcal{N}} g_i^{sp} = g^{\max} \tag{2.15b}$$

$\mathcal{P}_i(g_i^{sp})$ is given by 2.13 while the constraint 2.15b stem from 2.12d.

One of the local setpoints is calculated as follows,

$$g_N^{sp,k+1} = g^{\max} - (g_1^{sp,k+1} + \ldots + g_{N-1}^{sp,k+1}) \tag{2.16}$$

This is to ensure primal feasibility, we call this subsystem (e.g., subsystem N as displayed in 2.16) the compensator subsystem.

For the remaining subproblems, $j = \{0, \ldots, N-1\}$, each local setpoint g_i^{sp} at time step $k+1$ can be found using the steepest descent direction of the central problem 2.15a. This is given by the subgradient,

$$\nabla_{g_j^{sp}} \left(\sum_{i \in \mathcal{N}} \mathcal{P}_i(g_i^{sp,k}) \right) = -\lambda_j^k + \lambda_N^k \tag{2.17}$$

At the next time step the updated local setpoint is found by,

$$g_i^{sp,k+1} = g_i^{sp,k} + K_{I,i} \nabla_{g_i^{sp}} \left(\sum_{i \in \mathcal{N}} \mathcal{P}_i(g_i^{sp,k}) \right) \tag{2.18}$$

We may use integrating controllers with integral gain $K_{I,i} = \frac{1}{K_i(\tau_{c,i})}$, where K_i is the step response gain and $\tau_{c,i}$ is the closed-loop time constant.

To find the local setpoints the local Lagrange multiplier estimates is required in 2.17. Usually this is available when solving the numerical optimization problem in the traditional RTO framework. In this paper however, we are using feedback control, this is not directly available. Therefore, we have to estimate the multipliers. For all subsystems, the stationary point is reached when

$$\nabla_{\mathbf{u}_i} \mathcal{L}_i(\mathbf{u}_i, g_i^{sp}, \lambda_i) = 0 \tag{2.19}$$

according to Karush-Kuhn-Tucker (KKT) conditions. All multipliers, λ_i converge to the same optimal value. By rearranging 2.19 the local Lagrange multiplier λ_i can be computed,

$$\lambda_i = -\nabla_{\mathbf{u}_i} J_{\mathcal{N},i} (\nabla_{\mathbf{u}_i} g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i))^{-1} \tag{2.20}$$

The amount of local manipulated variables must be more than or equal to the amount of constraints, the solution also has to be unique.

2.3.2 Online Primal Decomposition Framework

By combining the idea of central constraint controllers and local Lagrange multiplier estimation with the concept of primal decomposition, as described earlier, Dirza et. al. proposed to solve the problem of real-time resource allocation in handling coupling constraint by using distributed feedback-optimizing control with a primal decomposition framework [6]. When this framework is used, we are guaranteed primal feasibility and it can theoretically reach steady-state optimal condition.

In figure 2.3 the structure of this framework is illustrated. The central constraint controllers, made up of both normal and compensator subsystems, give new set points for the local constraints, g_i. How this is calculated is presented in eq. 2.16 and 2.18. If there is any disturbance $\mathbf{d}_i$, it is possible to use the plants current information to estimate its current state and parameters using a local dynamic estimator as for instance Extended Kalman Filter (EKF). By looking at the inputs, estimated states and parameters, we can estimate cost gradient as well as constraint gradient to compute the local Lagrange multipliers, shown in eq. 2.20. After these are calculated, the multipliers are used to calculate the new setpoints in the central constraint controller.

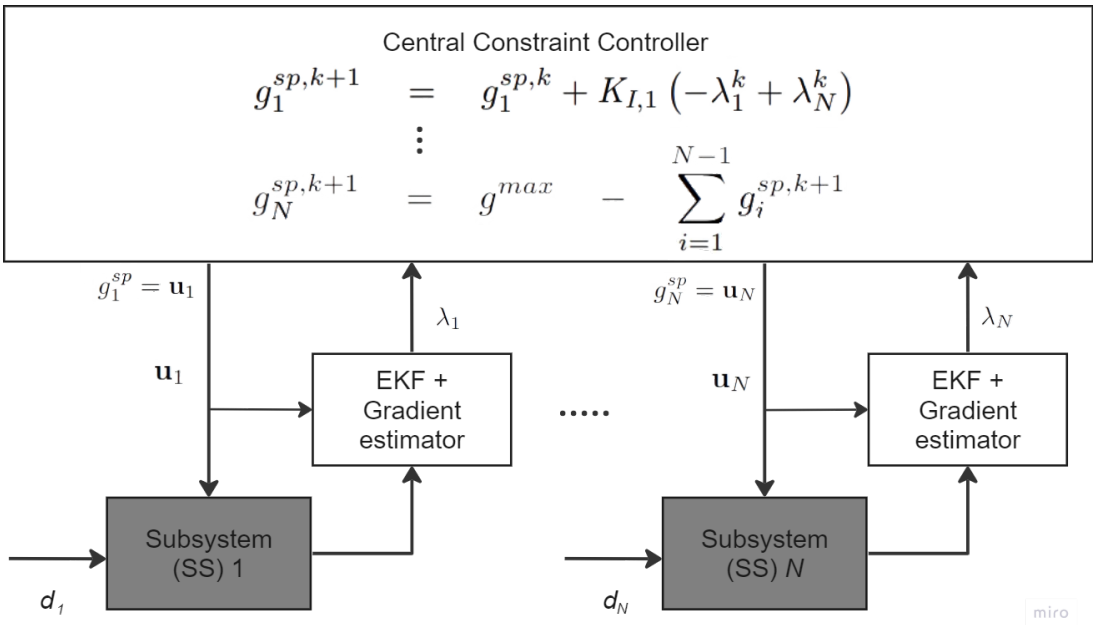

Figure 2.3: Online primal decomposition control structure.

2.3.3 Distributed Feedback-Optimizing Control using Dual Decomposition

Consider steady-state optimization problem 2.11 without constraint 2.11b

$$\min_{\mathbf{u}_i, \forall i \in \mathcal{N}} \quad J_{\mathcal{N}} \tag{2.21a}$$

$$s.t. \quad \mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) \leq 0 \tag{2.21b}$$

Introducing the Lagrangian function for problem 2.21,

$$\mathcal{L}(\mathbf{x}, \mathbf{u}, \mathbf{d}, \lambda) = J_{\mathcal{N}} + \lambda^\top \mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) \tag{2.22}$$

the KKT conditions for problem 2.21 can be expressed as

$$\nabla_u \mathcal{L}(\mathbf{x}, \mathbf{u}, \mathbf{d}, \lambda) = \nabla_u J_{\mathcal{N}} + \lambda^\top \nabla_u \mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) = 0 \tag{2.23a}$$

$$\mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) \leq 0 \tag{2.23b}$$

$$\lambda \geq 0 \tag{2.23c}$$

$$\lambda_i g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) = 0, \forall i \in \mathcal{N} \tag{2.23d}$$

where $\mathbf{u}$ and λ are the unknown variables. The equations in 2.23 can be solved using dual ascent.[1] 2.23a is solved with respect to $\mathbf{u}$ and a fixed value of λ. The λ is iteratively changed in an outer loop in order to satisfy the remaining equations. When we have active constraints, which is the case in this paper, it is important to keep $\mathbf{g} = 0$, this corresponds to a nonzero λ. As the constraint value $\mathbf{g}$ often is measured, we could use feedback control to solve the equations. An advantage of using feedback control when solving problem 2.23 with respect to λ is that we do not rely on the model for the constraints $\mathbf{g}$ in 2.23b and 2.23d, hence, we do not have to update the model for this part. As a result of this, we can reach stationary steady-state conditions by controlling

$$c(\lambda) := \nabla_u J_{\mathcal{N}} + \lambda^\top \nabla_u \mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) \tag{2.24}$$

to a constant setpoint $c^{sp} = 0$ for any given λ. The gradient controllers (fast time-scale) are responsible for this function. The Lagrange multipliers λ_i are used as manipulated variables in an outer loop to control the corresponding constraints $g_i(\mathbf{x}, \mathbf{u}, \mathbf{d})$ to the limit 0 for $i \in \mathcal{N}$. By implementing a max selector and pairing λ_i to $g_i(\mathbf{x}, \mathbf{u}, \mathbf{d})$, we ensure that the conditions $\lambda \geq 0$ 2.23c and the correspondent slackness condition 2.23d are satisfied. The centralized constraint controllers (slow time-scale) are responsible for this function. Together, the gradient and centralized constraint controllers satisfy the necessary optimality conditions 2.23 at steady-state.

As in section 2.3.1 we want to decompose the optimization problem and solve it in a distributed manner. Constraint 2.21b can be defined as a linear constraint

$$\mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d}) = \sum_{i=1}^{N} g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) - g^{\max} \tag{2.25}$$

where g^{max} is the limit of the constraint. The structure of 2.21a remain unchanged, as the cost is additively separable. Hence, by implementing equation 2.25 the integrated optimization problem 2.21 can be depicted as the separable problem.

$$\min_{\mathbf{u}_i, \forall i \in \mathcal{N}} \quad \sum_{i \in \mathcal{N}} J_{\mathcal{N},i} \tag{2.26a}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} g_i(\mathbf{x}_i, \mathbf{u}_i, \mathbf{d}_i) - g^{max} \leq 0, \forall i \in \mathcal{N} \tag{2.26b}$$

In this case the controlled variable becomes

$$c(\lambda) := \sum_{i=1}^{N} \nabla_{u_i} J_{\mathcal{N},i} + \lambda^\top \sum_{i=1}^{N} \nabla_{u_i} g_i(x_i, u_i, d_i) \tag{2.27}$$

which is additively separable. Thus, we can easily decompose equation 2.27 and each subsystem control the following

$$c_i(\lambda) := \nabla_{u_i} J_{\mathcal{N},i} + \lambda^\top \nabla_{u_i} g_i(x_i, u_i, d_i) \tag{2.28}$$

to the setpoint of $c_i^{sp} = 0$ by manipulating u_i. Considering only integral action the controller action is given by

$$\Delta u_i = K_{I,i} c_i(\lambda) \tag{2.29}$$

where $K_{I,i}$ is the integral gain. Then, u_i at time step $k+1$ can be found by

$$u_i^{k+1} = u_i^k + K_{I,i} c_i^k(\lambda) \tag{2.30}$$

In the central controller the Lagrange multiplier λ is updated to control the constraints $\mathbf{g}$($\mathbf{x}$, $\mathbf{u}$, $\mathbf{d}$) to the limit of 0,

$$\Delta\lambda = \alpha \left(\sum_{i=1}^{N} g_i(x_i, u_i, d_i) - g^{\max} \right) \tag{2.31}$$

where α is the integral gain. This controller action can be implemented with a max selector to ensure the satisfaction of 2.23c. The updated Lagrange multiplier is then given by

$$\lambda^{k+1} = \lambda^k + \alpha \left(\sum_{i=1}^{N} g_i^k(x_i^k, u_i^k, d_i^k) - g^{\max} \right) \tag{2.32a}$$

$$\lambda^{k+1} = \max(0, \lambda^{k+1}) \tag{2.32b}$$

In each subsystem $c_i(\lambda)$ is controlled in the fast time-scale, in the outer loop the Lagrange multiplier λ is updated in the slow time-scale to control the coupling constraint $\mathbf{g}(\mathbf{x}, \mathbf{u}, \mathbf{d})$. Assuming that the stationary point is the local minimum, we have optimal operation of all local subsystems for a given Lagrange multiplier λ. This gives optimal performance of the overall optimization problem 2.21 as the central constraint controller 2.32 updates λ.[5] [7]

2.3.4 Dual Decomposition Framework

The structure of distributed feedback-optimizing control with dual decomposition is shown in figure 2.4.[5] The figure shows the structure of N subsystems which are inside the dashed boxes, the grey boxes represent the physical plants. The central constraint controller provides an updated Lagrange multiplier λ to the subsystems as shown in equation 2.32.

The local controllers require local cost and constraint gradient information to find the updated input. As the gradient of the Lagrange function $\nabla_u \mathcal{L}(\mathbf{x}, \mathbf{u}, \mathbf{d}, \lambda)$ is not directly measurable it is necessary to estimate this gradient from available measurements. In this paper, we use forward sensitivity analysis to estimate the gradient explained in appendix A.2.

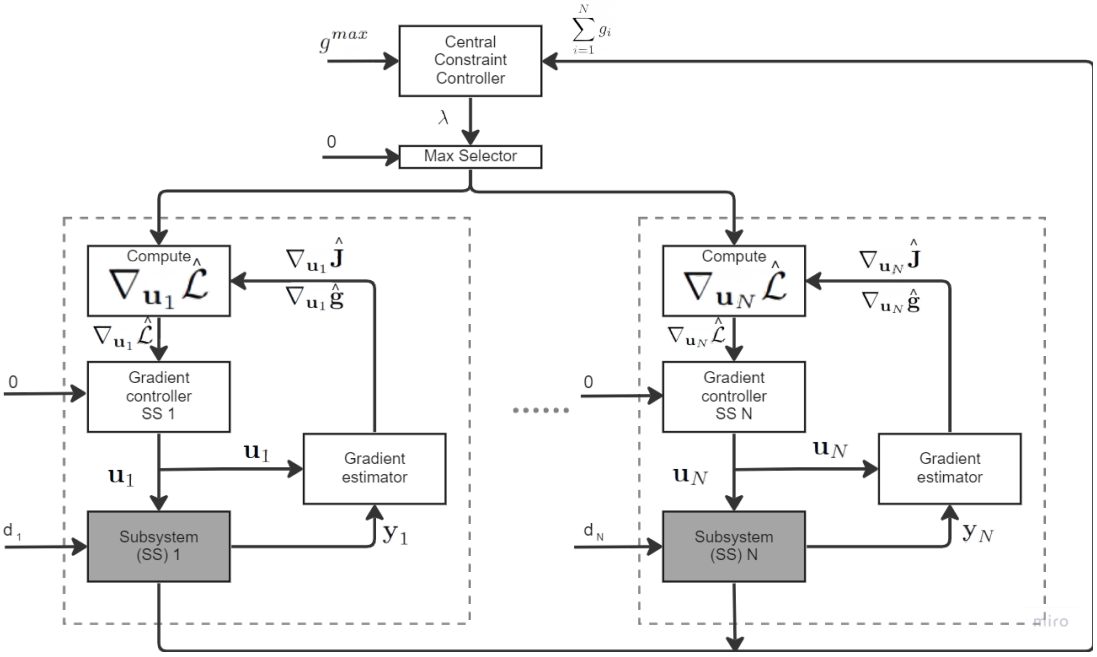

Figure 2.4: Block diagram of dual decomposition control structure.

3 Model

In this paper we test the primal and dual decomposition framework on a dynamic model of the experimental lab rig displayed in figure 3.1. This model is created in MATLAB R2021a based on a paper by Matias et al.[12] and has been tested and used in earlier papers related to this lab-rig[5]. The model equation is presented in appendix B The dynamic MATLAB model is developed using CasADi v3.5.5. A schematic of the information flow in the MATLAB model and the code can be found in appendix C.

The lab rig emulates a subsea gas-lifted oil production system with three wells, which is also the case for our simulation model. The pressure, which is equivalent to pressure measured by PI104 after the pump in the lab rig, is set to a fixed value of 1.2983 barg in the model. The reservoir valves, CV101, CV102 and CV103 in figure 3.1 acting as disturbances from the reservoir in the model by varying the valve opening, and as a consequence, the liquid flows. How the valve openings develop with time is shown in figure 4.1.

The gas lift is injected in the flow lines within the range of 1 sL/min to 5 sL/min via flow indication controllers FIC104, FIC105 and FIC106 in figure 3.1. In the MATLAB model these are not implemented as actual controllers, but are instead implemented by adding a five second delay and noise to the control input,

$$Q_{gl,i}^{k} = Q_{gl,i}^{sp,k-5} + n \tag{3.1}$$

where $Q_{gl,i}^{k}$ is the actual gas lift at time step k, $Q_{gl,i}^{sp,k-5}$ is the gas lift set-point at time step $k-5$ and n is the noise.

3.1 Optimization Problem

The objective of the optimization problem in this model setup is to maximize the liquid flow rate, which equals the sum of the liquid production of the three wells, with a limited amount of gas-lift injection. Considering problem 2.11 with the special structure of 2.12 and 2.26, the economic objective can be expressed as below,

$$\begin{aligned} \mathbf{J}(\mathbf{u},\mathbf{p}) :&= \sum_{i=1}^{3} f_i(u_i,p_i) \\ &= -20Q_{l,1}(u_1,p_1) - 25Q_{l,2}(u_2,p_2) - 30Q_{l,3}(u_3,p_3) \end{aligned} \tag{3.2}$$

where $Q_{l,1}$, $Q_{l,2}$ and $Q_{l,3}$ are the produced liquid flow rates of well 1, 2 and 3 respectively. We assume that the value of the different hydrocarbon flows varies as shown above in eq. 3.2. This is because we want to illustrate how different values affect the behavior of the subsystems. The input vector is defined as

$$\mathbf{u} = [Q_{gl,1} \quad Q_{gl,2} \quad Q_{gl,3}]^T$$

where $Q_{gl,1}$, $Q_{gl,2}$ and $Q_{gl,3}$ are the injected gas-lift flow rates of well 1, 2 and 3 respectively. Additionally, the three elements of $\mathbf{p}$, which represents the reservoir valve openings CV101, CV102 and CV103 from the lab-rig in fig. 3.1, are time varying. This implies that the cost also is a function of $\mathbf{p}$.

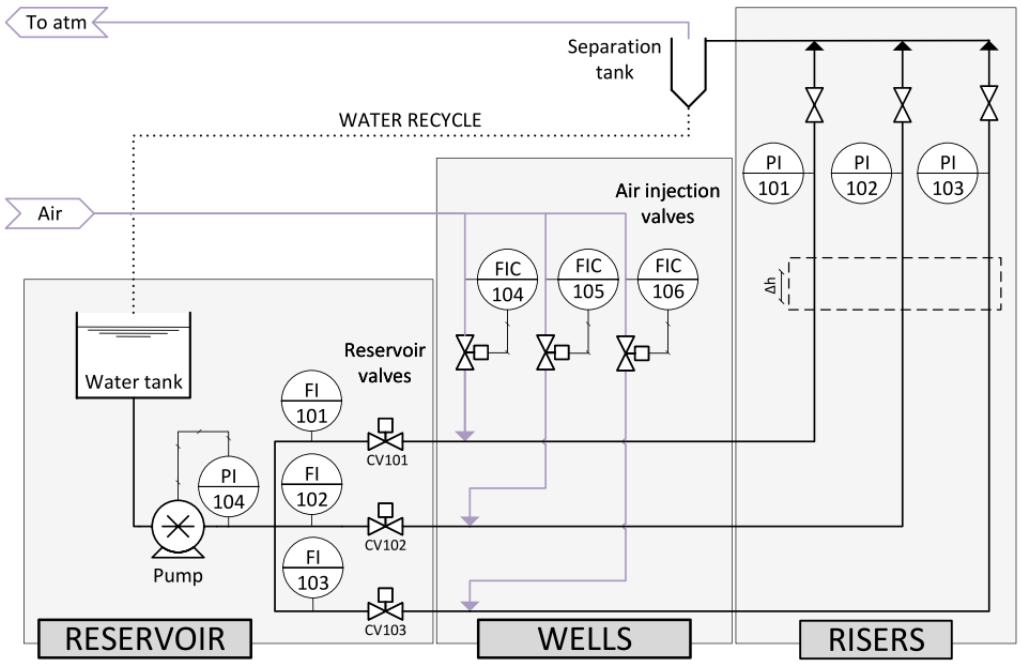

Figure 3.1: Schematic of lab rig which the model is based on. This schematic is adapted from Matias et al. [12]

The total gas availability, which in this case is a shared input constraint, can be expressed using the structure in 2.12 and 2.26

$$\begin{aligned} g(\mathbf{u},\mathbf{p}) :&= \sum_{i=1}^{3} g_i(u_i,p_i) - g^{\max} \\ &= Q_{gl,1} + Q_{gl,2} + Q_{gl,3} - Q_{gl}^{\max} \end{aligned} \tag{3.3}$$

We assume always active constraint, hence the constraint can be considered as an equality constraint. As a result of this we do not actually need the max selector 2.32b for λ update in the constraint control in the dual decomposition structure, as mentioned in section 2.3.3, this will always be positive in such cases.

3.2 Control Setup

3.2.1 Primal Decomposition Setup

Based on the problem formulation in section 2.3.1, we now implement the distributed feedback-optimizing control with online primal decomposition for our simulation model. As there are three wells in our model setup, we can decompose the problem into three subsystems. The local cost gradients and local constraint gradients, $\nabla Q_{l,i}$. and $\nabla Q_{gl,i}$, are estimated by a local gradient estimator for each subsystem. We use forward sensitivity analysis to do the gradient estimations, see appendix A.2. To derive the the local sensitivities, we need to estimate the states of the system. In this paper, we use an EKF in each individual subsystem to achieve this using only local measurements. The central constraint controller receives the estimated local Lagrange multipliers, which is derived from the local cost gradient and local constraint gradient as shown in equation 2.20. These Lagrange multipliers is then used to calculate the new setpoints, which we send to the individual subsystems. One subsystem needs to be chosen

as the compensator in the central constraint controller, and the new setpoint for this subsystem is provided by equation 2.16.

As the control structure is defined, we need to tune the central constraint controllers. The controllers are tuned using SIMC tuning rules, which is presented in section 2.1.2. In this paper, we use integral controllers, as a result the the integral gains for the central constraint controllers is given by

$$K_{I,i} = \frac{1}{K_{\lambda_i}(\tau_{c,i} + \theta_i)} \tag{3.4}$$

where $i = 1, 2, 3$ is the well index, K_{λ_i} and θ_i are the step response and the time delay of the constraints by the local Lagrange multipliers, and $\tau_{c,i}$ is the closed-loop time constants.

To determine K_{λ_i} and θ_i, the step response is analysed. Ideally we can choose the tuning parameters $\tau_{c,i} = 1$ as we want to drive the system to steady-state as fast as possible. However, this is probably to aggressive as the timescale of the central constraint controller has to be slower than the plant, which has a five second input delay as mentioned earlier. Therefore, we adjust the tuning parameters based on observation of the results and practical justification. Table 3.1 shows the obtained controller and tuning parameters for primal decomposition.

Description	Variable	Value
Simulation model sampling time	T_s	1 s
Time constant	$\tau_{c,1}$	10 s
Time constant	$\tau_{c,2}$	10 s
Time constant	$\tau_{c,3}$	10 s
Time delay	θ_{λ_1}	0 s
Time delay	θ_{λ_2}	0 s
Time delay	θ_{λ_3}	0 s
Lagrange multiplier gain	K_{λ_1}	-2.522
Lagrange multiplier gain	K_{λ_2}	-2.918
Lagrange multiplier gain	K_{λ_3}	-3.698
Integral gain	$K_{I,1}$	-0.040
Integral gain	$K_{I,2}$	-0.034
Integral gain	$K_{I,3}$	-0.027

Table 3.1: Controller and tuning parameters for primal decomposition

3.2.2 Dual Decomposition Setup

As in the section above, we implement the distributed feedback-optimizing control for our simulation model. However here we implement dual instead of primal. As for primal decomposition we can decompose the problem into three subsystems, as we have three wells in our model setup. We also have to estimate the local cost gradients and local constraint gradients, $\nabla Q_{l,i}$. and $\nabla Q_{gl,i}$, in the dual setup. The gradients for each subsystem are estimated by a local gradient estimator, which is done by forward sensitivity analysis, see appendix A.2. Each subsystem has a local gradient controller (I-controller) that controls $c_i(\lambda)$ to 0, from equation 2.28. The output from the local gradient controllers, $Q^{sp}_{gl,i}$ are the gas-lift setpoints which is provided to the simulation model. For the central constraint controller, we update the Lagrange multiplier as shown in equation 2.32, As the constraint $\mathbf{g}$ can be directly measured there is no need for any estimations here. As seen in figure 2.4 the dual decomposition has a cascade structure, where the central constraint controller act as the master and the gradient controllers are the slaves.

Once the dual decomposition structure is defined, we have to tune the controllers. The simulation model has a sampling rate of 1 s, therefore it is possible to execute the central constraint controller and gradient controllers at the same rate. However, they might compete against each other, depending on their tuning, this could lead to system instability. As a result of this, it is necessary with a time scale separation between these controllers.

The convergence rate to the setpoint is given by the closed-loop time constant τ_{c_i} of the control loop for each subsystem. For a linear first-order system, the approach to steady-state is given by $(1-e^{-\tau_{c,\lambda}/\tau_{c_i}})$ where $\tau_{c,\lambda}$ is the convergence time of the central constraint controller given in equation 2.32, and τ_{c_i} is the closed-loop time constant of gradient control in subsystem i. Thus, for $\tau_{c,\lambda}/\tau_{c_i} = 5$ we have reached 99.3 % of convergence, and is therefore considered reached. This could be regarded as the rule of thumb of having a time scale separation between control layers of at least 5.[15] Larger value would give more robust control in terms of process gain variations. However, for large values the controllers will converge slow, for practical reasons, a value of time scale separation of 5 to 10 is often recommended.

In this paper, we use integral controllers for both the central constraint control and gradient controllers. These are tuned using SIMC tuning rules, which is presented in section 2.1.2. The integral gain for the central constraint controller is given by

$$\alpha = \frac{1}{K_\lambda(\tau_{c,\lambda} + \theta_\lambda)} \tag{3.5}$$

where K_λ is the step response and θ_λ is the time delay of the constraint by the Lagrange multiplier. $\tau_{c,\lambda}$ is the closed-loop time constant that governs the evolution of the constraints. The integral gain for the three local gradient controllers is given by

$$K_{I,i} = \frac{1}{K_{\mathbf{u}_i}(\tau_{c,\mathbf{u}_i} + \theta_{\mathbf{u}_i})} \tag{3.6}$$

where $i = 1, 2, 3$ is the well index, $K_{\mathbf{u}_i}$ and $\theta_{\mathbf{u}_i}$ are the step response and time delay of the gradient by the inputs (gas-lift setpoints). $\tau_{c,\mathbf{u}_i}$ is the closed-loop time constant that governs th evolution of $c_i(\lambda)$.

To determine K_λ, θ_λ, $K_{\mathbf{u}_i}$ and $\theta_{\mathbf{u}_i}$ the step response is analysed. While the tuning parameters $\tau_{c,\lambda}$ and $\tau_{c,\mathbf{u}_i}$ should consider the concept of time scale separation where $\epsilon_i = \frac{\tau_{c,\mathbf{u}_i}}{\tau_{c,\lambda}}$ should be around 0.2, or at least less than 1, this implies that the outer loop is slower than the inner loop. The inner time constant is adjusted based on observation of the results and pratical justification, based on the same arguments as for the tuning parameter in primal decomposition, see section 3.2.1. Table 3.2 shows the obtained controller and tuning parameters for dual decomposition.

Description	Variable	Value
Simulation model sampling time	T_s	1 s
Central constraint controller		
Time constant	$\tau_{c,\lambda}$	50 s
Time delay	θ_λ	0 s
Lagrange multiplier gain	K_λ	0.907
Integral gain	α	0. 022
Local gradient controllers		
Time constant	τ_{c,u_1}	10 s
Time constant	τ_{c,u_2}	10 s
Time constant	τ_{c,u_3}	10 s
Time delay	θ_{u_1}	0 s
Time delay	θ_{u_2}	0 s
Time delay	θ_{u_3}	0 s
Input gain	K_{u_1}	2.53
Input gain	K_{u_2}	2.93
Input gain	K_{u_3}	3.70
Integral gain	$K_{I,1}$	0.040
Integral gain	$K_{I,2}$	0.034
Integral gain	$K_{I,3}$	0.027

Table 3.2: Controller and tuning parameters for dual decomposition

4 Results

Fig. 4.1 shows the disturbance in this simulation, which corresponds to the reservoir valve openings CV101 (p_1), CV102 (p_2), CV103 (p_3) in the lab-rig the simulation model is based on. The first disturbance occurs when p_1 gradually decreases from $t = 6$ to $t = 12.5$ min. During this time period, we expect the gas-lift injection in well 1 to decrease, and redistribution of the gas to the other wells. The second disturbance occurs when p_3 gradually decreases from $t = 14$ to $t = 18$ min. As before, we expect that the gas-lift injection rate in well 3 will go down. At the same time, we expect the other wells will gain a higher gas-lift injection. The third and fourth disturbance occur from $t = 21$ to $t = 27.5$ and $t = 31$ to $t = 35$ respectively. During this time p_1 and p_3 are gradually increased back up to the initial values. This is because we want to see how the controllers behave for both decrease and increase in the disturbance.

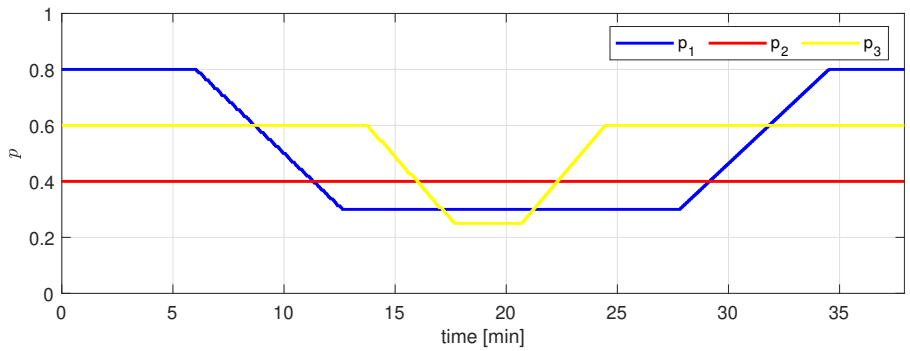

Figure 4.1: Disturbances

4.1 Determine Compensator Subsystem for Primal Decomposition Framework

Figure 4.2 shows a comparison of accumulated profit with the three different subsystems chosen as compensator. The profit difference is calculated as

$$P_{diff}^{k} = \frac{P_{3}^{k} - P_{i}^{k}}{P_{i}^{k}} \cdot 100 \tag{4.1}$$

where P_{diff}^{k} is the accumulated profit difference, P_{3}^{k} is the accumulated profit with well 3 chosen as compensator, P_{i}^{k} is the accumulated profit with well i chosen as compensator ($i = 1, 2$) at time k. The accumulated profit at time k with the different wells as compensator is calculated as

$$P^{k} = \sum_{j=1}^{k-1} P^{j} \tag{4.2}$$

It is clear from the plot that we obtain the best performance with subsystem 3 as compensator, although the difference between different compensators is modest. The difference in performance stems from the gains of the controllers in each subsystem, as shown in table 3.1 subsystem 3 has the highest gain magnitude. Hence, we conclude that for cases with active input shared constraint that the subsystem with highest gain magnitude should be selected as compensator.

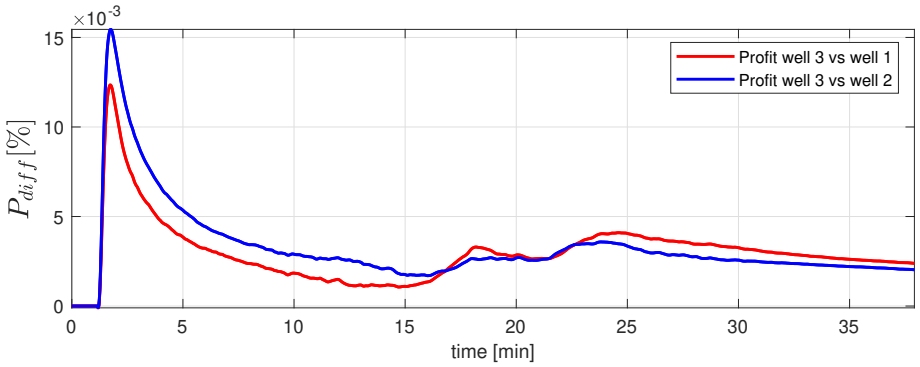

Figure 4.2: Comparison of accumulated profit with different compensator wells. Well 3 vs well 1 and well 3 vs well 2 denotes the comparison of accumulated profit with well 3 as compensator versus accumulated profit with well 1 and 2 as compensator respectively.

4.2 Comparison of Feedback-Optimizing Control

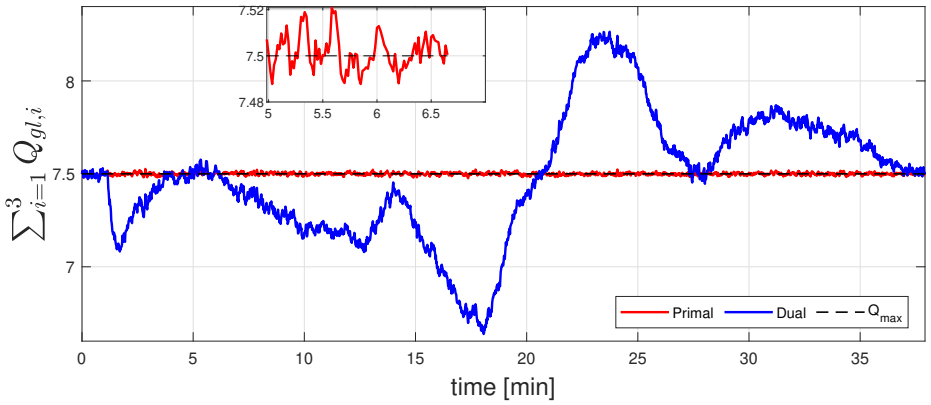

Figure 4.3: The constraint satisfaction (total implemented gas-lift) of both primal and dual decomposition. There is a magnifying plot in time window 5 to 6.5 min showing only constraint satisfaction of primal decomposition.

In figures 4.3 - 4.5 we compare the simulation results from Primal decomposition and dual decomposition. Figure 4.3 shows the constraint satisfaction. As expected, the primal decomposition performs much better than the dual decomposition here. This is because the compensator system, subsystem 3, "absorbs" the deviations from the constraint. The absorbing ability comes from how the compensator set-point is calculated, see eq. 2.16, and is the reason for the primal decomposition's capability to maintain the constraint active. In terms of what this means for the operation, with the use of primal decomposition we can run the system without any significant back off and still remain feasible at all times. On the other hand, for the dual decomposition we must implement back off, especially from $t = 21$ when the disturbances start to increase again. These constraint violations is a result of the necessary time-scale separation between the central constraint control and local gradient controllers. Because of the slower nature of the constraint control it does not manage to keep up with the disturbances to the same level as the primal decomposition.

In figure 4.4 the gas-lift setpoints given from the decomposition structures to the lab-rig model. As the dual decomposition violates the constraint, as seen above, one would expect that the dual decomposition have slower response in the setpoint changes than for the primal decomposition. With the disturbance behavior in this simulation, this is the case for two out of the three wells. In fact, in figure 4.4 from $t = 6$ to $t = 12.5$ when there is a disturbance in well 1, we can see that the dual decomposition provide a faster setpoint response for subsystem 1 than primal decomposition. While, for the other two subsystems primal decomposition has the faster response. This behavior is rooted in the gradient controllers in the dual decomposition framework. When there is a disturbance in one of the subsystem the local gradient controller response to this because equation 2.28 is affected by the local disturbance. The remaining local gradient controllers are only affected by their local disturbance, therefore there is no response here before the central constraint control responds to the constraint violation. This is due to the time scale separation needed between the central constraint control and local gradient controllers in the dual decomposition framework, which is explained in section 3.2.2. In comparison, for the primal decomposition structure all subsystems have response in the setpoint control simultaneously. As a result we have faster disturbance rejection in the local subsytems for dual decomposition, however, primal decomposition have better constraint control.

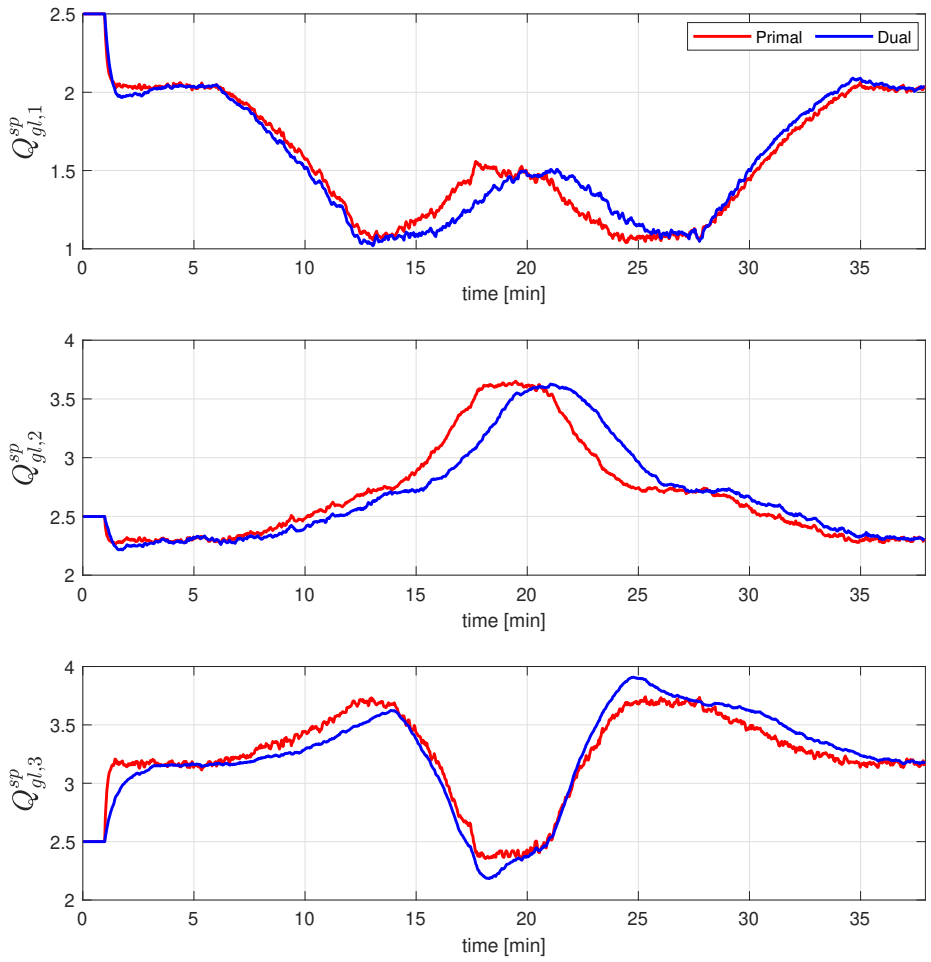

Figure 4.4: The gas-lift flow rate setpoints ($\mathbf{u}^{sp} = \mathbf{Q}_{gl}^{sp}$) for every well for both primal and dual decomposition control method.

The actual gas-lift flow rate is displayed in fig. 4.5, this differs slightly from the calculated input setpoints shown in fig. 4.4 due to the implemented measurement noise in the model. Another reason is how the gas flow rate controllers are implemented in the model, which is described earlier in the paper.

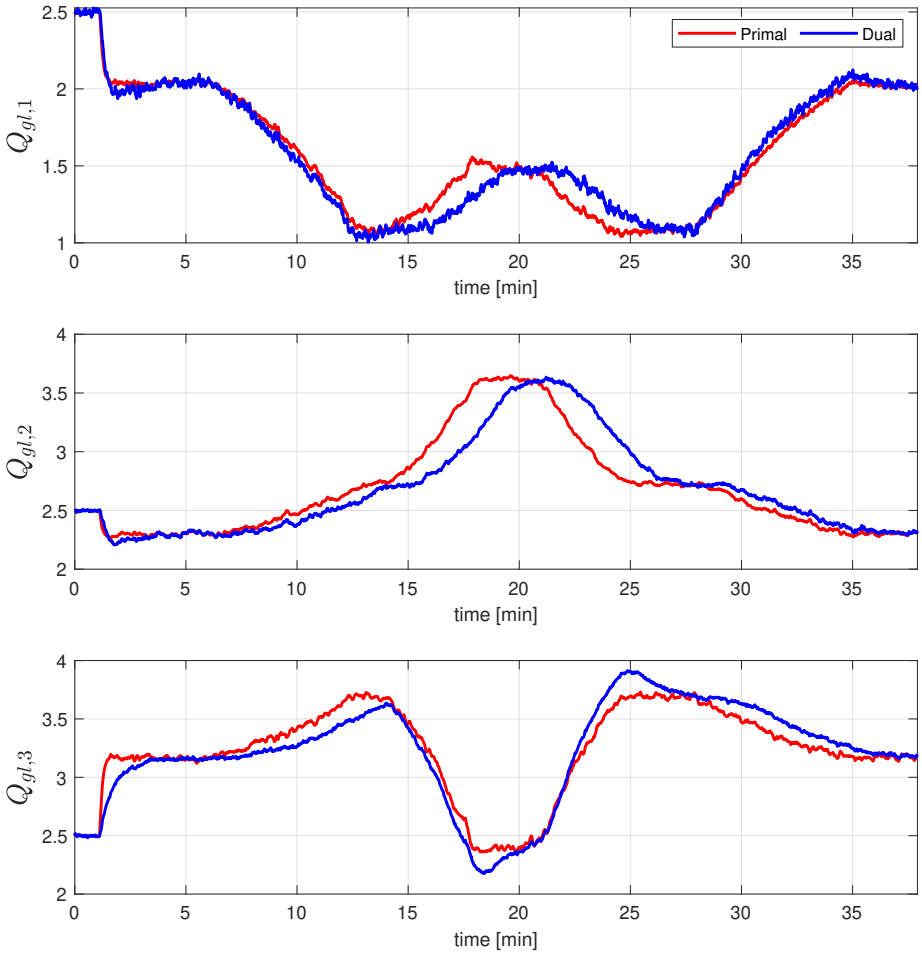

Figure 4.5: The actual implemented gas-lift flow rates in every well for both primal and dual decomposition control method.

In figures 4.6 - 4.8 the Lagrange multipliers for both decomposition methods are displayed. Figure 4.6 shows that the control of the Lagrange multipliers for primal decomposition is very fast and converges to the same value. However, it is quite noisy. This is due to the fast time scale which the central constraint controller in the primal decomposition structure operates in. In figure 4.7 the Lagrange multipliers are displayed without measurement noise to show clearly the fast converge rate of the primal decomposition, we can also see that the Lagrange multipliers reach the same value when steady-state operation is achieved.

Figure 4.8 show the Lagrange multiplier for the dual decomposition. We can observe that the multiplier converges slower for this method. This is due to the tuning of the controllers and the fact that the central controller for dual decomposition operates on a different timescale. When the Lagrange multiplier converges around $t = 4$ min, we can see that the active constraint is

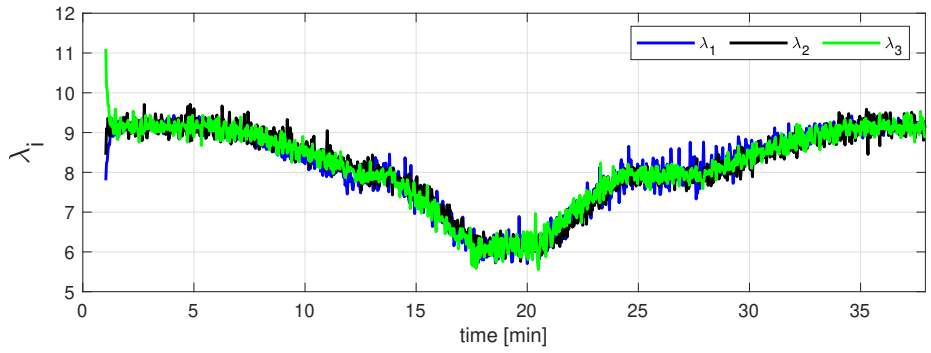

Figure 4.6: The local Lagrange multipliers for primal decomposition.

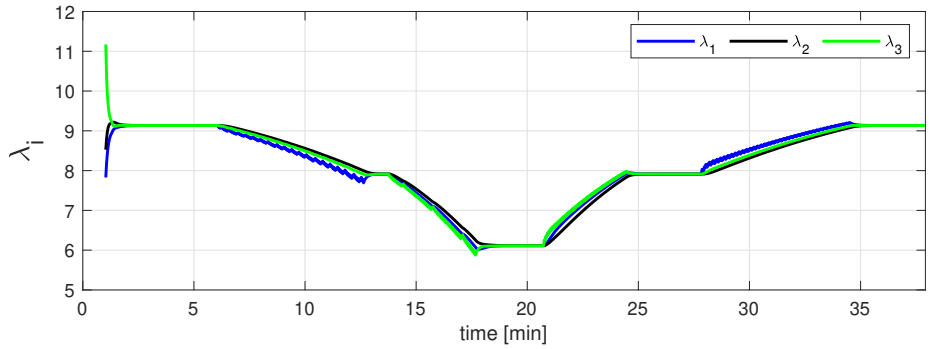

Figure 4.7: The local Lagrange multipliers for primal decomposition with no measurement noise.

controlled almost as good as for the primal case in figure 4.3. It is also worth noting that the Lagrange multipliers converge to the same value for both primal and dual. As mentioned in section 2.3.3, when always active constraint is considered it correspond to a nonzero Lagrange multiplier. In figure 4.6 and 4.8 we can confirm this statement, as the Lagrange multipliers are always positive during the simulation period.

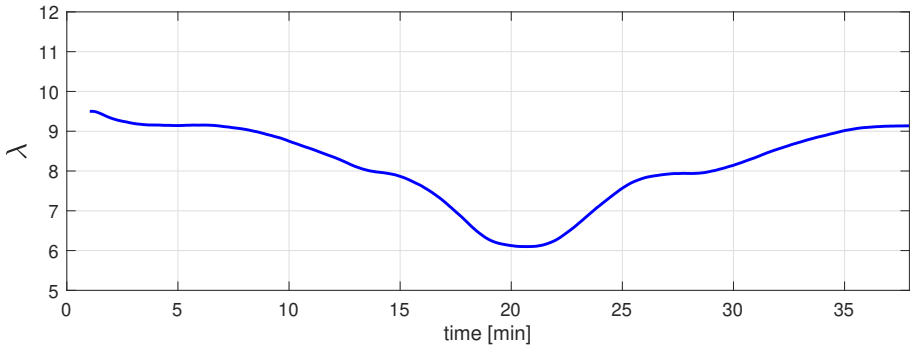

Figure 4.8: Lagrange multiplier for dual decomposition.

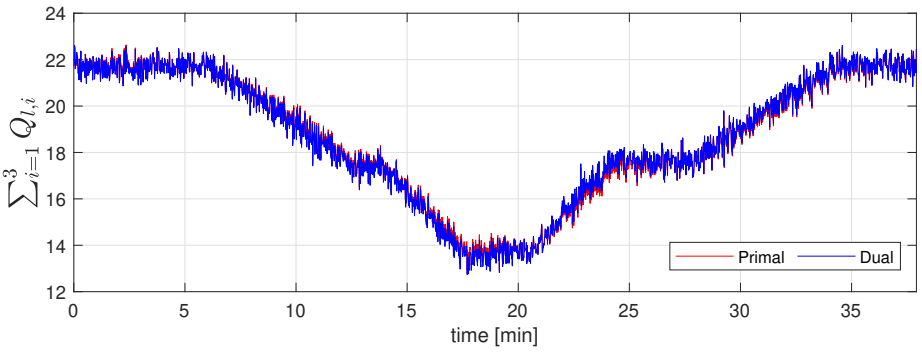

Figure 4.9: Comparison of the total liquid flow in the simulation model

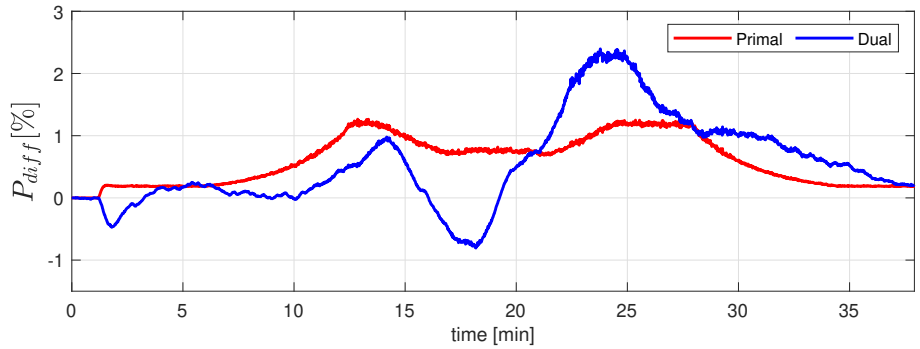

Figure 4.10: Instantaneous profit of primal and dual decomposition compared to a naive approach.

In figure 4.9 the total production of the two methods are shown, however, it is not possible to gather any valuable information on the two methods compared to each other. To better display the performance of the two methods, we have compared them to a naive approach in figure 4.10. Here the difference in percentage between the instantaneous profit of both methods and the naive approach is presented. In the naive approach we consider fixed inputs,

$$\mathbf{u} = \begin{bmatrix} \frac{Q_{gl}^{max}}{3} & \frac{Q_{gl}^{max}}{3} & \frac{Q_{gl}^{max}}{3} \end{bmatrix}^T \tag{4.3}$$

which represent the case where no information is available, hence the best approach is to divide the available gas equally among the wells. The difference in figure 4.10 is calculated as

$$P_{diff} = \frac{P - P_{naive}}{P_{naive}} \cdot 100 \tag{4.4}$$

where P is the profit of the method of interest, and P_{naive} is the profit of the naive approach. The results show that primal decomposition is more profitable than dual decomposition until t = 21 min, after this dual decomposition appears to be favorable. However, in figure 4.3 we see that after t = 21 min dual decomposition does not achieve primal feasibility, and therefore the profit here is not valid.

In this paper, we have only considered the case where the constraint always is active. However, if we have a change from active constraint to unconstrained the implemented primal decomposition framework would not handle this well. This is because of how the compensator subsystem is controlled, as this always utilize the remaining available gas-lift. The dual decomposition on the other hand, would be able to reach optimal operation in this case because

of the max selector, see equation 2.32b. As a result, the primal decomposition structure implemented in this paper is specific for the always active input shared constraint case, while the dual decomposition structure is more general.

The economical performance of the dual decomposition framework is limited because of the back-off necessary to avoid constraint violation. One solution to this is to implement dual with constraint override. This way, it is possible to ensure always feasible operation without any economic sacrifice.

5 Conclusion

In this work, we have done simulation with uncertainty and measurement noise to compare feedback-based real-time optimization with online primal and dual decomposition. Based on the results we can conclude both methods manage to optimize the subsea gas-lifted oil production problem without the use of any numerical solver.

However, it is shown that primal decomposition framework is able to more effectively ensure primal feasibility for a separable system with an active shared input constraint than the dual decomposition framework. This is mainly because of the necessary timescale difference between the central constraint controller and local gradient controllers in the dual decomposition structure. As a result, the primal decomposition framework can reduce the need for any back-off significantly, which results in more profitable operation.

5.1 Further Work

In the results, it was shown that the dual decomposition framework has a faster local disturbance rejection in the respective subsystem. Therefore it would be interesting to consider disturbances in multiple subsystems simultaneously and compare the economic performance of the two decomposition structures.

To reduce the requirement of back-off in dual decomposition, we suggest to implement dual decomposition with constraint override in order to handle the constraint better.

As the continuation of this work, we consider obtaining the experimental result in order to validate the results in this paper.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[2] S. Boyd, L. Xiao, A. Mutapcic, and J. Mattingley. Notes on decomposition methods. *Notes for EE364B, Stanford University*, 635:1–36, 2007.

[3] C. Y. Chen and B. Joseph. On-line optimization using a two-phase approach: An application study. *Industrial & engineering chemistry research*, 26(9):1924–1930, 1987.

[4] G. Cohen and G. Coon. Theoretical consideration of retarded control. *Transactions of the American Society of Mechanical Engineers*, 75(5):827–834, 1953.

[5] R. Dirza, J. Matias, S. Skogestad, and D. Krishnamoorthy. Experimental validation of distributed feedback-based real-time optimization in a gas-lifted oil well rig. *Control Engineering Practice*, 126:105253, 2022.

[6] R. Dirza, M. Rizwan, S. Skogestad, and D. Krishnamoorthy. Real-time optimal resource allocation using online primal decomposition. *IFAC-PapersOnLine*, 55(21):31–36, 2022.

[7] R. Dirza, S. Skogestad, and D. Krishnamoorthy. Optimal resource allocation using distributed feedback-based real-time optimization. *IFAC-PapersOnLine*, 54(3):706–711, 2021.

[8] R. A. Jose and L. H. Ungar. Pricing interprocess streams using slack auctions. *AIChE Journal*, 46(3):575–587, 2000.

[9] M. King. *Process control: a practical approach*. John Wiley & Sons, 2016.

[10] D. Krishnamoorthy. A distributed feedback-based online process optimization framework for optimal resource sharing. *Journal of Process Control*, 97:72–83, 2021.

[11] D. Krishnamoorthy, M. A. Aguiar, B. Foss, and S. Skogestad. A distributed optimization strategy for large scale oil and gas production systems. In *2018 IEEE Conference on Control Technology and Applications (CCTA)*, pages 521–526. IEEE, 2018.

[12] J. Matias, J. P. Oliveira, G. A. Le Roux, and J. Jäschke. Steady-state real-time optimization using transient measurements on an experimental rig. *Journal of Process Control*, 115:181–196, 2022.

[13] M. Morari, Y. Arkun, and G. Stephanopoulos. Studies in the synthesis of control structures for chemical processes: Part i: Formulation of the problem. process decomposition and the classification of the control tasks. analysis of the optimizing control structures. *AIChE Journal*, 26(2):220–232, 1980.

[14] S. Skogestad. Simple analytic rules for model reduction and pid controller tuning. *Journal of process control*, 13(4):291–309, 2003.

[15] S. Skogestad and I. Postlethwaite. *Multivariable feedback control: analysis and design*. john Wiley & sons, 2005.

[16] G. Stojanovski, L. Maxeiner, S. Krämer, and S. Engell. Real-time shared resource allocation by price coordination in an integrated petrochemical site. In *2015 European Control Conference (ECC)*, pages 1498–1503. IEEE, 2015.

[17] E. Walter, L. Pronzato, and J. Norton. *Identification of parametric models from experimental data*, volume 1. Springer, 1997.

[18] J. G. Ziegler, N. B. Nichols, et al. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

A Steady-State Gradient Estimation

In this paper forward sensitivity analysis is used to estimate gradients. The gradient estimation has to steps. The current plant (model) information is used to update the state and parameters of the model with use of a extended Kalman Filter. Then, the updated model is used to compute the stady-state gradients with the forward sensistivity analysis. This model - model parameter estimation is done because we want to implement the code used in this paper to a lab-rig setup.

A.1 Extended Kalman Filter

The model have to be linearized in order to use the Kalman filter equations. The model is an index-1 differential algebraic equation (DAE) system and can easily be re-arranged to an ordinary differential equation (ODE). The unknown parameters are assumed to be time-varying and a random walk model was used to decide their dynamics,

$$\mathbf{p}^{k+1} = \mathbf{p}^k + \mathbf{v}^k \tag{A.1}$$

where $\mathbf{v}^k$ follows a normal distribution with covariance V_θ and mean zero. We also assume that $\mathbf{v}^k$ are independent of $\mathbf{v}^{\neq k}$.

An extended model that we use for parameter estimation is obtained by combining the parameter and system dynamics. We can apply the extended Kalman filter equations for estimating $\mathbf{p}^k$, $\mathbf{x}^k$ and $\mathbf{z}^k$ simultaneously because the model was linearized. For a complete derivation of the EKF equations the reader is reffered to the work of Walter and Pronzato [17].

A.2 Forward Sensitivity Analysis

The original nonlinear DAE model is in the form

$$\begin{aligned} \mathbf{x}^{k+1} &= \check{F}(\mathbf{x}^k, \mathbf{z}^k, \mathbf{u}^k, \mathbf{p}^k) \\ 0 &= \check{G}(\mathbf{x}^k, \mathbf{z}^k, \mathbf{u}^k, \mathbf{p}^k) \end{aligned} \tag{A.2}$$

The stationary value of forward sensitivity equation are used to estimate the steady-state gradients

$$\begin{aligned} 0 =& \frac{\delta \check{F}^\top}{\delta \mathbf{x}} \mathbf{S}_{SS} + \frac{\delta \check{F}^\top}{\delta \mathbf{z}} \mathbf{R}_{SS} + \frac{\delta \check{F}^\top}{\delta \mathbf{u}} \\ 0 =& \frac{\delta \check{G}^\top}{\delta \mathbf{x}} \mathbf{S}_{SS} + \frac{\delta \check{G}^\top}{\delta \mathbf{z}} \mathbf{R}_{SS} + \frac{\delta \check{G}^\top}{\delta \mathbf{u}} \end{aligned} \tag{A.3}$$

where $\mathbf{S}_{SS}$ and $\mathbf{R}_{SS}$ are the sensitivities of the differential states $\mathbf{x}$ and algebraic states $\mathbf{z}$ with right to the inputs $\mathbf{u}$. In our case, the objective $\mathbf{J}$ and constraint $\mathbf{g}$ are linear functions of the algebraic states, $\mathbf{J} = \check{\mathbf{H}}_\mathbf{J} z$ and $\mathbf{g} = \check{\mathbf{H}}_\mathbf{g} z$. Therefore, the chain rule can be used to obtain $\nabla_\mathbf{u}\mathbf{J}$ and $\nabla_\mathbf{u}\mathbf{g}$,

$$\begin{aligned} \mathbf{J} = \check{\mathbf{H}}_\mathbf{J} z \Rightarrow \nabla_\mathbf{u}\mathbf{J} = \check{\mathbf{H}}_\mathbf{J} \mathbf{R}_{SS} \\ \mathbf{g} = \check{\mathbf{H}}_\mathbf{g} z \Rightarrow \nabla_\mathbf{u}\mathbf{g} = \check{\mathbf{H}}_\mathbf{g} \mathbf{R}_{SS} \end{aligned} \tag{A.4}$$

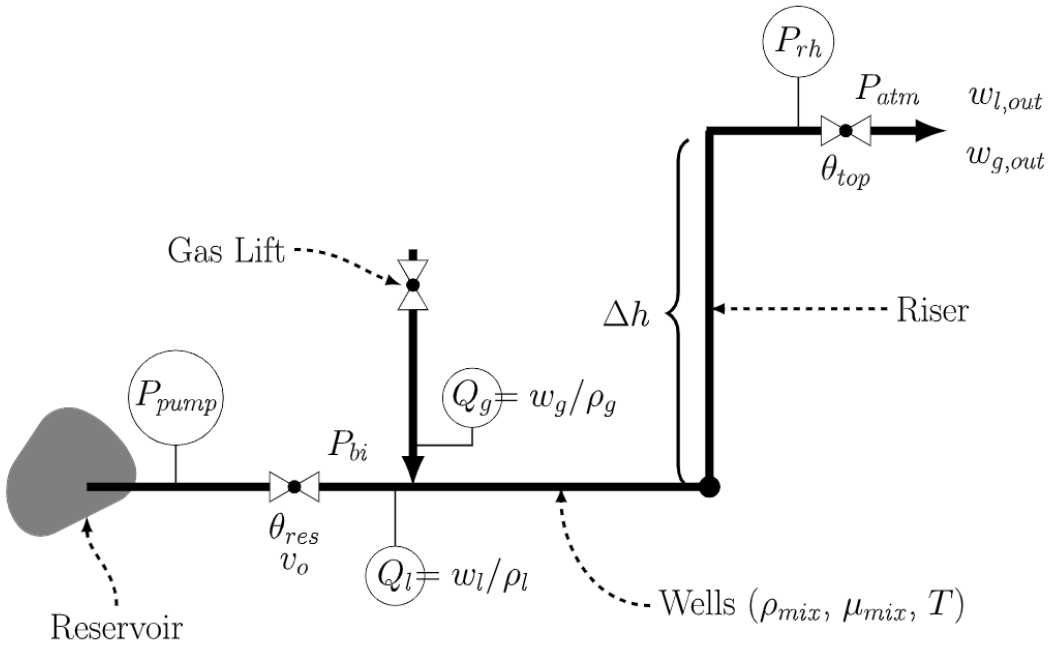

Figure B.1: Diagram of a single well model. Q_l, P_{rh} and P_{pump} are measured. The reservoir valve opening v_0 is assumed as a measured disturbance and Q_g is controlled by the gas flowrate controller.

B Dynamic Lab-Rig Model

The dynamic lab-rig model used in this paper is presented below, it was derived by Matias et. al[12]. Here we only show the equations for one well, extending the model to three wells is straightforward. A diagram of the model is shown in figure B.1.

The differential equations in the model i.e. the liquid and gas mass balances are represented by

$$\dot{m}_g = w_g - w_{g,out} \tag{B.1}$$
$$\dot{m}_l = w_l - w_{l,out} \tag{B.2}$$

where m_g and m_l are the gas and liquid mass holdup inside the well and riser. The time derivative is represented by the dot symbol. w_l is the liquid flowrate from the reservoir and w_g is the gaslift injection mass flowrate. $w_{l,out}$ and $w_{g,out}$ are the outlet production rate of liquid and gas.

The algebraic equations in the model are presented below. The outflow of the reservoir is obtained by the following relationship

$$w_l = v_0 \theta_{res} \sqrt{\rho_l (P_{pump} - P_{bi})} \tag{B.3}$$

where v_0 is the reservoir valve opening, θ_{res} is the valve flow coefficient, ρ_l is the liquid density. P_{pump} is the pressure at the pump outlet and the pressure before the injection point P_{bi} is computed with the hydrostatic pressure and the pressure drop due to friction taken into account. The Darcy-Weisbach expression for laminar flow in cylindrical pipes is used as a simplification. Thus, P_{bi} becomes

$$P_{bi} = P_{rh} + \rho_{mix} g \Delta h + \frac{128 \mu_{mix} (w_g + w_l) L}{\pi \rho_{mix} D^4} \tag{B.4}$$

where P_{rh} is the riser head pressure, D, L and Δh are the diameter, length (well + riser) and height of the pipes. g is the gravitational acceleration. μ_{mix} is the mixture viscosity, which is approximated as the liquid viscosity. The mixture density ρ_{mix} is obtained by

$$\rho_{mix} = \frac{m_{total}}{V_{total}} = \frac{m_g + m_l}{V_{total}} \tag{B.5}$$

where V_{total} is the summation of the gas V_g and liquid V_l volumetric holdup

$$V_{total} = V_g + V_l = \frac{m_g}{\rho_g} + \frac{m_l}{\rho_l} \tag{B.6}$$

It is assumed constant liquid density ρ_l, the gas density ρ_g is computed using the ideal gas law

$$\rho_g = \frac{P_{bi} M_g}{RT} \tag{B.7}$$

where M_g is the molecular weight of air, T is the room temperature and R is the gas universal constant. The total outlet flowrate is obtained by

$$w_{total} = w_{g,out} + w_{l,out} = \theta_{top} \sqrt{\rho_{mix} (P_{rh} - P_{atm})} \tag{B.8}$$

where θ_{top} is the flow coefficient of the top valve and P_{atm} is the atmospheric pressure. It is also assumed that the liquid fraction in the mixture, α_l, is the same as the proportion between liquid and total outlet flowrate.

$$\alpha_l = \frac{w_{l,out}}{w_{total}} = \frac{m_l}{m_{total}} \tag{B.9}$$

C MATLAB Code

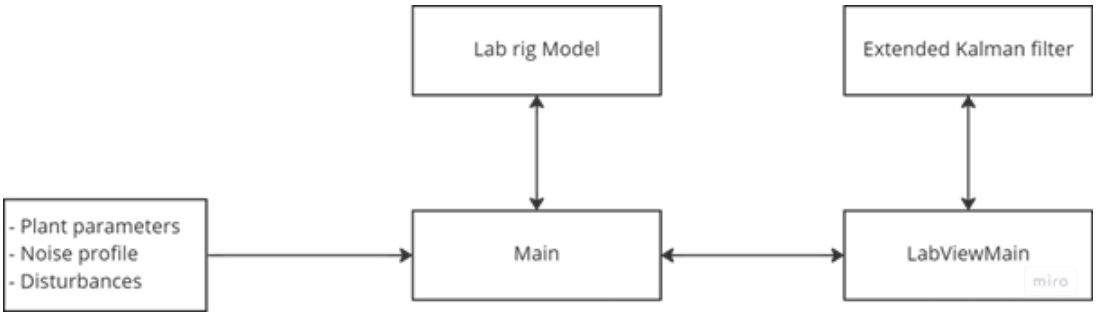

Figure C.1: Information flow in MATLAB code

C.1 LabViewMain - Primal Decomposition

```
% Main program
% Run Initialization file first

%%%%%%%%%%%%%%%%
% Get Variables
%%%%%%%%%%%%%%%%
% disturbances
%valve opening [%]
cv101 = P_vector(1);
cv102 = P_vector(2);
cv103 = P_vector(3);
% cv101 = 0.8; cv102 = 0.4; cv103 = 0.6;
% if value is [A] from 0.004 to 0.020
% if you want to convert to 0 (fully closed) to 1 (fully open)
% vo_n = (vo - 0.004)./(0.02 - 0.004);

%pump rotation [%]
pRate = P_vector(4);
% if value is [A] from 0.004 to 0.020
% if you want to convert to (min speed - max speed)
% goes from 12% of the max speed to 92% of the max speed
% pRate = 12 + (92 - 12)*(P_vector(4) - 0.004)./(0.02 - 0.004);

% always maintain the inputs greater than 0.5
% inputs computed in the previous MPC iteration
% Note that the inputs are the setpoints to the gas flowrate PID's
fic104sp = P_vector(5);
fic105sp = P_vector(6);
fic106sp = P_vector(7);
%current inputs of the plant
u0old=[P_vector(5);P_vector(6);P_vector(7)];


% cropping the data vector
nd = size(I_vector,2);
dataCrop = (nd - BufferLength + 1):nd;

% liquid flowrates [L/min] %INCLUDE NOISE???
fi101 = I_vector(1,dataCrop);
fi102 = I_vector(2,dataCrop);
fi103 = I_vector(3,dataCrop);
```

```
% actual gas flowrates [sL/min] %INCLUDE NOISE???
fic104 = I_vector(4,dataCrop);
fic105 = I_vector(5,dataCrop);
fic106 = I_vector(6,dataCrop);

% pressure @ injection point [mbar g]
pi105 = I_vector(7,dataCrop);
pi106 = I_vector(8,dataCrop);
pi107 = I_vector(9,dataCrop);

% reservoir outlet temperature [oC]
ti101 = I_vector(10,dataCrop);
ti102 = I_vector(11,dataCrop);
ti103 = I_vector(12,dataCrop);

% DP @ erosion boxes [mbar]
dp101 = I_vector(13,dataCrop);
dp102 = I_vector(14,dataCrop);
dp103 = I_vector(15,dataCrop);

% top pressure [mbar g]
% for conversion [bar a]-->[mbar g]
% ptop_n = ptop*10^-3 + 1.01325;
pi101 = I_vector(16,dataCrop);
pi102 = I_vector(17,dataCrop);
pi103 = I_vector(18,dataCrop);

% reservoir pressure [bar g]
% for conversion [bar g]-->[bar a]
% ptop_n = ptop + 1.01325;
pi104 = I_vector(19,dataCrop);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CODE GOES HERE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% number of measurements in the data window
dss = size(pi104,2);

% we do no consider SS detection here
flagSS = 1;

% check for first iteration
if ~exist('dxHatkk','var')
    %initial condition
    [dx0,z0,u0,theta0] = InitialConditionGasLift(par);

    dxHatkk = dx0;
    zHatkk = z0;
    thetakk = theta0;

    load('EKFconf');
    qThres = ekf.qThresk;
    Pkk = ekf.Pkk;
    ekf.R = noise.output; %added (different from Rig Implementation)

    lambda = 9.5;

    lambda_1 = 1;
```

```
    lambda_2 = 1;
    lambda_3 = 1;

    err0 = zeros(3,1);
end

if flagSS == 1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Estimating SS model parameters (dynamic) %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    yPlant = [fi101;
              fi102;
              fi103;
              1.01325 + 10^-3*pi101; %[mbarg]-->[bar a];
              1.01325 + 10^-3*pi102;
              1.01325 + 10^-3*pi103];

    uPlant = [fic104; %conversion [L/min] --> [kg/s]
              fic105;
              fic106;
              cv101*ones(1,dss); %workaround - i just have the last measurement here. Since it is the distu
              cv102*ones(1,dss);
              cv103*ones(1,dss);
              pi104 + 1.01325];

          try
              % running casadi - getting the last measurement and last
              % input that generated that measuremnt
              [dxHatkk,zHatkk,thetakk,Pkk,qThres] = RExtendedKalmanFilter(yPlant(:,end),uPlant(:,end - 10),

              % everything normal
              flagEst = 1;
          catch
              warning('Dynamic Estimation Problem!');
              beep
              % estimation problem
              flagEst = 0;

              % Note that in this case, we dont update
              % dxHatkk,zHatkk,thetakk,Pkk,qThres
          end


    if flagEst == 1
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Optimizing Systems %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        % Gradient Estimation
        %conversion
        CR = 60*10^3; % [L/min] -> [m3/s]

        %%% NEED TO CHECK
        %F_zp = full(S_zp(dxHatkk,zHatkk,[uPlant(:,end);thetakk;1e4]));

        tStart = cputime;
        F_zp = full(S_zp(dxHatkk,zHatkk,[uSPPlantArray(1:3,end);uPlant(4:end,end);thetakk;1e4]));

        J_qgl1 = - 20*((1*1e-2)*CR/par.rho_o(1))*F_zp(22,1);
```

```
J_qgl2 = - 25*((1*1e-2)*CR/par.rho_o(2))*F_zp(23,2);
J_qgl3 = - 30*((1*1e-2)*CR/par.rho_o(3))*F_zp(24,3);

g_qgl1 = 1;
g_qgl2 = 1;
g_qgl3 = 1;

%------- Calculating lambda's -------
lambda_1 = -(J_qgl1)/g_qgl1;
lambda_2 = -(J_qgl2)/g_qgl2;
lambda_3 = -(J_qgl3)/g_qgl3;

Lu(:,kk) = [lambda_1;lambda_2;lambda_3];

%------ g_i --------
%choose compensator
well_comp = 3;
lambda_c = Lu(well_comp, end);

%------  Controller Tunings  --------
%---------------  1  ----------------

K_1 = -2.522;
tauC_1 = 10;
theta_1 = 0;

Ki1 = 1/(K_1*(tauC_1 + theta_1));

%---------------  2  ----------------

K_2 = -2.918;
tauC_2 = 10;
theta_2 = 0;

Ki2 = 1/(K_2*(tauC_2 + theta_2));

%---------------  3  ----------------

K_3 = -3.698;
tauC_3 = 10;
theta_3 = 0;

Ki3 = 1/(K_3*(tauC_3 + theta_3));

%----- Primal Decomposition Controller -----

if well_comp == 1
    g2new = u0old(2) + Ki2*(-lambda_2 + lambda_c);
    u0pt(2,1) = max(1.0,min(7.5, g2new));
    g3new = u0old(3) + Ki3*(-lambda_3 + lambda_c);
    u0pt(3,1) = max(1.0,min(7.5, g3new));
    g1new = 7.5 - (u0pt(2,1) + u0pt(3,1));
    u0pt(1,1) = max(1.0,min(7.5, g1new));
end
if well_comp == 2
    g1new = u0old(1) + Ki1*(-lambda_1 + lambda_c);
    u0pt(1,1) = max(1.0,min(7.5, g1new));
    g3new = u0old(3) + Ki3*(-lambda_3 + lambda_c);
    u0pt(3,1) = max(1.0,min(7.5, g3new));
```

```
            g2new = 7.5 - (uOpt(1,1) + uOpt(3,1));
            uOpt(2,1) = max(1.0,min(7.5, g2new));
        end
        if well_comp == 3
            g1new = u0old(1) + Ki1*(-lambda_1 + lambda_c);
            uOpt(1,1) = max(1.0,min(7.5, g1new));
            g2new = u0old(2) + Ki2*(-lambda_2 + lambda_c);
            uOpt(2,1) = max(1.0,min(7.5, g2new));
            g3new = 7.5 - (uOpt(1,1) + uOpt(2,1));
            uOpt(3,1) = max(1.0,min(7.5, g3new));
        end

        %---- Assign setpoints ----


%         uOpt(1,1) = 2.500;
%         uOpt(2,1) = 2.500;
%         uOpt(3,1) = 2.500;


        flagOpt = 1;

        if flagOpt == 1
            %%%%%%%%%%%%%%%%%%%%%%%%%%%
            %Send Variable
            %%%%%%%%%%%%%%%%%%%%%%%%%%%
            % compute new values for the gas flow rate setpoints
            %Filter for optimum
            fic104sp = uOpt(1);
            fic105sp = uOpt(2);
            fic106sp = uOpt(3);

            O_vector = vertcat(fic104sp,fic105sp,fic106sp)';
            SS = 1;
            Estimation = 1;
            Optimization = 1;
            Result = lambda;
            Parameter_Estimation = thetakk';
            State_Variables_Estimation = (par.H*zHatkk)';
%             State_Variables_Optimization = ([J_qgl1;J_qgl2;J_qgl3;L_qgl1;L_qgl2;L_qgl3;])';
            Optimized_Air_Injection = uOpt';
        else
            %%%%%%%%%
            %(dummy)%
            %%%%%%%%%
            % compute new values for the gas flow rate setpoints
            O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

            SS = 1;
            Estimation = 1;
            Optimization = 0;
            Result = 0;
            Parameter_Estimation = [0,0,0,0,0,0];
            State_Variables_Estimation = [0,0,0,0,0,0];
            State_Variables_Optimization = [0,0,0,0,0,0];
            Optimized_Air_Injection = [0,0,0];
        end

    else
```

```
        %%%%%%%%%
        %(dummy)%
        %%%%%%%%%
        % compute new values for the gas flow rate setpoints
        O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

        SS = 1;
        Estimation = 0;
        Optimization = 0;
        Result = 0;
        Parameter_Estimation = [0,0,0,0,0,0];
        State_Variables_Estimation = [0,0,0,0,0,0];
        State_Variables_Optimization = [0,0,0,0,0,0];
        Optimized_Air_Injection = [0,0,0];
    end


else
    %%%%%%%%%
    %(dummy)%
    %%%%%%%%%
    % compute new values for the gas flow rate setpoints
    O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

    SS = 0;
    Estimation = 0;
    Optimization = 0;
    Result = 0;
    Parameter_Estimation = [0,0,0,0,0,0];
    State_Variables_Estimation = [0,0,0,0,0,0];
    State_Variables_Optimization = [0,0,0,0,0,0];
    Optimized_Air_Injection = [0,0,0];

end
```

C.2 LabViewMain - Dual Decomposition

```
% Main program
% Run Initialization file first

%%%%%%%%%%%%%%%%
% Get Variables
%%%%%%%%%%%%%%%%
% disturbances
%valve opening [%]
cv101 = P_vector(1);
cv102 = P_vector(2);
cv103 = P_vector(3);
% if value is [A] from 0.004 to 0.020
% if you want to convert to 0 (fully closed) to 1 (fully open)
% vo_n = (vo - 0.004)./(0.02 - 0.004);

%pump rotation [%]
pRate = P_vector(4);
% if value is [A] from 0.004 to 0.020
% if you want to convert to (min speed - max speed)
% goes from 12% of the max speed to 92% of the max speed
% pRate = 12 + (92 - 12)*(P_vector(4) - 0.004)./(0.02 - 0.004);
```

```
% always maintain the inputs greater than 0.5
% inputs computed in the previous MPC iteration
% Note that the inputs are the setpoints to the gas flowrate PID's
fic104sp = P_vector(5);
fic105sp = P_vector(6);
fic106sp = P_vector(7);
%current inputs of the plant
u0old=[P_vector(5);P_vector(6);P_vector(7)];


% cropping the data vector
nd = size(I_vector,2);
dataCrop = (nd - BufferLength + 1):nd;

% liquid flowrates [L/min] %INCLUDE NOISE???
fi101 = I_vector(1,dataCrop);
fi102 = I_vector(2,dataCrop);
fi103 = I_vector(3,dataCrop);

% actual gas flowrates [sL/min] %INCLUDE NOISE???
fic104 = I_vector(4,dataCrop);
fic105 = I_vector(5,dataCrop);
fic106 = I_vector(6,dataCrop);

% pressure @ injection point [mbar g]
pi105 = I_vector(7,dataCrop);
pi106 = I_vector(8,dataCrop);
pi107 = I_vector(9,dataCrop);

% reservoir outlet temperature [oC]
ti101 = I_vector(10,dataCrop);
ti102 = I_vector(11,dataCrop);
ti103 = I_vector(12,dataCrop);

% DP @ erosion boxes [mbar]
dp101 = I_vector(13,dataCrop);
dp102 = I_vector(14,dataCrop);
dp103 = I_vector(15,dataCrop);

% top pressure [mbar g]
% for conversion [bar a]-->[mbar g]
% ptop_n = ptop*10^-3 + 1.01325;
pi101 = I_vector(16,dataCrop);
pi102 = I_vector(17,dataCrop);
pi103 = I_vector(18,dataCrop);

% reservoir pressure [bar g]
% for conversion [bar g]-->[bar a]
% ptop_n = ptop + 1.01325;
pi104 = I_vector(19,dataCrop);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CODE GOES HERE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% number of measurements in the data window
dss = size(pi104,2);

% we do no consider SS detection here
flagSS = 1;
```

```
% check for first iteration
if ~exist('dxHatkk','var')
    %initial condition
    [dx0,z0,u0,theta0] = InitialConditionGasLift(par);

    dxHatkk = dx0;
    zHatkk = z0;
    thetakk = theta0;

    load('EKFconf');
    qThres = ekf.qThresk;
    Pkk = ekf.Pkk;
    ekf.R = noise.output; %added (different from Rig Implementation)

    lambda = 9.5;
    err0 = zeros(3,1);
end

if flagSS == 1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Estimating SS model parameters (dynamic) %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    yPlant = [fi101;
              fi102;
              fi103;
              1.01325 + 10^-3*pi101; %[mbarg]-->[bar a];
              1.01325 + 10^-3*pi102;
              1.01325 + 10^-3*pi103];

    uPlant = [fic104; %conversion [L/min] --> [kg/s]
              fic105;
              fic106;
              cv101*ones(1,dss); %workaround - i just have the last measurement here. Since it is the distu
              cv102*ones(1,dss);
              cv103*ones(1,dss);
              pi104 + 1.01325];

          try
              % running casadi - getting the last measurement and last
              % input that generated that measuremnt
              [dxHatkk,zHatkk,thetakk,Pkk,qThres] = RExtendedKalmanFilter(yPlant(:,end),uPlant(:,end - 10),

              % everything normal
              flagEst = 1;
          catch
              warning('Dynamic Estimation Problem!');
              beep
              % estimation problem
              flagEst = 0;

              % Note that in this case, we dont update
              % dxHatkk,zHatkk,thetakk,Pkk,qThres
          end


    if flagEst == 1
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Optimizing Systems %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Update Lambda
Total_Qgl = fic104(end) + fic105(end) + fic106(end);

K_ccc = 0.907;
tau_ccc = 50;
theta_ccc = 0;
KI_ccc = 1/(K_ccc*(tau_ccc+theta_ccc));
tStart = cputime;

lambda_next =  max(0,lambda + KI_ccc*(Total_Qgl   - 7.5));

tEnd_lambda = cputime - tStart;

lambda = lambda_next;

% Gradient Estimation
%conversion
CR = 60*10^3; % [L/min] -> [m3/s]

tStart = cputime;
F_zp = full(S_zp(dxHatkk,zHatkk,[uSPPlantArray(1:3,end);uPlant(4:end,end);thetakk;1e4]));
tEnd_lambda = cputime - tStart + tEnd_lambda;
%%% NEED TO CHECK

J_qgl1 = - 20*((1*1e-2)*CR/par.rho_o(1))*F_zp(22,1);
J_qgl2 = - 25*((1*1e-2)*CR/par.rho_o(2))*F_zp(23,2);
J_qgl3 = - 30*((1*1e-2)*CR/par.rho_o(3))*F_zp(24,3);

g_qgl1 = 1;
g_qgl2 = 1;
g_qgl3 = 1;

L_qgl1 = J_qgl1 + lambda.*g_qgl1;
L_qgl2 = J_qgl2 + lambda.*g_qgl2;
L_qgl3 = J_qgl3 + lambda.*g_qgl3;

Lu(:,kk) = [L_qgl1;L_qgl2;L_qgl3];

tauC = 10;
tauI = min(4*tauC,800);

% --------- Well 1 ---------
K_1 = 2.53;
tauC_1 = 10;
theta_1 = 0;
Ki1 = 1/(K_1*tauC_1 + theta_1);
err_1= -L_qgl1;
tStart = cputime;
        uOpt(1,1) = max(1,min(5.5, uOold(1) + (Ki1*err_1)));
tEnd_gr1 = cputime - tStart;

err0(1) = err_1;
% --------- Well 2 ---------
K_2 = 2.93;
tauC_2 = 10;
theta_2 = 0;
Ki2 = 1/(K_2*tauC_2 + theta_2);
```

```
    err_2 = -L_qgl2;
        tStart = cputime;
            uOpt(2,1) = max(1,min(5.5, uOold(2) + (Ki2*err_2)));
        tEnd_gr2 = cputime - tStart;
    err0(2) = err_2;
     % --------- Well 3 ---------
    K_3 = 3.70;
    tauC_3 = 10;
    theta_3 = 0;
    Ki3 = 1/(K_3*tauC_3 + theta_3);
    err_3= -L_qgl3;
        tStart = cputime;
             uOpt(3,1) = max(1,min(5.5, uOold(3) + (Ki3*err_3)));
        tEnd_gr3 = cputime - tStart;
    err0(3) = err_3;

    flagOpt = 1;

    tEnd = max(max(tEnd_gr1,tEnd_gr2),tEnd_gr3)+ tEnd_lambda;

    if flagOpt == 1
        %%%%%%%%%%%%%%%%%%%%%%%%%%%
        %Send Variable
        %%%%%%%%%%%%%%%%%%%%%%%%%%%
        % compute new values for the gas flow rate setpoints
        %Filter for optimum
        fic104sp = uOpt(1);
        fic105sp = uOpt(2);
        fic106sp = uOpt(3);

        O_vector = vertcat(fic104sp,fic105sp,fic106sp)';
        SS = 1;
        Estimation = 1;
        Optimization = 1;
        Result = lambda;
        Parameter_Estimation = thetakk';
        State_Variables_Estimation = (par.H*zHatkk)';
        State_Variables_Optimization = ([J_qgl1;J_qgl2;J_qgl3;L_qgl1;L_qgl2;L_qgl3;])';
        Optimized_Air_Injection = uOpt';
    else
        %%%%%%%%%
        %(dummy)%
        %%%%%%%%%
        % compute new values for the gas flow rate setpoints
        O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

        SS = 1;
        Estimation = 1;
        Optimization = 0;
        Result = 0;
        Parameter_Estimation = [0,0,0,0,0,0];
        State_Variables_Estimation = [0,0,0,0,0,0];
        State_Variables_Optimization = [0,0,0,0,0,0];
        Optimized_Air_Injection = [0,0,0];
    end

else
    %%%%%%%%%
    %(dummy)%
```

```
        %%%%%%%%%
        % compute new values for the gas flow rate setpoints
        O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

        SS = 1;
        Estimation = 0;
        Optimization = 0;
        Result = 0;
        Parameter_Estimation = [0,0,0,0,0,0];
        State_Variables_Estimation = [0,0,0,0,0,0];
        State_Variables_Optimization = [0,0,0,0,0,0];
        Optimized_Air_Injection = [0,0,0];
    end


else
    %%%%%%%%%
    %(dummy)%
    %%%%%%%%%
    % compute new values for the gas flow rate setpoints
    O_vector = vertcat(fic104sp,fic105sp,fic106sp)';

    SS = 0;
    Estimation = 0;
    Optimization = 0;
    Result = 0;
    Parameter_Estimation = [0,0,0,0,0,0];
    State_Variables_Estimation = [0,0,0,0,0,0];
    State_Variables_Optimization = [0,0,0,0,0,0];
    Optimized_Air_Injection = [0,0,0];

end
```

C.3 Extended Kalman Filter

```
function [dxHatkk,zHatkk,pHatkk,Pkk,qThres] = RExtendedKalmanFilter(yk,uk_1,dxk_1k_1,zk_1k_1,p_k_1k_1,PNk_1
%    Uses an reduced extended Kalman filter for estimating the model states and parameters
%    We set the parameters qThres and ekf.lambdaThres such that the rEKF
%    calculations become equivalent to the EKF

% Inputs:
%    yk, uk_1 = measurements + system inputs
%    dxk_1k_1,zk_1k_1,p_k_1k_1 = previously estimated states and parameters
%    PNk_1k_1 = previously computed estimate covariance matrix
%    F = dynamic model integrator
%    S_xx,S_zz,S_xz,S_xp,S_zp = sensitivty equations
%    ekf = ekf configuration parameters
%    par = system parameters

% Outputs:
%   pHatkk = estimated parameters
%   dxHatkk, zHatkk = estimated states
%   Pkk = updated value of the estimate covariance
%   qThres  = number of parameters + states estimated --> not used here

% Other m-files required: none
% MAT-files required: none

% addpath ('C:\Users\lab\Documents\casadi-windows-matlabR2016a-v3.4.5')
```

```
import casadi.*

% ===================================
%     Integrating results
% ===================================
    % we use the nonlinear model to evolve the states
    Fend = F('x0',dxk_1k_1,'z0',zk_1k_1,'p',[uk_1;p_k_1k_1;par.T]);
    %extracting solution
    dxkk_1 = full(Fend.xf);
    zkk_1 = full(Fend.zf);

% ===================================
%     Sensitivitits
% ===================================
    %states
    F_xx = full(S_xx(dxk_1k_1,zk_1k_1,[uk_1;p_k_1k_1;par.T]));
    F_zz = full(S_zz(dxk_1k_1,zk_1k_1,[uk_1;p_k_1k_1;par.T]));
    F_xz = full(S_xz(dxk_1k_1,zk_1k_1,[uk_1;p_k_1k_1;par.T]));
    F_zx = F_xz';

    FSk = [F_xx, F_xz; F_zx, F_zz];
    HSk = [zeros(length(yk),length(dxk_1k_1)),par.H];

    %inputs
    F_xp = full(S_xp(dxk_1k_1,zk_1k_1,[uk_1;p_k_1k_1;par.T]));
    F_zp = full(S_zp(dxk_1k_1,zk_1k_1,[uk_1;p_k_1k_1;par.T]));
    FPk = [F_xp;F_zp];
    FPk(:,1:length(uk_1)) = [];%excluding derivative in relation to feed inputs
    FPk(:,end) = [];%excluding derivative in relation to par.T

% ===================================
%     Filter
% ===================================
    %Arranging filter matrices - extended state vector
    Fk_1 = [FSk, FPk; zeros(length(p_k_1k_1),(length(dxk_1k_1) + length(zkk_1))), eye(length(p_k_1k_1))];

    %output transition
    Hk = [HSk, zeros(length(yk),length(p_k_1k_1))];

    %preparing extended vector
    xkk_1 = [dxkk_1; zkk_1; p_k_1k_1];

    %%%%%%%%%%%%%%%%%%%%
    % Filter Equations %
    %%%%%%%%%%%%%%%%%%%%
    %predicted covariance estimate
    Pkk_1 = Fk_1*PNk_1k_1*Fk_1'+ ekf.Qe;

    %%
    % N.B.: if EKF is used instead of rEKF, this section is not relevant
    %Reduced Filtering - spectral deconposition
    [V,D] = eig(Pkk_1);

    %main diagonal - abs value of eigenvalues
    d = diag(D);

    %sorting eigenvalues - descend
    %[e,i] = sort(d,'descend');
    [e,i] = sort(d);
```

```
%initializing the threshold. The initial value is equal to the total number
%of states + parameters
qThres = size(ekf.Qe,2);

%initializing counter
l = 1;
while l <= length(e)
    if e(l) > ekf.lambdaThres
        %checking if the eigenvalue is greater than the threshold. If it
        %is, the number of directions used changes
        qThres = l;
        l = length(e);%exiting the loop
    end
    l = l + 1;
end

%obtaining reduced covariance matrix
temp = zeros(size(Pkk_1,1));
for j = 1:qThres
    temp = temp + e(j)*V(:,i(j))*V(:,i(j))';
end
Pkk_1r = temp;

temp = [];

%%
%Updating
%Innovation covariance
Sk = Hk*Pkk_1r*Hk' + ekf.R;

%Kalman Gain
Kk = Pkk_1r*Hk'*pinv(Sk);

%Updating covariance estimate
%Galo's upload system! Reduced Matrix is used only in the gain calculations
Pkk = (eye(length(dxk_1k_1) + length(zk_1k_1) + length(p_k_1k_1)) - Kk*Hk)*Pkk_1*((eye(length(dxk_1k_1)

%guaranteeing symmetric matrix
temp = Pkk;

for i = 1:size(temp,1)
    for j = (i + 1):size(temp,2)
        temp(j,i) = Pkk(i,j);
    end
end
Pkk = temp;

%Measurement residual
yHat = Hk*xkk_1;

%Update state estimate
xHatkk = xkk_1 + Kk*(yk - yHat); %state variables are normalized

% ==================================
%     Results
% ==================================
%dividing extended vector
dxHatkk = xHatkk(1:length(dxk_1k_1));%differential
```

```
    zHatkk = xHatkk(length(dxk_1k_1) + 1:length(dxk_1k_1) + length(zk_1k_1));%algebraic
    pHatkk = xHatkk(length(dxk_1k_1) + length(zk_1k_1) + 1:end);%parameters

end
```

C.4 Dynamic Model of Lab Rig

```
function [F,S_xx,S_zz,S_xz,S_xp,S_zp,x_var,z_var,u_var,p_var,diff,alg,L] = ErosionRigDynModel(par)
%    Creates a dynamic model of the rig and computes the sensitivity
%    matrices for EKF

% Inputs:
%    par = system parameters
%
% Outputs:
%   F: system integrator
%   S's: system sensitivities
%   x_var,z_var,u_var,p_var, diff,alg,L: Model in CasADi form

% Other m-files required: none
% MAT-files required: none

addpath('C:\Users\Vegard\Documents\casadi-windows-matlabR2016a-v3.5.5')
import casadi.*

%% Parameters
%number of wells
n_w = par.n_w; %[]
%gas constant
R = par.R; %[m3 Pa K^-1 mol^-1]
%air molecular weigth
Mg = par.Mw; %[kg/mol] -- Attention: this unit is not usual

%properties
%density of water - dim:  nwells x 1
rho_l = par.rho_o; %[kg/m3]
%mixture viscosity
mu_mix = par.mu_oil;% [Pa s]

%project - dim:  nwells x 1
% well length
L_w = par.L_w; %[m]
% well pipes cross section area
A_w = par.A_w;%[m2]

% riser length
L_r = par.L_r; %[m]
% riser pipes cross section area
A_r = par.A_r;%[m2]
%riser height
H_r = par.H_r; %[m]
%well below injection
D = par.D_bh; %[m]

%% System states
% differential
%gas holdup
m_g = MX.sym('m_g',n_w);            % 1:3 [1e-4 kg]
```

```
%water holdup
m_l = MX.sym('m_l',n_w);              % 4:6 [kg]

% algebraic
%water rate from reservoir
w_l = MX.sym('w_l',n_w);              % 1:3 [1e-2 kg/s]
%total well production rate
w_total = MX.sym('w_total',n_w);      % 4:6 [1e-2 kg/s]

%riser head pressure
p_rh = MX.sym('p_rh',n_w);            % 7:9 [bar]
%pressure - before injection point (bottom hole)
p_bi = MX.sym('p_bi',n_w);            % 10:12 [bar]

%mixture density in system
rho_mix =  MX.sym('rho_mix',n_w);     % 13:15 [1e2 kg/m3]
%density gas
rho_g= MX.sym('rho_g',n_w);           % 16:18 [kg/m3]

%well outlet flowrate (gas)
w_gout = MX.sym('w_gout',n_w);        % 19:21 [1e-5 kg/s]
%riser head gas production rate gas
w_lout = MX.sym('w_lout',n_w);        % 22:24 [1e-2 kg/s]

%% System input
%gas lift rate
Q_gl = MX.sym('Q_gl',n_w);          % 1:3 [sL/min]
%valve oppening
vo = MX.sym('vo',n_w);              % 4:6 [0-1]
%pump outlet pressure
Ppump = MX.sym('Ppump',1);          % 7 [bar]

%% parameters
%%%%%%%%%
% fixed %
%%%%%%%%%
%room temperature
T = MX.sym('T',1); %[K]
%atmospheric pressure
p_atm = MX.sym('p_atm',1); %[bar]

%time transformation: CASADI integrates always from 0 to 1 and the USER does the time
%scaling with T --> sampling time
t_samp = MX.sym('t_samp',1); %[s]

% estimable
%scaled reservoir valve parameters
res_theta = MX.sym('res_theta',n_w);
%scaled top valve parameters
top_theta = MX.sym('top_theta',n_w);

%% Modeling
% Algebraic
%conversion
CR = 60*10^3; % [L/min] -> [m3/s]
%reservoir outflow
f1 = -Ppump*ones(n_w,1)*1e5 + (w_l.*1e-2).^2.*(res_theta.*1e9)./(vo.^2.*rho_l) + p_bi.*1e5 ;
% total system production
f2 = - (w_total.*1e-2) + ((w_gout.*1e-5) + (w_lout.*1e-2));
```

```
%riser head pressure
f3 = -p_rh.*1e5 + (w_total.*1e-2).^2.*(top_theta.*1e8)./(rho_mix.*1e2) + p_atm.*1e5 ;
%before injection pressure
f4 = -p_bi.*1e5 + (p_rh.*1e5 + (rho_mix.*1e2).*9.81.*H_r + 128.*mu_mix.*(L_w+L_r).*(w_l.*1e-2)./(3.14.*D.^4
%mixture density
f5 = -(rho_mix.*1e2) + (((m_g.*1e-4) + m_l).* p_bi.*1e5.*Mg.*rho_l)./(m_l.*p_bi.*1e5.*Mg + rho_l.*R.*T.*(m_
%gas density (ideal gas law)
f6 = -rho_g + p_bi.*1e5.*Mg/(R*T);

% Simplifying assumption!
% liquid fraction in the mixture
xL = (m_l./((m_g.*1e-4) + m_l));
%Liquid outlet flowrate
f7 = -(w_lout.*1e-2) + xL.*(w_total.*1e-2);

% Total volume constraint
f8 = -(A_w.*L_w + A_r.*L_r) + (m_l./rho_l + (m_g.*1e-4)./rho_g);

% Differential
% gas mass balance
df1=  1e4*(-(w_gout.*1e-5) + Q_gl.*rho_g/CR);
% liquid mass balance
df2= -(w_lout.*1e-2) + (w_l.*1e-2);

% Form the DAE system
diff = vertcat(df1,df2);
alg = vertcat(f1,f2,f3,f4,f5,f6,f7,f8);

% give fixed parameter values
alg = substitute(alg,p_atm,par.p_s);
alg = substitute(alg,T,par.T_r);

% concatenate the differential and algebraic states
x_var = vertcat(m_g,m_l);
z_var = vertcat(w_l,w_total,p_rh,p_bi,rho_mix,rho_g,w_gout,w_lout);
u_var = vertcat(Q_gl,vo,Ppump);
p_var = vertcat(res_theta,top_theta,t_samp);

%objective function
L = 20*((w_lout(1)*1e-2)*CR/rho_l(1)) + 25*((w_lout(2)*1e-2)*CR/rho_l(2)) + 30*((w_lout(3)*1e-2)*CR/rho_l(3
%end modeling

%% Casadi commands
%declaring function in standard DAE form (scaled time)
dae = struct('x',x_var,'z',z_var,'p',vertcat(u_var,p_var),'ode',t_samp*diff,'alg',alg);

%calling the integrator, the necessary inputs are: label; integrator; function with IO scheme of a DAE (for
F = integrator('F','idas',dae);

% ================================================
%       Calculating sensitivity matrices
% ================================================

S_xx = F.factory('sensStaStates',{'x0','z0','p'},{'jac:xf:x0'});
S_zz = F.factory('sensStaStates',{'x0','z0','p'},{'jac:zf:z0'});
S_xz = F.factory('sensStaStates',{'x0','z0','p'},{'jac:xf:z0'});

S_xp = F.factory('sensParStates',{'x0','z0','p'},{'jac:xf:p'});
S_zp = F.factory('sensParStates',{'x0','z0','p'},{'jac:zf:p'});
```

```
end
```

C.5 Main

```
% Runs a mock-up ROPA problem, where the model and the system are equal and the disturbance setup is pre-de
% The system setup is based on the paper:
%   Implementation of Steady-state Real-time Optimization Using Transient Measurements on Experimental Rig
%   by Matias et al.

% Other m-files required: SmoothPlantParameters2.mat; Disturbances2.mat; RigNoise.mat; InitialState_2021-03
% MAT-files required:
%       For ROPA:
%           2. RExtendedKalmanFilter.m
%           3. ErosionRigSSOptimization.m

%       For Mockup Labview Interface:
%           1. InitializationLabViewMain.m
%           2. LabViewMain.m

%       For "Plant":
%           1. ErosionRigDynModel.m

%       Bounds, Parameters and Initial conditon:
%           1. InitialConditionGasLift.m
%           2. OptimizationBoundsGasLiftRiser2.m
%           3. ParametersGasLiftModel.m
clear
%close all
clc

%saving data
name = 'ROPA_DT_Cycle';
%noise seed
rng('default')

%% Loading .mat files
% previously defined disturbance profiles
%disturbances = load('Disturbances3');
%disturbances = load('DisturbancesLonger');
disturbances = load('disturbances_updown');

% previously computed system parameter profiles - used as "plant" true
% parameters
%parProfile = load('SmoothPlantParametersLonger');
parProfile = load('SmoothPlantParameters3');

% rig noise characteristics - compute previously from actual rig data
noise = load('RigNoise');

% Take out Noise
noise.input = 1.*noise.input;
noise.output = 1.*noise.output;

%% Simulation tuning
% Buffer length
BufferLength = 60; %[s]
```

```
% Optimization sampling time
nExec = 2; %[s]% Requested by Sigurd to make faster shadow prices update (EKF is not suggested to faster th

% nExec = 10; %[s]% Initial

%plant parameters
parPlant = ParametersGasLiftModel;

%simulation parameters
nInit = 0; %[s]
%nFinal = 1*size(parProfile.thetaPlant,2); %[s] ! Adding a 60 second buffer in comparison to experimental d
nFinal = size(disturbances.values,2)-1;
parPlant.T = 1; %rig measurements samplint time[s]
tgrid = (nInit:parPlant.T:nFinal)/60; %[min] one measurements per second

%initial condition
[dxPlant0,zPlant0,uPlant0,thetaPlant0] = InitialConditionGasLift(parPlant);

%states to measurement mapping function
parPlant.nMeas = 6;
parPlant.H = zeros(6,24);
parPlant.H(1,1) = 1e-2*60*1e3/parPlant.rho_o(1); %wro-oil rate from reservoir, well 1 [1e-2 kg/s] --> [L/mi
parPlant.H(2,2) = 1e-2*60*1e3/parPlant.rho_o(2); %wro-oil rate from reservoir, well 2
parPlant.H(3,3) = 1e-2*60*1e3/parPlant.rho_o(3); %wro-oil rate from reservoir, well 3
parPlant.H(4,7) = 1; %prh - riser head pressure well 1
parPlant.H(5,8) = 1; %prh - riser head pressure well 2
parPlant.H(6,9) = 1; %prh - riser head pressure well 3

% Model representing the rig
[F,~,~,~,~,~,~,~,~,~,~,~,~] = ErosionRigDynModel(parPlant);

% mockup controller dynamic parameters
tau_C = 1;

%% Run configuration file
InitializationLabViewMain %here we use the same syntax as in the rig

%% Initializing simulation
% Plant states
dxk = dxPlant0;
zk = zPlant0;

% Inputs
uk = [uPlant0(1);                   %FIC-104 [sl/min]
      uPlant0(2);                   %FIC-105 [sl/min]
      uPlant0(3);                   %FIC-106 [sl/min]
      disturbances.values(1,1);     %CV-101 opening [-]
      disturbances.values(2,1);     %CV-102 opening [-]
      disturbances.values(3,1);     %CV-103 opening [-]
      disturbances.values(4,1)];    %PI-104 [bar]

% Setpoints for the gas lift controllers
O_vector = [uk(1); uk(2); uk(3)];

% "Plant" parameters
thetak = thetaPlant0;

%% Run mock-up loop
% arrays for plotting
```

```
%%%%%%%%%%%%%%%%
% "Plant" Data %
%%%%%%%%%%%%%%%%
xPlantArray = dxk;
zPlantArray = zk;
uPlantArray = uk;
uSPPlantArray = O_vector; % decision variable of the optimization problem are the SP of the PI controllers
measPlantArray = parPlant.H*zk;
thetaPlantArray = thetak;

%%%%%%%% - THIS NEEDS TO CHECK!!!
%ofPlantArray = 20*(measPlantArray(1)) + 10*(measPlantArray(2)) + 30*(measPlantArray(3));
ofPlantArray = 20*(measPlantArray(1)) + 25*(measPlantArray(2)) + 30*(measPlantArray(3));
%%%%%%%% - THIS NEEDS TO CHECK!!!

State_Variables_Optimization = zeros(1,6);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Production Optimization Methods Data %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% we save the same data that is saved in the Rig's labview code
flagArray = []; % flag SSD, Model Adaptation and Economic Optimization [flag == 0 (failed), == 1 (success)]
ofArray = [];   % OF value computed bu the Economic Optimization
thetaHatArray = []; % estimated parameters
yEstArray = []; % model prediction with estimated states
yOptArray = []; % model prediction @ new optimum
uOptArray = []; % computed inputs (u_k^\star)
uImpArray = []; % filtered inputs to be implemented (u_{k+1})

compuTimeArray = []; % computational time

for kk = 1:nFinal

    % printing the loop evolution in minutes
    fprintf('     kk >>> %6.4f [min]\n',tgrid(kk + 1))

    % integrating the "plant" (the model sampling time is defined by parPlant.T)
    Fend = F('x0',dxk,'z0',zk,'p',[uk;thetak;parPlant.T]);

    %extracting the results (from Casadi symbolic to numerical)
    dxk = full(Fend.xf);
    zk = full(Fend.zf);

    % saving the results
    xPlantArray = [xPlantArray, dxk];
    zPlantArray = [zPlantArray, zk];
    measPlantArray = [measPlantArray, parPlant.H*zk + noise.output*randn(6,1)]; %  adding artificial noise

    %%%%%%%% - THIS NEEDS TO CHECK!!!
    %ofPlantArray = [ofPlantArray, 20*(measPlantArray(1,end)) + 10*(measPlantArray(2,end)) + 30*(measPlantA
    ofPlantArray = [ofPlantArray, 20*(measPlantArray(1,end)) + 25*(measPlantArray(2,end)) + 30*(measPlantAr
    %%%%%%%% - THIS NEEDS TO CHECK!!!

    % we execute the production optimization:
    % a. after the initial [BufferLength]-second buffer
    % b. every [nExec] seconds
    if kk > BufferLength && rem(kk,nExec) == 0
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Rearranging the data vectors and units here, so they can the be exactly %
```

```
        % the same as in the actual rig. The goal is that LabViewRTO.m can be     %
        % directly plug in the Labview interface and it will work                  %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % measurement buffer (dim = 19 X BufferLength)
        I_vector = [measPlantArray(1,(kk - BufferLength + 2):kk + 1); % FI-101 [L/min]
                    measPlantArray(2,(kk - BufferLength + 2):kk + 1); % FI-102 [L/min]
                    measPlantArray(3,(kk - BufferLength + 2):kk + 1); % FI-103 [L/min]
                    uPlantArray(1,(kk - BufferLength + 1):kk);    % FI-104 [sL/min]
                    uPlantArray(2,(kk - BufferLength + 1):kk);    % FI-105 [sL/min]
                    uPlantArray(3,(kk - BufferLength + 1):kk);    % FI-106 [sL/min]
                    ones(3,BufferLength); %dummy values --> in the actual rig, they will the pressure at in
                    ones(3,BufferLength); %dummy values --> in the actual rig, they will the well temperatu
                    ones(3,BufferLength); %dummy values --> in the actual rig, they will be dP in a given p
                    (measPlantArray(4,(kk - BufferLength + 2):kk + 1) - 1.01325)*10^3; % PI-101 [mbar g]
                    (measPlantArray(5,(kk - BufferLength + 2):kk + 1) - 1.01325)*10^3; % PI-102 [mbar g]
                    (measPlantArray(6,(kk - BufferLength + 2):kk + 1) - 1.01325)*10^3; % PI-103 [mbar g]
                    uPlantArray(7,(kk - BufferLength + 1):kk) - 1.01325];     % PI-104 [bar g]


        % values of the input variables at the previus rig sampling time (dim = nu[7] X 1)
        P_vector = [uk(4); % CV101 opening [-]
                    uk(5); % CV102 opening [-]
                    uk(6); % CV103 opening [-]
                    1;%dummy values --> in the actual rig, they will the pump rotation. Not used here
                    uSPPlantArray(1,end); % FI-104sp [sL/min]
                    uSPPlantArray(2,end); % FI-105sp [sL/min]
                    uSPPlantArray(3,end)]; % FI-106sp [sL/min]
        % values of the inputs (gas lift) of the last optimization run (dim = nQg[3] X 1)
        O_vector = uSPPlantArray(:,kk - nExec);

        tic

        % Run Labview/Matlab interface file
        LabViewMain

        %compuTimeArray = [compuTimeArray, toc];
%        compuTimeArray = [compuTimeArray, tEnd];

        flagArray = [flagArray, [SS;Estimation;Optimization]];
        ofArray = [ofArray, Result];
        thetaHatArray = [thetaHatArray, Parameter_Estimation'];
        yEstArray = [yEstArray, State_Variables_Estimation'];
        yOptArray = [yOptArray, State_Variables_Optimization'];
        uOptArray = [uOptArray, Optimized_Air_Injection'];

        uImpArray = [uImpArray, O_vector'];

    else
         % update with dummy values
         flagArray = [flagArray, [0;0;0]];
         ofArray = [ofArray, 0];
         thetaHatArray = [thetaHatArray, zeros(1,6)'];
         yEstArray = [yEstArray, zeros(1,6)'];
         yOptArray = [yOptArray, State_Variables_Optimization'];
         uOptArray = [uOptArray, zeros(1,3)'];

         uImpArray = [uImpArray, zeros(1,3)'];
     end
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % updating input and parameter vectors %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %1. Saving setpoints
    uSP = [O_vector(1);  %FIC-104 SP [sl/min]
           O_vector(2);  %FIC-105 SP [sl/min]
           O_vector(3)]; %FIC-106 SP [sl/min]
    uSPPlantArray = [uSPPlantArray, uSP];

    %2. Adding "controller" action
    % instead of coding a controller, we simply add a 5s delay + input noise
    if kk > 5
        uImple = uImple + (uSPPlantArray(:,kk - 5) - uImple + noise.input*randn(3,1))*exp(-parPlant.T/tau_C
    else
        uImple = uSPPlantArray(:,kk) + noise.input*randn(3,1);
    end

    %3. Saving actual implemented values
    uk = [uImple;
        disturbances.values(:,kk + 1)]; % for obtaining the full plant input vector, we need to add the val

      uPlantArray = [uPlantArray, uk];

     %4. Updating plant parameters according to pre-computed array
     thetak = thetaPlant0;
     thetaPlantArray = [thetaPlantArray, thetak];
end

save(name,'flagArray','ofArray','ofPlantArray','thetaHatArray','yEstArray','yOptArray','uOptArray','uImpArr
```